FAKULTÄT FÜR !NFORMATIK

FROM LEGACY WEB APPLICATIONS TO WEBML MODELS

A Framework-based Reverse Engineering Process

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Wirtschaftsinformatik

ausgeführt von

Max Rieder

Matrikelnummer 0126507

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung:
Betreuerin: O.Univ.Prof. Mag. Dipl.-Ing. Dr.techn. Gertrude Kappel
Mitwirkung: Univ.Ass. Mag. Dr.rer.soc.oec. Manuel Wimmer

Wien, 01.12.2009

(Unterschrift Verfasser)

(Unterschrift Betreuer)

Erklärung zur Verfassung der Arbeit

Max Rieder
Tautenhayngasse 15/2/2303
1150 Wien

„Hiermit erkläre ich, dass ich diese Arbeit selbstständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel voll angegeben habe und dass ich die Stellen der Arbeit einschließlich Tabellen, Karten und Abbildungen , die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.“

Wien, 07. Dezember 2009

Acknowledgements

I would like to thank all the people that supported me during my studies and during the time of writing my master thesis. These are especially my parents, Wilfried and Rosemarie Rieder, who gave me a lot of financial and moral support. I also want to thank Dr. Manuel Wimmer, who helped me with technical issues and the principles of scientific writing, as well as Prof. Dr. Gerti Kappel, for supervising my work.

Abstract

In the last decade the adoption of web applications instead of desktop applications has grown rapidly. Also the patterns and technologies for developing and running web applications have changed a lot over time. The World Wide Web has evolved from a collection of linked static documents to a space of countless dynamic, data centric applications. One of the oldest and most popular languages for developing dynamic web applications is PHP. Although nowadays there are proved techniques for developing web applications in PHP, many older PHP web applications are written without the notion of applying well-defined design patterns. Those web applications are hard to understand, maintain, extend as well as hard to migrate to new web platforms.

Nowadays many web applications are developed using Model Driven Engineering (MDE) techniques where software systems are described as models and code artifacts are generated out of these models. But often the requirement is not to develop a completely new web application but to capture the functionality of an existing legacy application. As it usually takes a lot of time for humans to understand the source code, it can be helpful to have a tool that analyzes the source artifacts and transforms them into a model on a higher level of abstraction. This process is called reverse engineering. The requirements for such a tool to work is the existence of well-known patterns in the source code, which is typically found in Model-View-Controller (MVC) web applications.

In this thesis a reverse engineering process from a legacy PHP web shop application into a model of the Web Modeling Language (WebML), based on static code analysis, is presented. First of all the requirements for the source code are analyzed in order to apply an automatic reverse engineering process on it. The source application is refactored to fulfill these requirements, which leads to a MVC version of the example application. The refactored application is the source for the next step, a code to model transformation into an intermediate model of the MVC web application. The last step is a model to model transformation from the the MVC model into a WebML model.

The result is a WebML model that shows the most important structural and behavioral aspects of the example application. The benefit of such a model is that that it provides a realistic documentation of the current state of the application. Whenever the application changes, the process can be repeated so the documentation never gets outdated. It helps humans to understand the connections between different parts of the application and can be used to support refactoring activities or the migration to another platform.

Kurzfassung

In den letzten Jahren ist der Einsatz von Webanwendungen als Ersatz für Desktop Anwendungen rapide angestiegen. Auch die Entwurfsmuster und die Technologien, die zur Entwicklung und zum Betrieb von Webanwendungen verwendet werden, haben sich im Laufe der Zeit geändert. Das World Wide Web hat sich von einer Sammlung aus statischen, verlinkten Dokumenten zu einem Netz aus unzähligen dynamischen Applikationen für unterschiedlichste Zwecke entwickelt. Eine der ältesten und auch beliebtesten Sprachen zur Entwicklung von Webanwendungen ist PHP. Obwohl es heutzutage erprobte Techniken zur Entwicklung von PHP Anwendungen gibt, wurden viele ältere PHP Anwendungen ohne dem Einsatz von Entwurfsmustern geschrieben. Solche Webanwendungen sind schwierig zu verstehen, zu warten, zu erweitern und umzuschreiben.

Heutzutage werden viele Web Anwendungen unter Einsatz von Techniken des Model Driven Engineerings (MDE) entwickelt, wo Software Systeme als Modelle beschrieben werden und Code Artefakte aus diesen Modelle generiert werden. Doch häufig ist es nicht gefordert, eine komplett neue Web Anwendung zu entwickeln, sondern die Funktionalität einer existierenden Legacy Applikation zu erfassen. Da ein Mensch üblicherweise lange braucht, um den Quellcode zu verstehen, kann ein Tool hilfreich sein, das den Quellcode analysiert und in ein Model auf einer höheren Abstraktionsebene transformiert. Diesen Vorgang nennt man Reverse Engineering. Dafür müssen wohldefinierte Muster im Quellcode vorhanden sein, wie man sie typischerweise in Model-View-Controller (MVC) Web Anwendungen findet.

In dieser Arbeit wird ein Reverse Engineering Prozess von einer Legacy PHP Webshop Anwendung in ein Model der Web Modeling Language (WebML), basierend auf statischer Codeanalyse, vorgestellt. Zuerst werden die Anforderungen analysiert, die der Quellcode erfüllen muss, um überhaupt einen automatischen Transformationsprozess darauf anwenden zu können. Die Beispielapplikation wird einem Refactoring unterzogen, das zu einer MVC Webanwendung führt. Der nächsten Schritt ist eine Model to Code Transformation in eine Zwischenmodell der MVC Web Anwendung. Der letzte Schritt ist eine Model to Model Transformation des MVC Modells in ein WebML Modell.

Das Ergebnis ist ein WebML Modell, das die wichtigsten Aspekte der Struktur und des Verhaltens der Beispielanwendung darstellt. Der Nutzen so eines Modells ist, dass es eine realistische Dokumentation über den aktuellen Zustand der Applikation darstellt. Sobald sich die Applikation ändert, kann der Prozess wiederholt werden, so dass die Dokumentation niemals veraltet. Das Modell hilft Menschen, die Zusammenhänge der verschiedenen Anwendungsteile zu verstehen und kann für Refactoringzwecke oder für die Migration auf eine neue Plattform verwendet werden.

Table of Contents

Chapter 1

Introduction

1.1 Motivation and Goal of this Thesis

Since the early days of the World Wide Web, websites have evolved from simple collections of HTML pages presenting static content to dynamic applications that are able to interact with the user and to generate dynamic content [19]. A dynamic web application is based on one or more data sources (usually a relational database, although others sources such as web services or semantic web data are possible). To operate on the data provided by a data source web applications have to somehow implement basic CRUD (create, read, update and delete) operations.

As web applications have a client-server architecture, where the browser communicates with the web server via the stateless HTTP protocol, described in RFC 2616 [13], the application logic is placed on the web server and the browser is simply a thin client, mainly responsible for rendering the user interface, reacting on user triggered events and to send and receive HTTP messages (although it is possible to execute application logic in the browser, e.g by using JavaScript, described in the ECMAScript Language Specification [11]).

Due to the stateless nature of the HTTP protocol, a web application not only has to implement the CRUD operations but also has to take care about session handling and request/response parameter parsing. Nowadays there are numerous programming languages and frameworks capable or especially dedicated to implement web applications. One of the oldest and most popular scripting language for web application development is PHP.

While PHP is easy to learn and provided with many features required for web applications out of the box, it comes with certain drawbacks, such as that it is hard to debug and to refactor (as it is not a statically typed language) and that it encourages a bad programming style. Still many productive web applications nowadays are written in PHP. There are different approaches to measure the popularity of programming languages. O'Reilly Radar [20] presents a diagram with the relative share trends of books on programming language sold by the publishing company O'Reilly, where sales on books about PHP slightly oscillate between eight and ten percent, between 2003 and 2006. TIOBE Software [53] presents a monthly index indicating the popularity of programming languages, based on ratings calculated using popular search engines. According to the TIOBE index [52] PHP has a popularity of 9.921% in May 2009. In both statistics Java [37] is the most popular programming language with 19.537% in [52] in May 2009. As those statistics are considering programming languages for all purposes and not especially for web applications, it can be assumed that the share of PHP based web applications amongst all web applications is even higher.

As business requirements change over time, it is inevitable that legacy IT systems have to be adapted to the new requirements. This is not only true for COBOL applications written in the 1960s but also for web applications written in the 1990s and in this century. In his book, W. Ulrich [54] has identified the following typical characteristics for legacy application architecture:

- Humans cannot understand how the system functions.
- The system is hard to modify with confidence that a given change is correct.
- Business logic is hard to distinguish from logic that controls data access, user interface and environmental management functions.
- Business logic is redundantly and inconsistently defined within and across systems.
- The system lacks functional or technical documentation, or both.
- It is difficult to integrate the system with other systems not built under the same architecture.

One approach to reduce the impact of those characteristics is to leverage modeling techniques. For a newly built system it is possible to start with modeling different aspects of the system and then generate the system out of the model. But what about existing legacy systems where the above mentioned characteristics apply to? In order to obtain a model of such a system, some kind of reverse engineering process has to be performed. The aim of this thesis is to provide an example of how reverse engineering can be a applied on a poorly designed web shop application, written in PHP. It is analyzed which structures can be easily automatically reverse engineered, which parts need to be re engineered by hand and which parts can not be represented in the target model at all. The modeling language chosen for the target model is the web modeling Language (WebML). It is described in Ceri et al. [5] and on the WebML website [24]. Its advantages over other modeling languages is that it is especially designated to the modeling of web applications, that it consists of modeling elements reflecting typical functionality of data intensive web applications, that a WebML model is is easy to understand and that it provides a commercial tool support.

1.2 Structure of this Thesis

This thesis consists of ten chapters. In Chapter 2 the example application, which is the source for the reverse engineering process is described. In Chapter 3 an overview of the whole reverse engineering process is given. Chapter 4 analyzes, which requirements the example application must fulfill in oder to apply an automatic transformation process on it. The application is refactored according to these requirements. In Chapter 5 the necessary ingredients for the conceptual design of the reverse engineering process are described and in Chapter 6 the conceptual design is presented. In Chapter 7 the necessary ingredients for the implementation of the reverse engineering process are described and in Chapter 8 the implementation is presented. In Chapter 9 and overview of related work is given. In Chapter 10 the result of the reverse engineering process is analyzed and possibilities for future work are outlined.

Chapter 2

The Example Application

In this Chapter the example application, which is the starting point of the reverse engineering process, is described. In Section 2.1 the technologies used to run the application are described. In Section 2.2 the example application is presented from a user's point of view. In Section 2.3 the implementation of the application is shown.

2.1 Used Technology: LAMP

LAMP is an acronym for Linux, Apache, MySQL and PHP and describes the combined application of these technologies. This bundle of freely available open source software is very often used as a platform for real world web applications, as well as for the example application presented in this thesis. The technologies used are:

Linux "Linux is a free Unix-type operating system originally created by Linus Torvalds with the assistance of developers around the world. Developed under the GNU General Public License , the source code for Linux is freely available to everyone." [46]

Apache The Apache httpd project is an open-source HTTP server for UNIX and Windows based operation systems.

MySQL A relational database management system (RDBMS), which is described in [40]. The MySQL dump tool [38], bundled with the RDBMS, is used to retrieve the SQL create table statements used for the database of the example application.

PHP A scripting language for web applications. A more detailed overview is given in the remaining part of this Section.

PHP

In the early days of the World Wide Web, websites were mainly a collection of simple HTML pages with static content, linked amongst each other. Each page was a text file in the file system of the web server's machine and the web server simple returned the content of those files to requesting clients.

With the introduction of the Common Gateway Interface it became possible for the web server to interact with other applications. Hence the content presented to the client was no longer limited to hard coded text files but could be generate dynamically, e.g. by executing a Perl script. Clients were now able to influence the behavior of the website by providing form inputs, which were interpreted on the server side. But still there was no programming language which satisfied the special requirements for dynamic web applications. In 1995 Rasmus Lerdorf wrote a set of Perl scripts and later a C implementation of a program called ”Personal Home Page Tools”. The program enabled the development of simple dynamic Web applications providing database communication, Perl-like variables, automatic interpretation of form variables and HTML embedded syntax. Rasmus released the source code for everybody under the name PHP/FI, which stood for Personal Home Page / Forms Interpreter. PHP/FI 2.0 was released in November 1997 but was shortly after succeeded by the official release of PHP 3.0 in June 1998, which closely resembles PHP as we know it today. PHP 3 is a complete rewrite of the original language implementation, written by Andi Gutmans and Zeev Suraski, which provides a solid infrastructure for lots of different databases, protocols and APIs, as well as strong extensibility features. The name was changed to simply PHP as the recursive acronym for PHP: Hypertext Preprocessor, in order to remove the implication of limited personal use. PHP 3 enabled the development of complex web applications, but the implementation was not designed to handle such applications efficiently. Therefore Andi Gutmans and Zeev Suraski rewrote the core implementation of PHP to improve performance under the name Zend Engine (comprised of their first names, Zeev and Andi). PHP 4 is based on this engine and was released in May 2000. In addition to improved performance, PHP 4 provides support for more web servers, HTTP sessions, output buffering, more secure ways of handling user input and several new language constructs. PHP 5 was released in July 2004 based on its new core, the Zend Engine 2.0 with a new object model and dozens of other new features.

Table 2.1 gives a short overview of the PHP functions and language elements used in the example application. A detailed description of all functions can be found in the PHP Function List [49]. The object oriented features of PHP are described in the PHP Manual [47].

2.2 A functional Description from a User’s Perspective

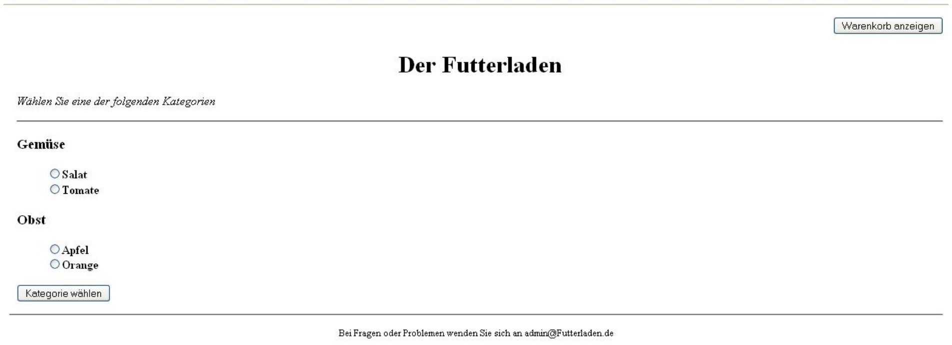
The example to be reverse engineered is a simple shopping cart application, based on the example in the German version of the book PHP and MySQL For Dummies by J. Valade [56]. It’s a shop for online purchasing of food. The entry point for the customer is the shop catalog page, shown in Figure 2.1, which displays the available product categories: Fruits and vegetables. The categories are further subdivided into sub categories: Vegetables can be salad or tomato, fruit can be apple or orange. The user may choose one subcategory via a radio button and a click on the `Choose category` button. Furthermore there is a button to reach the shopping cart.

A click on the button to view the shopping cart leads the user to the page shown in Figure 2.2 which displays the information that the cart is currently empty, together with a link back to the categories page.

After choosing a subcategory, the user gets a list of all available products as shown in Figure 2.3 where apples have been chosen. Each line has a text input field, where the user may enter the desired amount to be ordered. There are buttons for placing the order, changing the category and

Function/Variable	Description
`__construct()`	The constructor of a class
`__destruct()`	The destructor of a class
`_GET`	Associative array containing the request parameters of a get request.
`_POST`	Associative array containing the request parameters of a post request.
`array`	Creates an array
`date`	Returns a data string with the current date formatted according to the given format string
`die`	Does the same as `exit`
`echo`	Output one or more strings
`empty`	Determine whether a variable is empty
`exit`	Terminates the execution of the current script
`header`	Sends an raw HTTP header
`include`	Includes and evaluates the specified file
`include_once`	Includes and evaluates the specified file if hasn't been included yet
`isset`	Checks if a variable exists
`mysqli_query`	Performs a query against the database
`mysqli_insert_id`	Returns the auto generated id used in the last query
`mysqli_fetch_assoc`	Returns an associative array that corresponds to the fetched row or NULL if there are no more rows
`mysqli_fetch_array`	Fetch a result row as an associative, a numeric array, or both
`mysqli_error`	Returns the text of the error message from previous MySQL operation
`session_destroy`	Destroys all data registered to a session
`session_start`	Starts a new user session
`sizeof`	Count all elements in an array
`substr`	Returns a part of a string

Table 2.1: PHP functions used



Der Futterladen

Wählen Sie eine der folgenden Kategorien

Gemüse

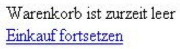
○ Salat
○ Tomate

Obst

○ Apfel
○ Orange

Bei Fragen oder Problemen wenden Sie sich an admin@Futterladen.de

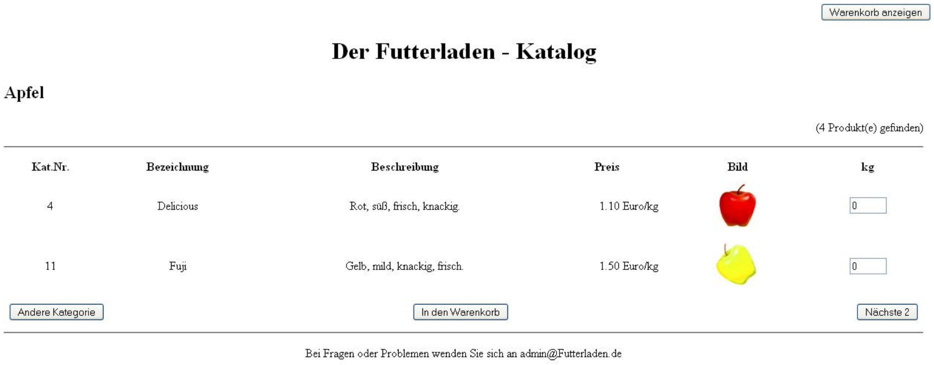
Figure 2.1: The shop catalog page



Warenkorb ist zurzeit leer

Figure 2.2: Empty shopping cart

displaying the cart. As the length of the list is limited to two items, but four items where found, the application offers a paginating functionality. By clicking the button in the lower right corner, the users gets to see the next two results.



Der Futterladen - Katalog

Apfel

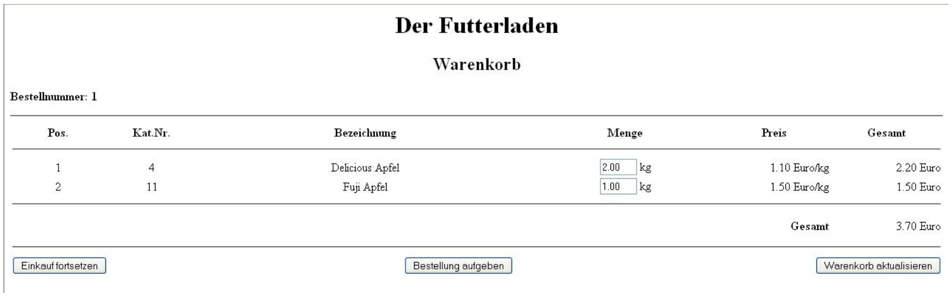
(4 Produkt(e) gefunden)

Kat.Nr.	Bezeichnung	Beschreibung	Preis	Bild	kg
4	Delicious	Rot, süß, frisch, knackig.	1.10 Euro/kg		0
11	Fuji	Gelb, mild, knackig, frisch.	1.50 Euro/kg		0

Bei Fragen oder Problemen wenden Sie sich an admin@Futterladen.de

Figure 2.3: The products page for apples

If the user decides to order 2 kg of Delicious and 1 kg of Fuji, he is redirected to the shopping cart. Figure 2.4 shows a line for each products, together with the price per kilogram, the total price and a text input field with the currently selected amount. Furthermore there is a line for the total price. The user has three choices now: To proceed shopping, to place the order or to update the cart by entering a different amount for a product. The user may change the amount of Fuji apples to 4 kg and finally clicks on the `Place order` button. On the next page, shown in Figure 2.5, he may enter the payment and shipping information.



Der Futterladen

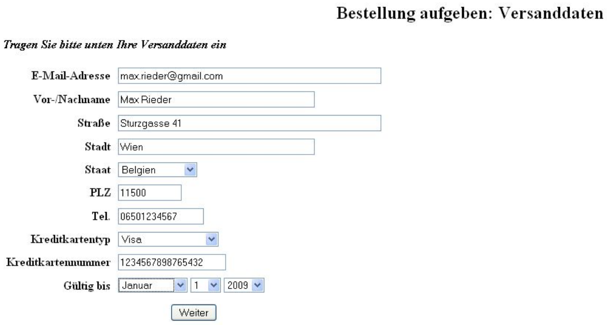
Warenkorb

Bestellnummer: 1

Pos.	Kat.Nr.	Bezeichnung	Menge	Preis	Gesamt
1	4	Delicious Apfel	2.00 kg	1.10 Euro/kg	2.20 Euro
2	11	Fuji Apfel	1.00 kg	1.50 Euro/kg	1.50 Euro
				Gesamt	3.70 Euro

Figure 2.4: The shopping cart with two products

The next page displays a summary of the shipping address, the order positions, and the total amount to be payed. Now the user may choose to continue his shopping tour, to change the shipping and payment information, to cancel or to submit the order. Finally, a click on the submit order button



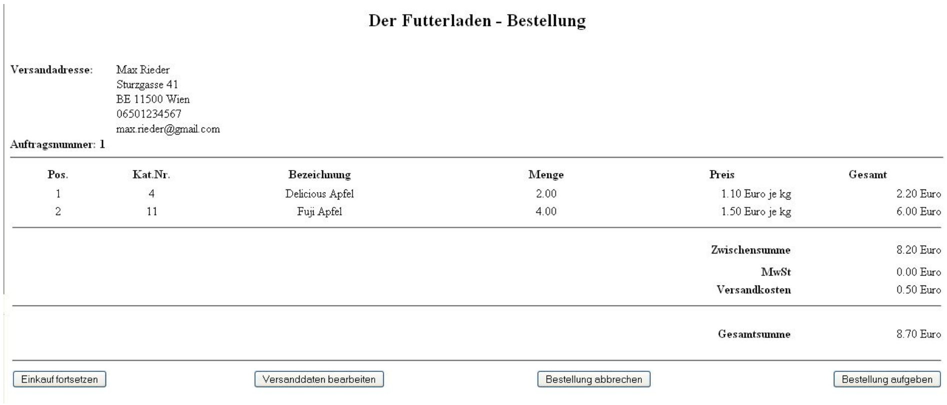
Bestellung aufgeben: Versanddaten

Tragen Sie bitte unten Ihre Versanddaten ein

E-Mail-Adresse	max.rieder@gmail.com
Vor-/Nachname	Max Rieder
Straße	Sturzgasse 41
Stadt	Wien
Staat	Belgien
PLZ	11500
Tel.	06501234567
Kreditkartentyp	Visa
Kreditkartennummer	1234567898765432
Gültig bis	Januar 1 2009

Figure 2.5: A form to enter shipping and payment information

stores the order into the database.



Der Futterladen - Bestellung

Versandadresse:
Max Rieder
Sturzgasse 41
BE 11500 Wien
06501234567
max.rieder@gmail.com

Auftragsnummer: 1

Pos.	Kat.Nr.	Bezeichnung	Menge	Preis	Gesamt
1	4	Delicious Apfel	2.00	1.10 Euro je kg	2.20 Euro
2	11	Fuji Apfel	4.00	1.50 Euro je kg	6.00 Euro
				Zwischensumme	8.20 Euro
				MwSt	0.00 Euro
				Versandkosten	0.50 Euro
				Gesamtsumme	8.70 Euro

Figure 2.6: Summary of the order

2.3 Description of the Source Code

The implementation of the webshop does not follow any good programming practice or design pattern for web applications, such as the Model-View-Controller pattern. There is not even any separation of concerns, e.g into template files containing HTML and simple display logic and files containing only application and business logic.

2.3.1 The Data Model

The shopping cart application is based on three tables in a MySQL Database. The `food` table holds information about the products offered by the web shop. The `customer_order` table stores each order placed by a customer together with the shipping and payment information. The `order_item` table is a link table between the `food` table and the `customer_order` table and stores each item the user puts into the shopping cart. It also stores the requested quantity for each item and the price for each item line. An obvious shortcoming of the database model is that the food table is not normalized as the food category and type is stored in attributes of the food table and not in separate tables.

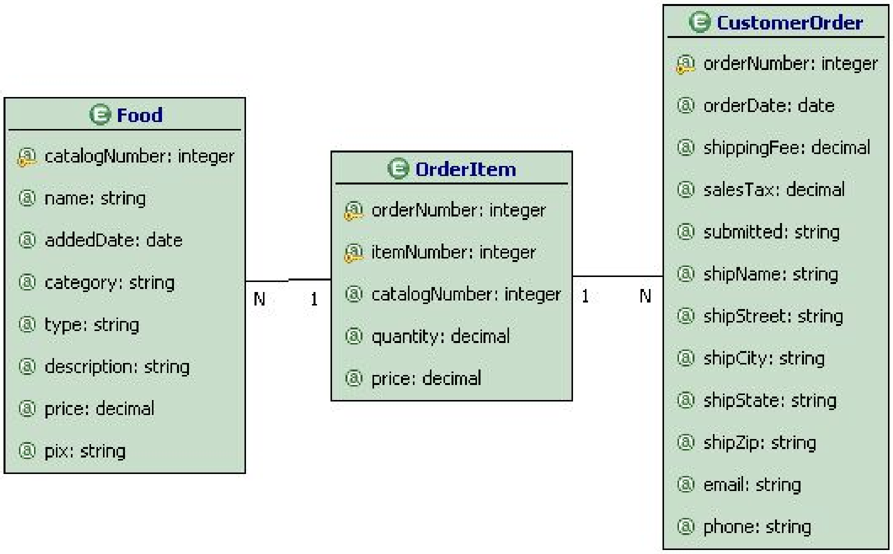


Figure 2.7: The database model of the webshop application

2.3.2 The Shop Catalog Script

The entry page is called ShopCatalog.php with the purpose to display a product catalog. The script either displays the product categories available, or the product page, after a category was chosen. The script first starts a user session with a call to session_start(). Then another script called functions_main.inc, containing the function Connect_to_db() to establish a connection to the databases, is included with a call to include_once(). The rest of the script consists of several nested if/else blocks. First it is checked, whether the post parameters Products and interest are set, as shown in Code Snippet 1.

```
if(isset($_POST['Products'])
  && isset($_POST['interest'])){ ...
```

Code Snippet 1: Check if request parameters are set

On the first request this condition fails and the corresponding else block, shown in Code Snippet 2 is executed:

A connection to the database is established, and the food table containing all available products is queried. The next step is to iterate over the result and put it into a two dimensional array. As shown in this example, PHP allows associative arrays (hashes). The category name is the key of

```
} else {
  $connect = connect_to_db("Vars.inc");
  $sql_cat = "SELECT DISTINCT category,type FROM Food
    ORDER BY category,type";
  $result = mysqli_query($connect,$sql_cat)
    or die("sql_cat: ".mysqli_error($connect));
  while($row = mysqli_fetch_array($result)){
    $food_categories[$row['category']][]=$row['type'];
  }
  include("fields_index_page.inc");
  include("catalog_index_page.inc");
}
```

Code Snippet 2: Else block

the first dimension and with each iteration the type is is added to the end of its corresponding type. Finally there are two include statements. fields_index_page.inc defines some constants to be displayed to the user. catalog_index_page.inc, shown in Code Snippet 3, produces the HTML displayed to the user.

The catalog_index_page.inc script is a mixture of inline HTML, PHP echo statements outputting HTML or variables and some iteration statements. This is very bad programming style, as all view related code should be placed into template files, containing mostly inline HTML and usually only one line long PHP statements for echoing variable values and opening or closing iteration blocks or conditional statements. Outputting HTML via echo statements should be avoided.

```
<html>
<head><title><?php echo $page['title'] ?></title></head>
<body>
<?php
echo "<form action='ShoppingCart.php' method='POST'>\n
...
echo "<form action='$_SERVER[PHP_SELF]' method='POST'>\n";
foreach($food_categories  as $key => $subarray){
   echo "<h3>$key</h3>"; echo "<ul>";
   foreach($subarray as $type){
      echo "<input type='radio' name='interest'
                   value='$type'><b>$type</b><br>\n";
   }
   echo "</ul>";
}
echo "<p><input type='submit' name='Products'
  value='Kategorie w&auml;hlen'>\n  </form>\n";
?> </div> <hr> ...
```

Code Snippet 3: catalog_index_page.inc

After the user submits the lower form together with the selected product, the request is again handled by the ShopCatalog.php as indicated by the $_SERVER[PHP_SELF] variable which contains the name of the currently executed script.

This time as the `Products` and `interest` parameters are set, the if block in Code Snippet 1 is entered. First it it checked if the value of the `Products` parameter is `Add to cart`. If so the user has submitted his request from the products view by clicking the `Add to cart` button. Otherwise he has submitted the request from the categories view by clicking the `Choose category` button.

The else block contains code to implement the pagination functionality and a database query to select all products of the selected subcategory. Then there is a while loop to put the query result into a two-dimensional array. Finally there are again two `include` statements. One to include the `fields_products_page.inc` as above and one to include the `shopping_product_page.inc` script, which is responsible to render the HTML for the product list. The `shopping_product_page.inc` follows a similarly bad programming style as the `catalog_index_page.inc` script and is omitted for brevity.

```
if($_POST['Products'] == "In den Warenkorb"){
...
} else {
  // code to implement a pagination functionality
  // and to select the submitted product
  // from the food table
  include("fields_products_page.inc");
  include("shopping_product_page.inc");
}
```

Code Snippet 4: Request handling to display the product page

The `if` block shown in Code Snippet 4 is executed when the user clicks the `Add to cart` button on the products page. The content of the `if` block is shown in Code Snippet 5. First it is checked whether an order number is already available in the session. If not, a new entry into the `customer_order` table is created. The auto generated id is fetched and and stored in the session as the order number and the number of items is set to 0. If the order number is already available in the session, it is retrieved together with the number of items already listed in the order.

Then in a foreach block it is iterated over all parameters of the post request. The catalog number is extracted from each request value and the database is queried for the corresponding price. Then the order number, the item number, the catalog number, the quantity and the price is inserted into the `order_item` table. Finally the number of items is stored into the session and the user is redirected to the shopping cart page.

2.3.3 The Shopping Cart Script

The handling of the shopping cart is done by the `ShoppingCart.php` script. Again it starts with a call to `session_start()` followed by an include of `functions_main.inc`. Next it is checked whether an order number has been stored into the session as shown Code Snippet 6. If not, a message informing the user that the shopping cart is empty and a link back to the shop catalog page is echoed and the script terminates with a call to `exit()`.

In case the order number exists, which means that there are products in the shopping cart, the

```
if(!isset($_SESSION[’order_number’])){
  // code to create a new order
  // in the customer_order table
} else {
  $order_number = $_SESSION[’order_number’];
  $n_items = $_SESSION[’n_items’];
}
foreach($_POST as $field => $value){
  // code to select the price of the product
  // and to create a new entry in the
  // order_item table
}
$_SESSION[’n_items’] = $n_items;
header("Location: ShoppingCart.php");
exit();
```

Code Snippet 5: Request handling to a product to the cart

```
if(!isset($_SESSION[’order_number’])
   or empty($_SESSION[’order_number’])){
  echo "Warenkorb ist zurzeit leer<br>\n
    <a href=’ShopCatalog.php’>Einkauf fortsetzen</a>\n";
  exit();
}
```

Code Snippet 6: Empty shopping cart

script continues with a switch block with the value of the Cart parameter of the post request as argument. This is shown in Code Snippet 7.

```
switch (@$_POST['Cart']){
  case "Einkauf fortsetzen":
    header("Location: ShopCatalog.php");
    break;
  case "Warenkorb aktualisieren":
    // code to update the cart
    include("fields_cart.inc");
    include("table_page.inc");
    break;
  case "Bestellung aufgeben":
    header("Location: ProcessOrder.php?from=cart");
    exit(); break;
  default:
    include("fields_cart.inc");
    include("table_page.inc");
    break;
}
```

Code Snippet 7: Non-empty shopping cart

There are four cases of interest. The first three cases reflect the buttons the user can click on the shopping cart page. The default case is executed if the user comes from another page or has pressed the refresh button. First the fields_cart.inc script is included, which defines some variables used in the view script. Then there is a query to select all current items in the shopping cart from the order_item table. If the query returns an empty result set, a message is echoed, informing the user that the cart is empty together with a link back to the shop catalog page and the script terminates. Next there is a while loop iterating over all the rows and a nested foreach loop iterating over the fields of each row. For each order item there is another query to the food table, selecting the corresponding name and type of the product. An array is constructed, combining the data of tables.

The table_page.inc has a similar appearance as the catalog_index_page.inc. There are several echo statements for outputting HTML and a for loop to render the content of the array containing the order items into a table. The table displays the item number, the catalog number, the name and the price multiplied by quantity for one item line. Furthermore there is a text field input for the quantity, so the user may change the desired amount of a product. At the end the total price is calculated. There are three submit buttons rendered to the user: Continue shopping, Place order and Update cart.

Another case of special interest is "Update cart". This block is executed if the user changes the amount of certain order items. A foreach loop iterates over the quantity array submitted via a post request. Each order item for the current order number in the order_item is updated with the quantity submitted. Next, all items with the quantity 0 are deleted. The numbering of the items might be wrong now as the deleted items leave holes in the order. To restore the correct order, all items are selected and put into array. Then they are deleted from the table, renumbered and reentered. In case that there are no more items left in the cart (the user has set all quantities to 0), a message is echoed to the user, informing him that the cart is empty together with a link back to the shop catalog page and the script terminates. Otherwise the shopping cart is redisplayed by including `fields_cart.inc` and `table_page.inc`, as it is done in the `default` case.

The remaining two cases simply perform redirects to other scripts. The `"Continue shopping"` case redirects the user back to the shop catalog page and the `"Place order"` case redirects the user to the process order script.

2.3.4 The Process Order Script

The process order script `ProcessOrder.php` basically handles the processing of the shipping and payment for the order. Again it starts with a call to `session_start()` and to `include("functions_main.inc")`. Then it is checked whether an order number has been set, as shown in Code Snippet 8. If not, the user is redirected to the catalog page and the script terminates. The rest of the script consists of several `if`/`elseif` blocks which are executed depending on the submitted request parameters.

```
if(!isset($_SESSION['order_number'])){
  // redirect to ShopCatalog.php
}
if(@$_GET['from'] == "cart"){
   // if the user comes form the shopping cart
   // page, display shipping info entry form
}
elseif(isset($_POST['Summary'])){
   // validate all data submitted
   // and show a summary
}
elseif(isset($_POST['Ship'])){
   // redisplay shipping info entry form
   // if the user wants to change his data
}
elseif(isset($_POST['Final'])){
  // Finish the ordering process
  // store everything to the customer_order table
}
```

Code Snippet 8: Process order

If the user was redirected from the shopping cart page, the `if(@$_GET['from'] == "cart"){...}` block is executed. First the script `fields_ship_info.inc` is included, which contains some arrays with field names and elements for the shipping info form. Then the script `single_form.inc` is included which renders the actual shipping info form. When the user submits the shipping details form, the `name` of the submit button is set to `Summary`, so in this case the block containing the `elseif(isset($_POST['Summary']))` check is entered. This block performs some input validation and displays redisplays the shipping info form together with an error message if the input validation failed. If not, the data submitted by the user is written to the `customer_order` table and the summary page is displayed by including `fields_summary.inc` and `summary_page.inc`. Now the user has the choice to click the button for changing the shipping info data or the button to confirm the order. In the first case the block containing elseif(isset($_POST['Ship'])) check is entered and the shipping info form is redisplayed. In the latter case the elseif(isset($_POST['Final'])) block is entered. If the user has chosen to cancel his order, the order_number is unset from the session, the session is destroyed and the user is redirected back to the catalog page. If the user decides to continue the shopping tour he is also redirected to the catalog page. And finally if the user decides to confirm the order submitted flag in the customer_order table is set to yes and the session is destroyed.

Chapter 3

Overview of the Reverse Engineering Process

In this Chapter an overview of the reverse engineering process, the requirements for applying the process and the methodology for the implementation is given.

3.1 The Reverse Engineering Process

Figure 3.1 gives an overview of the whole reverse engineering process. In the first step the requirements for the reverse engineering process are analyzed and the source application is refactored according to these requirements, which results in a MVC version of the example application. In the second step an automatic code to model transformation is performed. The result of this transformation is an intermediate model of the MVC web application. The third step is the automatic transformation from the intermediate MVC model into the target WebML model. The automatic transformation steps require the definition of a meta model for the intermediate MVC data structure and a meta model for the target WebML data structure.

3.2 Methodology of this Thesis

The the reverse engineering process is developed in two phases. Phase 1 is about the definition of mappings between the source data structure (i.e. a PHP-MVC application) and the target data structure (i.e. WebML). This is done by the means of intermediate data structures. Phase 2 consists of the implementation of the reverse engineering program according to the mappings defined in phase 1. The program is implemented in Java.

3.2.1 Requirements for the Reverse Engineering Process

In Chapter 4 it is analyzed what requirements a web application must fulfill in order to be able to develop a reverse engineering process for it at all. The transformation process is based on automatic pattern recognition in the source artifacts. The most widely adopted pattern for web applications

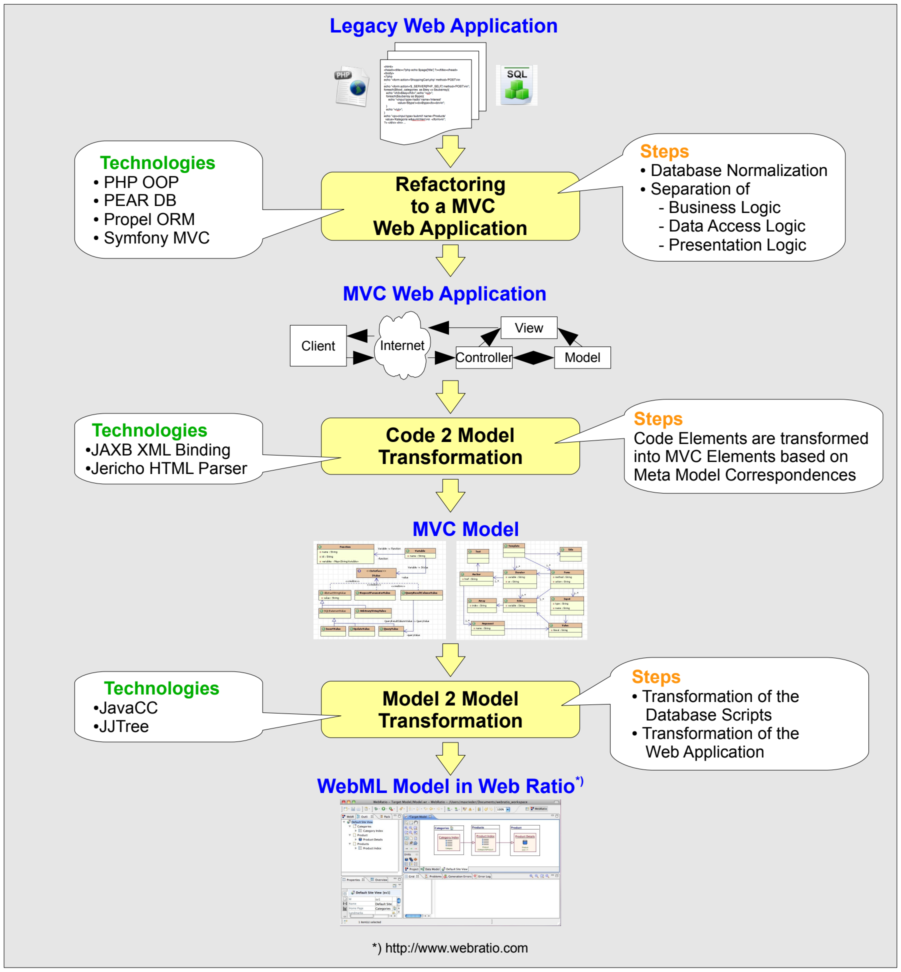


Figure 3.1: Overview of the Reverse Engineering Process

is the Model-View-Controller pattern (MVC). There are countless MVC frameworks available for PHP. To demonstrate the similarity between those kind of frameworks, the example application is manually rewritten for two different MVC platforms. The first one is a simple MVC framework, presented in an article on the O'Reilly website, which is rather intended to introduce the principles of the MVC pattern in PHP than being a full-fledged development platform. The second one is Symfony, one of the most popular and powerful PHP MVC frameworks available at the time of writing. Symfony offers much more features than the simple MVC framework. However the aim of rewriting the example application for both frameworks is to show that the basic principles of how the code is structured are similar. Hence a reverse engineering process developed for one framework should be easily adaptable to fit for the other framework. The process presented in the following chapters is developed for the simple MVC framework.

3.2.2 Conceptual Design

In Chapter 5 the necessary ingredients for phase 1 of the reverse engineering process are described. The most important modeling elements and patterns of the Web Modeling Language are presented. The target data structure (i.e. the WebML model) as well a the intermediate data structure for mapping the view parts of the application are expressed in XML. For the implementation of the mappings between the compiler program and these data structures the JAXB XML binding framework is used, which is briefly introduced. The view parts of the application mainly consist of HTML code that has to be parsed. Therefore the HTML parsing tool Jericho is used in the implementation phase, which is also presented in Chapter 5.

Chapter 6 describes phase 1 of the reverse engineering process. This phase is divided into three steps:

1. The target data structure is defined. This involves the creation of Java classes that represent the required modeling elements of WebML. These classes are mapped to an XML representation that can be viewed and processed by the commercial WebRatio tool for WebML modeling. The mapping is done using the JAXB XML binding framework.

2. A mapping between the patterns used in the view layer of the source application and the elements of the the WebML hypertext model is defined. The source artifacts of the view layer are template files that consist mostly of HTML code with some small parts of PHP code in between which is limited to statements for echoing variable values and for iterating over list values. To make the mapping easier an intermediate XML representation is defined, which only contains the parts that are relevant for the mapping, such as forms, input elements, hyperlinks, iteration and echo statements.

3. A mapping between the patterns used in the model layer of the source application and the content management model of the target application is defined. Again an intermediate data structure is used that helps to map framework and source language specific concepts such as model classes, functions, database queries, request parameters or variables to WebML modeling elements.

3.2.3 Implementation

In Chapter 7 the necessary ingredients for phase 2 of the reverse engineering process are described. In order to implement the reverse engineering program, it is necessary to understand the basic principles of how a compiler works. This is described by the means of a simple calculator example. Then the parser generator tool JavaCC and the preprocessor JJTree for the generation of abstract syntax trees are described.

Chapter 8 describes phase 2 of the reverse engineering process, the implementation of the reverse engineering program. This includes two major steps:

1. The first step is to write a compiler that takes the database creation script of the source application as its input and creates a WebML data model out of it. This includes the writing of a grammar file and the creation of an abstract syntax tree. The mapping between SQL create table statements and a WebML data model is almost straightforward.

2. In step two a compiler is written that creates a WebML hypertext model using the data model from step one, the model and the view artifacts of the source application. The work that the compiler has to do is rather complex and is therefore divided into several sub steps:

   (a) Building and abstract syntax tree of the PHP code.

   (b) Transforming the view templates into the intermediate XML representation for the view layer.

   (c) Transforming the model classes into WebML Operation Modules using the intermediate data structure for the model layer.

   (d) Creating the WebML Pages using the intermediate XML representation.

   (e) Creating the Links between the Pages and the Operation Units.

   (f) Serializing the complete WebML project to its XML representation.

Chapter 4

Requirements for the automatic Reverse Engineering Process

In this chapter the refactoring of the example application into a MVC web application is described. This is the requirement for further automatic processing of the source code.

4.1 A simple MVC Framework

The purpose of the MVC design pattern is to separate the logic of the user interface from the business logic of an application by dividing it into three concerns:

- The model encapsulates the business logic and the operations on the application data.
- The view presents the data from the model to the user.
- The controller receives all incoming requests and acts as a moderator between the model and the view. Furthermore the controller might perform tasks like user authentication or the filtering of requests.

For the first iteration in the reverse engineering process of the example application a slightly modified version of a simple model-view-controller (MVC) framework as described in an article on the O'Reilly website [32] is used. It does not have a name, so it is referred to it as *MVC framework*.

The MVC framework allows the application developer to organize his application code into modules. A module is a directory containing model and view parts of the application. For each web page of the application there is one file containing one model class and one corresponding template file to render the view. A model class file has to have the same name as the class it contains with the ending `.php`. The corresponding template file has to have the name of the model class with the ending `tpl.php`.

Table 4.1 describes the first level of the directory structure of the MVC Framework.

To create a module the developer has to put a directory inside the `modules` directory and call it as the name of the module should be. All model class files belonging to this module are placed immediately inside this directory. In the original version of the MVC Framework there is just one

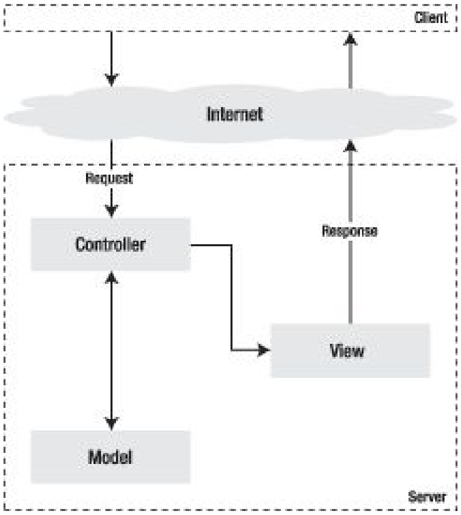


Figure 4.1: The MVC pattern (Source: [25])

Resource name	Type[a]	Description
`includes`	D	Contains all classes provided by the MVC framework
`modules`	D	Contains the application code written by the developer
`resources`	D	The place to keep things like images, stylesheets etc.
`.htaccess`	F	Contains rewrite rules for the Apache Web Server
`config.php`	F	Contains application settings, such as the Database DSN
`index.php`	F	The controller script

[a] D = directory, F = file

Table 4.1: The first level of the MVC framework directory structure

directory inside the module directory called `tpl`, which holds templates written for the smarty template engine [31]. In the modified version used for this thesis there is another directory called `phptpl` which holds templates written in plain PHP. For this thesis only plain PHP templates are used.

4.1.1 The Model

Most of the application logic resides in the model. It is also the place where operations on the database are performed. A model class has to extend from one of two classes, provided by the framework. The first one is `FR_Auth`, which is the parent class for all model classes that should only be accessible by authenticated users. The second one is `FR_Auth_No`, which is the parent class for all model classes that do not require user authentication. Those two classes are part of a class hierarchy that provides access to different functions and objects which are useful for many web applications.

The `FR_Auth` declares an abstract `authenticate` function. An implementation of this function is responsible to check the user credentials an has to return `true`, if the authentication was successful or `false` otherwise. `FR_Auth_No` is a convenience class that extends *FR_Auth* and implements the `authenticate` by simply returning `true`. The `authenticate` function is called by the controller as described in Subsection 4.1.2.

All public member functions of a model class can be called by the controller. Such a call is triggered by the user of the web application via a certain request query string. This is described in Subsection 4.1.2.

Code Snippet 9 outlines the structure of a model class. A constructor with a call to the parent constructor and a destructor with a call to the parent destructor is required by the framework. Furthermore the constructor can be used to initialize the model object with the desired information. In his example the `presenter` property is set to `phptpl`, which causes the framework not to use the PHP based templates in the `phptpl` directory of a module instead of the smarty based templates.

As many web applications operate on relational databases, the MVC framework has built in support for database access and database manipulation by providing a class called `FR_Object_DB` in the class hierarchy of the model classes. `FR_Object_DB` has a protected field called `db` that points to a DB container object of the PEAR DB API [48]. This object can be used in custom model classes to perform database queries. The object is is initialized in the constructor of the `FR_Object_DB` class.

The `set` function is inherited from the `FR_Module` class which is part of the class hierarchy for model classes. This function is used to set data for a module that can be accessed by the view. The second argument is the object passed to the view and the first argument is the variable name under which the object will be available in the view template.

The `__default` function in Code Snippet 9 is a simple example for a typical task to be performed by a model class. The aim is to perform a database query and to pass the result to the view, wrapped into an array. The first statement is a query string to select the `id` and the `name` of all categories in the categories table. The query is performed by a call to the `query` method of the `db` object. In a `while` loop it is iterated over all result rows and the value of the `name` field of each row is put into an array named `category_names`. Finally the `category_names` array is passed to the view, where it will be accessible as `categories`.

```
class categories extends FR_Auth_No {
  public function __construct() {
    parent::__construct();
    $this->presenter = "phptpl";
  }
  public function __default(){
    $sql = "SELECT * FROM Categories";
    $result = $this->db->query($sql);
    $categories = array();
    while ($row =& $result->fetchRow()) {
      $category_names[]=$row['name'];
    }
    $this->set('categories',$category_names);
  }
  public function performAction(){
    // do something else
  }
  public function __destruct() {
      parent::__destruct();
  }
}
```

Code Snippet 9: categories.php, a model class extending from FR_Auth_No

4.1.2 The Controller

The controller in the MVC Framework is the file `index.php`. All requests addressed to the MVC Framework are routed to this script together with a HTTP query string [2] used for determining the responsible model class and member function to handle the request. The query string may contain the keys listed in Table 4.2.

Key	**Value**	**Required?**
`module`	The name of the module	Yes
`class`	The name of the model class	No
`event`	The name of the public function to be invoked	No

Table 4.2: Key-values pairs of the query string for method invocations

For example, given that the class outlined in Code Snippet 9 belongs to a module called `webshop` and the user of the web application wants to trigger the invocation of the public member function `performAction`, he could do so by sending the request `/index.php?module=webshop&class=categories&event=performAction` to the web server. The `__default()` method is called in case that the event argument is omitted in the request. If the `class` argument is missing the controller assumes that there is a class with the same name as the module.

Code Snippet 10 shows the code of the controller script. The `module`, the `event` and the `class` parameters are read from the _GET array. If the `event` is omitted it is set to `__default`. Then it is tried to load the model class file associated with the request and to create an instance of this class. The static `isValid` method of the class `FR_Module`, provided by the framework, checks, whether the instance just created is a valid model class, meaning that it has to be an instance of FR_Module and FR_Auth. The next step is a check whether the user is authenticated. If so, the requested member function of the module is invoked. If the function returns a string with a path to another location, the user is redirected to it. Otherwise the a presenter object is retrieved and its display method is called which is responsible for rendering the view to the user.

```
if (isset($_GET[’module’])) {
  $module = $_GET[’module’];
  if(isset($_GET[’event’])){$event = $_GET[’event’];}
  else {$event = ’__default’;}
  if(isset($_GET[’class’])){$class = $_GET[’class’];}
  else {$class = $module;}
  $classFile =
    FR_BASE_PATH.’/modules/’.$module.’/’.$class.’.php’;
  if (file_exists($classFile)) {
    require_once($classFile);
    if (class_exists($class)) {
      try {
        $instance = new $class();
        if (!FR_Module::isValid($instance)) {
          die("Requested module is not
            a valid framework module!");
        }
        $instance->moduleName = $module;
        if ($instance->authenticate()) {
          try {
            $result = $instance->$event();
            if(isset($result)){
              header("Location: $result");
            }
            if (!PEAR::isError($result)) {
              $presenter =
                FR_Presenter::factory(
                  $instance->presenter,$instance);
              if (!PEAR::isError($presenter)) {
                $presenter->display();
              } else {
                die($presenter->getMessage());
              }
            }
          }
   // several catch statemens come here
}
```

Code Snippet 10: The controller - index.php

A problem in the format of the URL path and the query string presented so far is, that it is not search engine friendly and that post requests would not work as the controller only checks the _GET array. A solution to this problem is the use of the apache web server's mod_rewrite module [45] for rule-based URL rewriting.

The mod_rewrite module allows the developer to define an unlimited number of rewrite rules using the syntax RewriteRule Pattern Substitution [flags]. The apache mod_rewrite documentation [43] says: ”Pattern is a perl compatible regular expression, which is applied to the current URL. “Current” means the value of the URL when this rule is applied. This may not be the originally requested URL, which may already have matched a previous rule, and have been altered." The order in which the rewrite rules are defined is important as they will be applied in this order at runtime. Table 4.3 gives some hints on the syntax of regular expressions.

Expression	**Description**
`.`	Any single character
`[chars]`	Character class: Any character of the class `chars`
`[^chars]`	Character class: Not a character of the class `chars`
`text1|text2`	Alternative: `text1` or `text2`
`?`	0 or 1 occurrences of the preceding text
`*`	0 or N occurrences of the preceding text (N >0)
`+`	1 or N occurrences of the preceding text (N >1)
`^`	Start-of-line anchor
`$`	End-of-line anchor

Table 4.3: The basics of the regular expressions syntax in mod_rewrite

The aim is to avoid the mentioning of the controller file `index.php` together with the query string consisting of key-value pairs and to use a path that looks similar to paths used for directory structures in file systems instead. A model function should be accessible via the pattern
`http://<host-name>/<module-name>/<class-name>/`
`<function-name>`.
To achieve this behavior the url rewriting rules in Code Snippet 11 are used. This code has to be placed in a file called `.htaccess` in the root directory of the web application.

```
RewriteEngine on
RewriteRule ^$ /index.php?module=welcome [L]
RewriteRule ^resources/([.]+)$ /resources/$1 [L]
RewriteRule ^([a-zA-Z0-9]*)$ /index.php?module=$1 [L]
RewriteRule ^([a-zA-Z0-9]*)/([a-zA-Z0-9]*)$
  /index.php?module=$1&class=$2 [L,QSA]
RewriteRule ^([a-zA-Z0-9]*)/([a-zA-Z0-9]*)/([a-zA-Z0-9]*)$
  /index.php?module=$1&class=$2&event=$3 [L]
```

Code Snippet 11: URL rewriting rules used in the MVC framework

The first line is required to activate the rewrite engine. The URL of each incoming request is sequentially matched against each rewrite rule. The `L` flag at the end of each rule stands for "last rule" It is telling the rewrite engine to stop the rewriting process by not applying any more rewrite rules.

- The first rewrite rule is matched if no path is provided. In this case the request is routed to the `welcome` module. The module defined here should be the homepage of the web application.
- The second rewrite rule matches all requests directed to `resources`, followed by a slash and an argument of arbitrary length, but with at least one character. The argument after the

slash should be the file name of a resource in the `resources` directory. As mentioned in 4.1 this directory contains artifacts to be included into a web site such as images or stylesheets. Therefore this path element must be treated differently to all other paths, which denote names of a modules. `$1` in the substitution part is a placeholder for the regular expression in the pattern part. The number `1` after the `$` sign says that the string that matched the first pattern should be inserted here.

- The fourth rule matches an arbitrary sequence of alphanumeric characters that might be written in upper case or in lower case letters, which should be the name of the requested module, in other words the pattern `/<module-name>`.

- The third rule matches the pattern `/<module_name>/<class_name>` and the third rule matches the pattern
`/<module-name>/<class-name>/<function-name>`

4.1.3 The View

The view consists of several template files, one for each model class. The purpose of a template file is to present the data passed from the model to the user. As template files should contain as little program code as possible, the biggest part consists of plain HTML. The only snippets of PHP code that should be used are statements to print the value of variables and to iterate over arrays.

Code Snippet 12 is an example of how a simple template to display the content of the `categories` array of Code Snippet 9 might look like.

```
<html xmlns="http://www.w3.org/1999/xhtml">
  <head><title>Categories</title></head>
  <body>
    <ul><?php foreach($categories as $category){ ?>
        <li><?php echo $category ?></li>
    <?php } ?><ul>
  </body>
</html>
```

Code Snippet 12: The categories template

The `FR_Module` class in the class hierarchy of the model classes contains a member variable called `presenter`. The value of this variable determines the template technology used to present the view to the user. Smarty is the default template engine used by the MVC framework but for the examples described in this thesis it has to be set to `phptpl`. This is done in the constructor of each model class.

4.2 A Transformation into a MVC Framework Application

4.2.1 Transforming the Data Model

To be able to reasonably map the database model to WebML entities and to operate on those entities it is necessary to normalize the database model first. There is a separate table created for the `category` and for the `type` of the food. A `cart` table is created to store the items the user adds to the cart and a `shippinginfo` table is created to store the data the user enters on the shipping information page. Only when the user confirms his order this temporary data is copied to the `order` and to the `order_product` table. The database schema is shown in Figure 4.2.

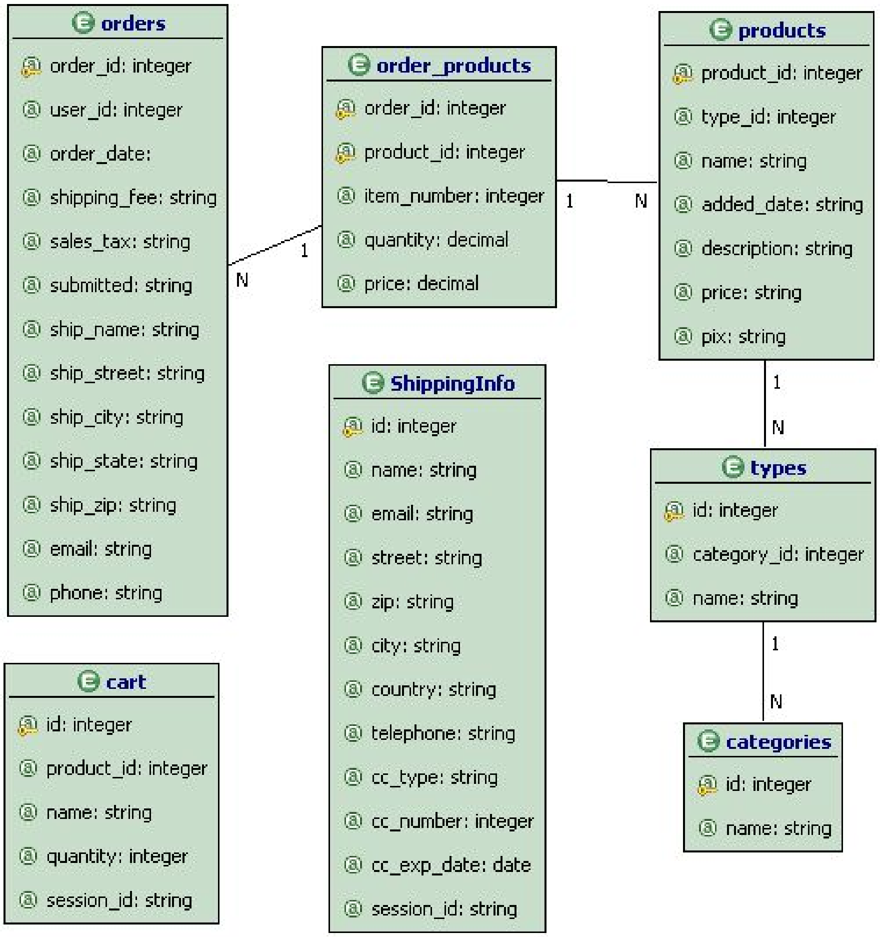


Figure 4.2: The transformed database model

4.2.2 Implementing the Model and the View

In the MVC framework all application code is placed into modules. The code of the example application is placed into a single module called `webshop`. There is one model class and one corresponding template for each page of the example application. All model classes extend from `FR_Auth_No`. The model-template pairs shown in Table 4.4 are created.

Model element	Description
`categories`	A list of all product categories available in the webshop
`types`	A list of all types that belong to a certain category
`products`	A list of all products of a certain type
`productDetails`	Detailed information about a certain product and the possibility to put the desired amount of this product into the shopping cart
`cart`	The shopping cart
`shippingInfo`	A form for entering the shipping information
`summary`	A summary of the order information to be submitted
`processOrder`	The final script for processing an order

Table 4.4: The model and the view elements of the webshop

In the original example application the product types are displayed together with their parent categories on one page and the user directly selects a type. To simplify the reverse engineering process the display of the product categories and the product types is divided into two separate pages. So the user gets to see a list of all categories first together with radio buttons to select one of the categories and after selecting one category he gets to see a list of the corresponding types on another page.

Categories

The `categories` class only implements the default event as shown in Code Snippet 13. The primary key column `id` and the `name` columns of all records of the `Categories` table is queried. In a `while` loop the query result is put into an array which is made available to the template as the variable `food_categories`.

```
public function __default(){
  $sql = "SELECT id, name FROM Categories";
  $result = $this->db->query($sql);
  while ($row =& $result->fetchRow()) {
    $food_categories[]=$row;
  }
  $this->set('food_categories',$food_categories);
}
```

Code Snippet 13: The default event in categories

The corresponding template renders a form with one `input` element of type `radio` for each category in the `food_categories` array as shown in Code Snippet 14. The `value` of each input element is the id of the corresponding category. The name of the category is echoed to the user. The action attribute of the form element points to /webshop/types.

```
<html> ...
<form method="get" action="/webshop/types">
  <ul>
  <?php foreach($food_categories as $category){ ?>
    <li style="list-style: none">
      <input type="radio" name="interest"
        value="<?php echo $category['id'] ?>" />
      <b><?php echo $category['name'] ?></b><br />
    </li>
  <?php } ?>
  </ul>
  <p>
    <input type="submit" name="selectCategory"
      value="Kategorie w&auml;hlen" />
  </p>
</form>
... </html>
```

Code Snippet 14: The categories template

Types

A click on the submit button on the categories page triggers a request to the default event of types, which is shown in Code Snippet 15. The implementation of the method is very similar to Code Snippet 13 but it is a little bit more complex. The form in Code Snippet 14 that triggers the request to /webshop/types passes the id of the chosen category as a request parameter, which is the primary key attribute of the Categories table and a foreign key of the Types table. The sql query string defined is a prepared statement to select the id and the name of the types that belong to the category with the the id passed in the request. The id is retrieved from the _GET array and the sql query is executed with this id. The rest of the code is almost the same as in Code Snippet 13. The selected food types are passed to the template and displayed in the same manner as in Code Snippet 14. The action attribute of the form in the types template points to /webshop/products to select and display all products of a certain type.

Products

The products template renders a table showing id, name, description and price of each product in the products array as well as an image of the product in a table row as shown in Code Snippet 16. For each product there is also an a element rendered that allows to navigate to a page that shows all details of the product. The href attribute points to /webshop/productDetails and the information about what product to select is passed via the query string ?product=<?php echo $product['product_id'] ?>.

```
public function __default(){
  $sql = "SELECT id, name FROM types WHERE category_ID = ?";
  $sth = $this->db->prepare($sql);
  $interest = $_GET[’interest’];
  $result = $this->db->execute($sth, array($interest));
  $food_types = array();
  while ($row =& $result->fetchRow()) {
    $food_types[]=$row;
  }
  $this->set(’food_types’,$food_types);
}
```

Code Snippet 15: The default event in types

```
<html> ...
<table border="0" cellpadding="5" width="100%">
  <!-- table header definitions -->
  <?php foreach ($products as $product){ ?>
  <tr>
    <td><?php echo $product[’product_id’] ?></td>
    <td><?php echo $product[’name’] ?></td>
    <td><?php echo $product[’description’] ?></td>
    <td><?php echo $product[’price’] ?> Euro/kg</td>
    <td><img src="/resources/images/
      <?php echo $product[’pix’] ?>"/></td>
    <td><a href="/webshop/productDetails?
      product=<?php echo $product[’product_id’] ?>">
        Auswaehlen</a></td>
  </tr>
  <?php } ?>
</table>
<form action="/webshop/categories" method="get">
  <p><input type="submit" value="Andere Kategorie" /></p>
</form>
<form action="/webshop/cart" method="get">
  <p><input type="submit" name="Cart"
        value="Warenkorb anzeigen" /></p>
</form>
... </html>
```

Code Snippet 16: The products template

ProductDetails

The default event of the productDetails class selects a single product from the database and passes it to the template as shown in Code Snippet 17.

```
public function __default(){
  $sth = $this->db->prepare(’SELECT * FROM Products
           WHERE product_id=? ORDER BY name’);
  $data = array ($_GET [’product’] );
  $result = $this->db->execute ( $sth, $data );
  $this->set ( ’product’, $result->fetchRow () );
}
```

Code Snippet 17: The default event in productDetails

The productDetails template is outlined in Code Snippet 18. The product_id, name, description, price and an image of the chosen product are displayed inside a table which is embedded inside a form. The action attribute of the form points to /webshop/productDetails/addToCart. The user can add the current product to the shopping cart by clicking the submit button. The desired amount has to be entered into the input field of type text. Furthermore the product_id and the name of the product are passed as parameters on form submission.

```
<html> ...
<form method="post" action="/webshop/productDetails/addToCart">
  <table border="0" cellpadding="5" width="100%">
    <tr>
      <td>Produkt Nummer:</td>
      <td><?php echo $product[’product_id’] ?></td>
    </tr>
    <!-- The same for name, description and price. -->
    <tr>
      <td>Bild:</td>
      <td><img src="/resources/images/
        <?php echo $product[’pix’] ?>"/></td>
    </tr>
    <tr>
      <td>Menge:</td>
      <td><input type="text" name="amount"
        value="0" size="4"></td>
    </tr>
  </table>
  <input type="hidden" name="product_id"
    value="<?php echo $product[’product_id’] ?>">
  <input type="hidden" name="name"
    value="<?php echo $product[’name’] ?>">
  <p><input type="submit" value="In den Warenkorb" /></p>
</form>
<!-- Forms linking to the category and to the cart page -->
... </html>
```

Code Snippet 18: The productDetails template

The addToCart event first queries the cart table to check whether this product has already been added to the cart. If so, the record representing this product belonging to this session already exists in the the table, so an SQL update statement is executed to adapt the amount the user has entered into the form. Otherwise the product has not been added to the cart yet and a new record representing this product is created. Finally the event redirects to /webshop/cart to display the content of the shopping cart to the user.

```
public function addToCart() {
  $session_id = session_id();
  $product_id = $_POST [’product_id’];
  $amount = $_POST [’amount’];
  $name = $_POST [’name’];
  $sth = $this->db->prepare(
    "SELECT * FROM cart WHERE
      product_id = ? AND session_id = ?");
  $result = $this->db->execute($sth,
    array ($product_id, $session_id));
  $row = $result->fetchRow ();
  if ($row) {
    $sth = $this->db->prepare(
      "UPDATE cart SET quantity = ?
        WHERE product_id = ? AND session_id = ?");
    $data = array ($amount, $product_id, $session_id);
    $result = $this->db->execute ( $sth, $data );
  } else {
    $sth = $this->db->prepare(
      "INSERT INTO cart (product_id, name, quantity, session_id)
        VALUES (?, ?, ?, ?)" );
    $data = array ($product_id, $name, $amount, $session_id);
    $result = $this->db->execute ( $sth, $data );
  }
  return "/webshop/cart";
}
```

Code Snippet 19: The addToCart event of the productDetails class

Cart

The default event of the cart class selects all records of the cart table that belong to the current session and passes them to the template. The cart template renders a table that shows the product_id the name and the quantity of each cart item as well as a link to /webshop/productDetails to change the amount of a product to be ordered. The template also renders a form with its action attribute pointing to /webshop/shippingInfo.

The default event of the shippingInfo class is an empty method because there is no data passed to the template. The template renders a form containing several input elements of type text that allow the user to enter data necessary for the shipping process such as name, address or credit card information, as shown in Code Snippet 20. A click on the submit button triggers a request to /webshop/shippingInfo/add.

```
<form method="post" action="/webshop/shippingInfo/add">
  <table border="0" cellpadding="5" cellspacing="0">
    <tr>
      <td>E-Mail-Adresse</td>
      <td><input type="text" name="email" value="" /> </td>
    </tr>
    <!-- More input fields for name, street, city, zip code,
    telephone, country, credit cart type, number
    and expiry date are displayed here -->
    <tr>
      <td colspan="2">
        <p"><input type="submit" name="Summary"
          value="Weiter" /></p>
      </td>
    </tr>
  </table>
</form>
```

Code Snippet 20: The shippingInfo template

ShippingInfo

The `add` method of the `shippingInfo` class functions similar to the `addToCart` method of the `productDetails` class. The shipping information submitted by the user is retrieved from the `_POST` array and stored into variables. Then the `ShippingInfo` table is queried to check whether a record that corresponds to the current session already exists or not. If a record exists the fields are updated with the newly submitted values, otherwise a new record is created. Finally the user is redirected to `/webshop/summary`.

Summary

The summary page presents the content of the shopping cart and the shipping information to the user. In the default event of the `summary` class shown in Code Snippet 21 the `shippingInfo` table and the `cart` table are queried and the results are made available in two arrays, which are passed to the template. The shipping information is presented in a `table` which is nested inside a `form`. The form is pointing to `/webshop/shippingInfo` where the user can change the shipping information. The content of the shopping cart is presented in another table which is not nested inside a form. There are two more forms in the template. One is pointing to `/webshop/cart` which allows the user to modify the content of the shopping cart. The other one is pointing to `/webshop/summary/processOrder`.

The `processOrder` method first queries the `ShippingInfo` table and then inserts this data into the `orders` table. The next step is to query the cart table. In a while loop it is iterated over the resulting array of cart items and each item is inserted into the `order_products` table. Finally the session is destroyed and the user is redirected to `/webshop/processOrder`, which displays a message to the user that the order has been stored successfully.

```
public function __default(){
  $session_id = session_id ();
  $sth = $this->db->prepare(
    "SELECT * FROM ShippingInfo WHERE session_id = ?");
  $data = array($session_id);
  $result = $this->db->execute($sth,$data);
  $row = $result->fetchRow();
  $shippingInfo = $row;
  $sth = $this->db->prepare(
    "SELECT * FROM cart WHERE session_id = ?");
  $data = array($session_id);
  $result = $this->db->execute($sth, $data);
  while($row = $result->fetchRow()){
    $order[] = $row;
  }
  $this->set(’order’,$order);
  $this->set(’shippingInfo’,$shippingInfo);
}
```

Code Snippet 21: The default event in summary

4.3 Symfony

Symfony [29] [25] is one of the most popular open-source PHP 5 MVC frameworks available at the moment. It is in use for various real-world projects and high-demand e-business sites. According to Potencier et al. [25] symfony fulfills the following requirements:

- Easy to install and configure on most platforms (and guaranteed to work on standard *nix and Windows platforms)
- Database engine-independent
- Simple to use, in most cases, but still flexible enough to adapt to complex cases
- Based on the premise of convention over configuration–the developer needs to configure only the unconventional
- Compliant with most web best practices and design patterns
- Enterprise-ready–adaptable to existing information technology (IT) policies and architectures, and stable enough for long-term projects
- Very readable code, with phpDocumentor comments, for easy maintenance
- Easy to extend, allowing for integration with other vendor libraries

4.3.1 Fundamental Concepts

Symfony utilizes the same technologies and concepts as the MVC framework presented in Section 4.1. Those are PEAR, magic methods and object oriented programming (OOP). Furthermore it introduces some new concepts which are described briefly in this Section.

```
public function processOrder() {
  $session_id = session_id ();
  $sth = $this->db->prepare(
    "SELECT * FROM ShippingInfo WHERE session_id = ?");
  $result = $this->db->execute ( $sth, array ($session_id ));
  $row = $result->fetchRow ();
  $name = $row[’name’];
  // the same for $street, $city, $country, $zip, $email
  // and $telephone follows here
  // $now is set to the current date, $user_id is set to 0
  $order_id = $this->db->nextId ( "orders" );
  $sth = $this->db->prepare("INSERT INTO orders
    (order_id, user_id, order_date, ship_name, ship_street,
    ship_city, ship_state, ship_zip, email, phone)
    VALUES (?, ?, ?, ?, ?, ?, ?, ?, ?, ?)");
  $result = $this->db->execute($sth,
    array($order_id, $user_id, $now, $name, $street, $city,
      $country, $zip, $email, $telephone));
  $sth = $this->db->prepare(
    "SELECT * FROM cart WHERE session_id = ?" );
  $data = array ($session_id );
  $result = $this->db->execute ( $sth, $data );
  while($row = $result->fetchRow()) {
    $sth = $this->db->prepare(
      "INSERT INTO order_products(order_id, product_id, quantity)
        VALUES (?,?,?)");
    $product_id = $row[’product_id’];
    $quantity = $row[’quantity’];
    $result = $this->db->execute($sth,
      array($order_id, $product_id, $quantity));
  }
  $session->destroy ();
  return "/webshop/processOrder";
}
```

Code Snippet 22: The processOrder event in summary

Object-Relational Mapping

Object-Relational Mapping (ORM) is a concept to access the data stored in relational databases in an object oriented way, by introducing an object/relational abstraction layer. This prevents the need to write database dependent SQL queries as calls to model objects are automatically translated into queries optimized for the current database. Each table is mapped to a model class and table records are represented by instances of the model classes. Each field of the database table is represented by a member variable of the corresponding model class or by accessor and mutator methods respectively. This enables the developer to add new accessor. For instance if there is a table called `Customer` with two fields called `FirstName` and `LastName` and the developer just requires a `Name`, it is possible to add a new accessor as shown in Code Snippet 23.

```
public function getName(){
  return $this->getFirstName().’ ’.$this->getLastName();
}
```

Code Snippet 23: Adding an accessor to a model class (Source: [25])

Relationships between different table records are also reflected in the object structure by accessor and mutators. For instance in a webshop application there might be a table called `ShoppingCart` and another table called `Item` that have a one to many relationship amongst each other. To reflect this relationship in ORM there would be a model class called `ShoppingCart` with a `getItems` method that returns all references to the related `Item` objects. All the data-related business logic is also placed in such model classes. For instance the `ShoppingCart` might have a `getTotal` to calculate the price of all items in the shopping cart as shown in Code Snippet 24.

```
public function getTotal(){
  $total = 0;
  foreach ($this->getItems() as $item){
    $total += $item->getPrice() * $item->getQuantity();
  }
  return $total;
}
```

Code Snippet 24: Adding an accessor to a model class (Source: [25])

The default ORM framework bundled with symfony is Propel [50] but it is also possible to use Doctrine [57].

Rapid Application Development (RAD)

Symfony applies the programming strategy of rapid application development (RAD) [21]. One of the ideas of RAD is to start developing as soon as possible, without producing tons of documents for requirement analysis first of all. This idea is supported through symfony’s ability to generate much of the application code automatically, based on simple text files.

YAML

YAML is an acronym for ”YAML Ain’t Markup Language” According to the YAML website [59], ”YAML is a human friendly data serialization standard for all programming languages.” In other words, YAML is a language to describe data structures in a way similar to XML but with a simpler syntax. For instance it can be used to describe data that can be translated into array as shown in Code Snippet **??**.

```
house:
  family:
    name: Doe
    parents:
      - John
      - Jane
    children:
      - Paul
      - Mark
      - Simone
  address:
    number: 34
    street: Main Street
    city: Nowheretown
    zipcode: "12345"
```

```
$house = array(
  ’family’ => array(
    ’name’      => ’Doe’,
    ’parents’   => array(
      ’John’, ’Jane’),
    ’children’ => array(
      ’Paul’, ’Mark’, ’Simone’)
  ),
  ’address’ => array(
    ’number’    => 34,
    ’street’    => ’Main Street’,
    ’city’      => ’Nowheretown’,
    ’zipcode’   => ’12345’
  )
);
```

Code Snippet 25: A data structure in YAML (Source: [25])

The hierarchy of data in YAML is described by indentation, sequence items are indicated by a dash and key/value pairs are separated by a colon. YAML also has a shorthand syntax where arrays are described with `[]` and hashes with {}. Thus, the data described in Table **??** can also be written as shown in Code Snippet 26.

```
house:
  family: {
    name: Doe, parents: [John, Jane],
    children: [Paul, Mark, Simone]
  }
  address: {
    number: 34, street: Main Street,
    city: Nowheretown, zipcode: "12345"
  }
```

Code Snippet 26: The shorthand syntax for Code Snippet **??** (Source: [25])

4.3.2 Symfony’s MVC Implementation

Symfony’s MVC implementation is based on the same principles as described in Subsection **??** but the pattern is further subdivided.

The *model* is separated into a *data access layer* and a *database abstraction layer* so the developer does not have to write database-dependent query statements. The database abstraction layer which performs the queries transparently is used for this purpose instead.

The *view* is split into a *layout* and a *template*. The layout usually contains parts of the view that re-occur on several pages, such as the header, the footer or the global navigation bar. It is typically applied to the whole application or to a group of pages. The template renders the variables made available by the controller. The logic used to combine the functionality of the layout and the template is referred to as the *view*.

The *controller* is devided into a *front controller* which performs tasks unique to the whole application and into *actions* which contain only code specific to one page.

Figure 4.3 illustrates how the MVC pattern is realized in symfony.

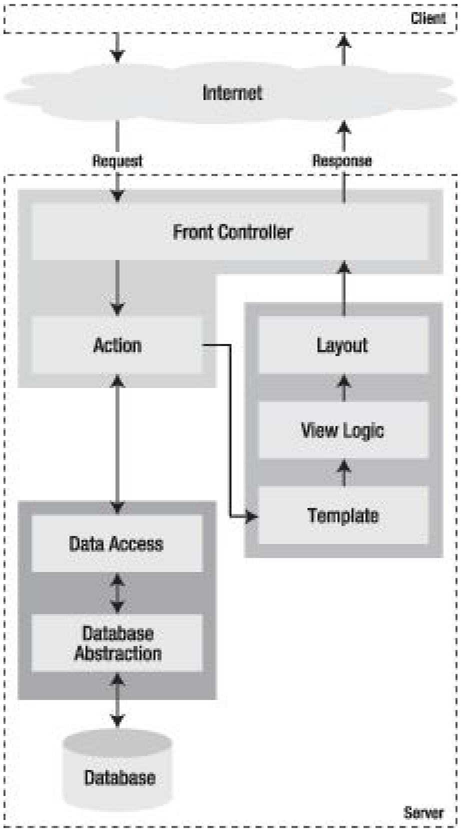


Figure 4.3: The MVC pattern in symfony (Source: [25])

4.3.3 The Project and Code Organization

All code in symfony is organized into a predefined structure as follows:

- A *project* is the directory that contains all artifacts of a symfony website. According to Potencier et al. [25] ”a project is a set of services and operations available under a given domain name, sharing the same object model.”.

- A project contains one or more *applications*. Very often there is a backend application for administrative tasks and a frontend application for the users of the website.
- Each application is subdivided into *modules* A module is responsible for one page or a group of related pages.
- Modules provide *actions* that perform different tasks. For instance a ShoppingCart module might contain an `add` action which adds a new item to the cart.

An example of code organization is given in Figure 4.4. The sub directories contained in the root directory of a project are described in Table 4.5.

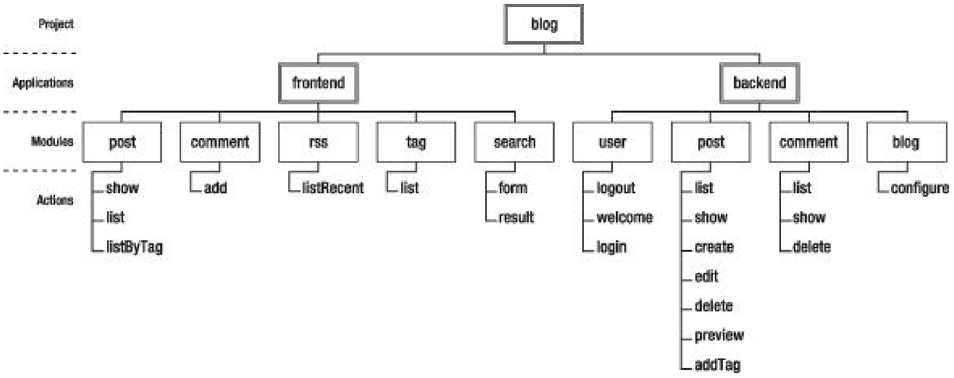


Figure 4.4: Example of code organization (Source: [25])

Directory	**Description**
`apps`	Contains one directory for each application.
`cache`	Holds chached versions of the project configuration, the actions and the templates.
`config`	Stores configuration files for the whole project.
`data`	Contains data files such as a database schema or SQL scripts.
`doc`	Holds the project documentation.
`lib`	Stores foreign classes or libraries and the code that is shared amongst all applications. The model classes also belong to this directory.
`log`	Contains log files generated by symfony.
`plugins`	Stores plugins (this is not discussed in this thesis).
`test`	Contains unit tests (this is not discussed in this thesis).
`web`	Only files in this directory are accessible from the web.

Table 4.5: Sub directories inside the project root directory

4.3.4 The Controller Layer

The controller layer is connecting the business logic and the presentation. According to Potencier et al. [25] it is subdivided into different components:

- The front controller is the unique entry point to the application. It loads the configuration and determines the action to execute.
- Actions contain the applicative logic. They check the integrity of the request and prepare the data needed by the presentation layer.
- The request, response, and session objects give access to the request parameters, the response headers, and the persistent user data. They are used very often in the controller layer.
- Filters are portions of code executed for every request, before or after the action. For example, the security and validation filters are commonly used in web applications. You can extend the framework by creating your own filters.

The Front Controller

The front controller uses a routing system to match an URL submitted by the user with a module and an action. For example the URL `http://localhost/index.php/mymodule/myAction` is addressed to the front controller `index.php` and will be translated into a call to the action `myAction` of `myModule`

Actions

Actions contain the logic of the application. They retrieve request parameters, work with the model and hand over variables to the view. Each module has a corresponding action class called `<my_module_name>Actions` that has to extend from symfony's `sfActions` class. Actions are public member functions inside this class called `execute<ActionName>`. A web request in symfony is always addressed to an action of a module.

The return value of an action method determines which template is used for rendering the view. The return value `sfView::SUCCESS` causes symfony to call the default view by looking for a template called `<action_name>Success.php` This behavior is also triggered if the return value is omitted. In case of an error, the action might return `sfView::ERROR` which causes symfony to look for a view called `<action_name>Error.php`. A custom view can be called by returning `'<my_view>'` which causes symfony to look for a template called `actionName<my_view>.php`.

In some cases an action requests another action after its execution. The action class provides two ways to execute another action:

- A call can be forwarded to another action by calling
  `$this->forward('otherModule', 'index');`

- A call can be redirected by calling
  `$this->redirect(’otherModule/index’);` or
  `$this->redirect(’http://www.tuwien.ac.at/’);`

In case of a forward the URL displayed in the user’s browser stays the same whereas a redirect triggers the browser to submit a new request resulting in a change of the displayed URL. A redirect instead of a forward should always be done if the action is called from a form submitted with the post method. The advantage is that a refresh of the resulting page or a click on the back button by the user does not cause a resubmit of the post request.

The way to access controller-related information and the core symfony objects is demonstrated in Code Snippet 27.

```
class mymoduleActions extends sfActions {
  public function executeIndex($request){
    // Retrieving request parameters
    $password    = $request->getParameter(’password’);
    // Retrieving controller information
    $moduleName  = $this->getModuleName();
    $actionName  = $this->getActionName();
    // Retrieving framework core objects
    $userSession = $this->getUser();
    $response    = $this->getResponse();
    $controller  = $this->getController();
    $context     = $this->getContext();
    // Passing information to the template
    $this->setVar(’foo’, ’bar’);
    $this->foo = ’bar’;  // Shorter version
  }
}
```

Code Snippet 27: Accessing application objects and context information in an action (Source: [25])

4.3.5 The View Layer

The view renders the output of a certain action. It is separated into different parts:

- The actual presentation of a web site is the job of the *templates*, which render the data of current action and the *layout* which is usually globally used for all pages.
- Recurring parts of templates can be put into partials or components that can be reused in different templates (this is not discussed in this thesis).
- The view can be configured by the means of YAML configuration files.

Templating

Code Snippet 28 shows a simple template, containing mostly HTML code and some basic PHP statements. The `name` variable echoed in line two has to be set in the corresponding action via

`$this->name = 'foo';`. The `link_to` function is a so called helper. Helpers are functions that return HTML code. They can be used in templates. The call to `link_to('Read the last articles', 'article/read')` renders an HTML anchor. The second argument indicates that the link is directed to the `read` action of the `article` module. The first argument is the text inside the anchor tag. The `url_for` function works similarly but it only takes a string as an argument, which contains the target to which the URL should be directed.

```
<h1>Welcome</h1>
<p>Welcome back, <?php echo $name ?>!</p>
<ul>What would you like to do?
  <li><?php
    echo link_to('Read the last articles', 'article/read')
  ?></li>
  <li><?php
    echo link_to('Start writing a new one', 'article/write')
  ?></li>
</ul>
```

Code Snippet 28: An example template (Source: [25])

The template in Code Snippet 28 is not a valid XHTML document. Therefore it has to be decorated with a layout. The default layout is shown in Code Snippet 29.

```
<html>
  <head>
    <?php include_http_metas() ?>
    <?php include_metas() ?>
    <?php include_title() ?>
    <link rel="shortcut icon" href="/favicon.ico" />
  </head>
  <body>
    <?php echo $sf_content ?>
  </body>
</html>
```

Code Snippet 29: The default layout (Source: [25])

The content of a template is inserted via the `echo $sf_content` statement in the source code of the layout.

4.3.6 The Data Access Layer

As discussed in Subsection 4.3.1, in a symfony project, all data stored in the database is accessed and modified via objects.

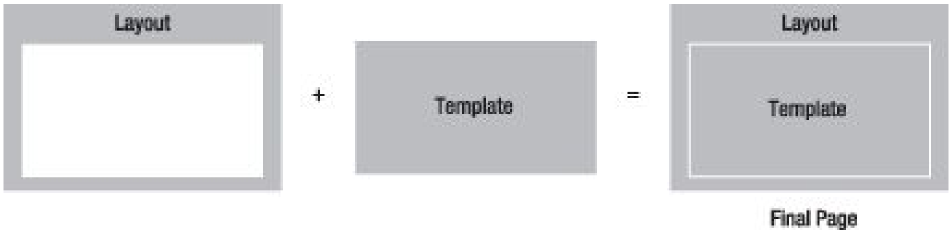


Figure 4.5: A template decorated with a layout (Source: [25])

The Database Schema

The database schema tells symfony how to create a mapping between the relational data model of the database and the PHP object data model to be used. The tables, their relationships and columns are described in the schema, using the YAML syntax. The file `schema.yml` that contains the schema is located in the `myproject/config/` directory.

For instance in a webshop application there might be two tables: `Products` - storing all products of the webshop and `Types` - dividing the products into types. A schema description for these tables might look like the one shown in Code Snippet 30.

```
propel:
  type:
    id:            ~
    name:          { type: varchar(255), required: true,
      index: unique }
  product:
    id:            ~
    type_id:       { type: integer, foreignTable: type,
      foreignReference: id, required: true }
    name:          { type: varchar(255), required: true,
      index: unique }
    description:   { type: longvarchar, required: true }
    price:         { type: float, required: true }
```

Code Snippet 30: A sample database schema

The first key in the file is the connection name, `propel` in this example, referencing a connection to a database, defined in a different file in the same directory, called `databases.yml`. The `type` and the `product` keys are denoting the two tables. In YAML, the keys end with a colon, and the structure is described through indentation. The keys under the table keys denote their attributes. The ~ character tells symfony to guess the definition of the attribute. In the case of the id attribute it is interpreted as the primary key with an auto-incremented integer value. The `name` attribute is typed as `varchar(255)` and defined as a required attribute with a unique index. The product table has a `type_id` attributed defined as a foreign key for the `type` table. This is done by the `foreignTable: type` and `foreignReference: id` statements.

Model Classes

The model classes are generated automatically using the schema definition. This is done by executing the command line task `php symfony propel:build-model`. The following classes are generated into the `lib/model/om/` directory when using the schema shown in Code Snippet 30: `BaseProduct.php`, `BaseProductPeer.php`, `BaseType.php` and `BaseType-Peer.php`. There are four more classes generated into the `lib/model/` directory: `Prod-uct.php`, `ProductPeer.php`, `Type.php` and `TypePeer.php`.

The classes generated into the `lib/model/om/` contain code the should not be modified by the developer whereas the classes in `lib/model/` are provided to the developer for adding custom business logic. For instance the `BaseProduct` and the `BaseType` class already contain accessors, mutators and instance variables for the table fields, as well as some other methods. `Product` extends `BaseProduct` and `ProductPeer` extends `BaseProductPeer`. Objects of the `Product` class for instance represent database records whereas `ProductPeer` provides static methods to perform data access and manipulation operations.

To create a new database record a new instance of the corresponding model class has to be created and the properties of the object can be set via the accessor methods. A call to the save method commits the instance values to the database.

Each model class has a corresponding peer class that is used for object retrieval. A peer class has a `retrieveByPk` method that takes a primary key value as argument and returns the object to the corresponding database record. If several objects should be retrieved the `doSelect` method has to be used. This method takes a `Criteria` object as an argument. With an empty `Criteria` object all instances of the class are retrieved. To restrict a query by value comparison the `add` method is used. The method takes two or three arguments respectively. The first argument is a column, the second argument is a value and the third argument is a comparison operator. If the third argument is omitted, the equal operator is used.

A database record can be deleted via a call to the `delete` method of the corresponding model instance.

Code Snippet Code Snippet 31 gives an example of how to work with the model.

```
// Create and save a new Product instance
$product = new Product();
$product->setName('Pizza Margarita');
$product->save();
// Retrieve a Product object by primary key
$product = ProductPeer::retrieveByPk(7);
// Delete a product
$product->delete();
// Retrieve a Products by name
$c = new Criteria();
$c->add(ProductPeer::NAME, 'Pizza Margarita', Criteria::LIKE);
$products = ProductPeer::doSelect($c);
```

Code Snippet 31: Working with the model

Populating Data to a Database

When developing a web application it is necessary to fill the database with some test records. In symfony this can be done by providing a text file that contains data structured with a simple YAML syntax. The file has to be stored in the `data/fixtures` directory.

The data is organized class-wise. Each class section starts with the class name and contains several records, each one labeled with a unique string. A record consists of fieldname-value pairs. Foreign key references can be expressed by writing the label of the referenced record as the value of the foreign key attribute. The example in Code Snippet 32 defines two class sections, `Category` and `Type`. Two records are defined for the `Category` table, `category_one` and `category_two`, each one defining some data for the `name` and the `description` field. In the `Type` class section two records are defined, each of them having their `category_id` field pointing to the `Fruit` category, labeled with `category_one`.

```
Category:
  category_one:
    name: Fruit
    description: Lorem ipsum
  category_one:
    name: Vegetables
    description: Lorem ipsum
Type:
  type_apple:
    category_id: category_one
    name: Apple
  type_citrus_fruit:
    category_id: category_one
    name: Citrus Fruit
```

Code Snippet 32: A database fixture in YAML syntax

4.3.7 Setting up an Example Project in Symfony

Symfony supports the RAD programming strategy described in Subsection 4.3.1 by providing a command line interface (CLI) to perform common tasks required for the building and the maintenance of a web application. A good overview of how to use the CLI is given in the symfony cookbook [30]. The CLI is implemented as the `symfony` PHP script that lies at the root of a a project. The script requires a task name as a commandline argument and possibly some additional parameters. The syntax is `php symfony <TASK> [parameters]`.

Structure Generation

- A new project is initialized by executing `php symfony init-project` <PROJECT_NAME>.
- A new application is initialized by executing `php symfony init-app` <APPLICATION_NAME>.

- `php symfony init-module <APPLICATION_NAME> <MODULE_NAME>` initializes a new module.

Model Generation

The connection settings for database related tasks are specified in `config/propel.ini`.

- The Propel model classes are generated based on the YAML schema file in the `config` directory of the current project by executing `php symfony propel-build-model`
- The SQL code to create the tables described in `schema.yml` is generated by executing `php symfony propel-build-sql`. The SQL code is written to `data/schema.sql`.
- An empty database is created by executing `php symfony propel-build-db`.
- The sql code form `data/schema.sql` is inserted into the database by executing `php symfony propel-insert-sql`.
- The tasks `propel-build-model`, `propel-build-sql` and then `propel-insert-sql` are all executed when `php symfony propel-build-all` is executed.

Data Management

- `php symfony propel-load-data <APPLICATION_NAME> [<ENVIRONMENT_NAME>] [<FIXTURES_DIR_OR_FILE>]` loads all fixtures contained in `data/fixtures` if not specified differently.
- `php symfony propel-build-all-load <APPLICATION_NAME> [<ENVIRONMENT_NAME>] [<FIXTURES_DIR_OR_FILE>]` first executes `propel-build-all` and then `propel-load-data`.

4.4 A Transformation into a Symfony Application

The first step in transforming the web application into a symfony project is to define the database schema in the YAML format, which is placed into the the `schema.yml` file. The schema used is the same as described in Section 4.2.1. The YAML format to define a database schema is described in Subsection 4.3.6. The propel model classes are generated using the symfony CLI tool. All the application code is put into a single application called `frontend` which contains one module for each model class.

The entry point to the application is the `Categories` module. The `categoriesAction` class only contains a single `index` action that performs a query of all categories available. The result of the query is made available to the template as `category_list`. The content of the `executeIndex` method is shown in Code Snippet 33.

The template displays the content of `category_list` in a `table`. The category's `name` and a link to the `index` action of the module `types` are echoed. The value of the `category_id` field is attached to the query string. A part of the template is shown in Code Snippet 34.

```
$this->category_list = CategoryPeer::doSelect(new Criteria());
```

Code Snippet 33: The Category index action

```
<?php foreach ($category_list as $category): ?>
<tr>
  <td><?php echo $category->getName() ?></td>
  <td><a href="<?php echo url_for(’types/index?category_id=
    ’.$category->getId()) ?>">Kategorie auswaehlen</a></td>
</tr>
<?php endforeach; ?>
```

Code Snippet 34: The Category index action

As the `categories` module the `types` module only contains an `index` action. The `category` table is queried for all types of the category whose `category_id` has been passed as a request parameter. The content of the `executeIndex` method is shown in Code Snippet 35.

```
$criteria = new Criteria();
if($request->hasParameter(’category_id’)){
  $criteria->add(TypePeer::CATEGORY_ID,
    $request->getParameter(’category_id’), Criteria::EQUAL);
}
$this->types_list = TypePeer::doSelect($criteria);
```

Code Snippet 35: The Types index action

The template looks very similar to the category index template outlined in Code Snippet 34. The user may follow a link to see all products of a certain type.

The `products` module contains an `index` action to query all products of a certain type and a `show` action to show the details of a certain product. The content of `executeShow` is shown in Code Snippet 36.

The template displays all the information about the chosen product and it renders a form that points to the `addToCart` action of the `cart` module. The form contains an `input` element of type `text` to enter the desired amount of the product and two hidden fields for passing the `product_id` and the `name`. The template is outlined in Code Snippet 37.

The `addToCart` action queries the `cart` table for the `product_id` submitted and the current `session_id`. If the query returns a result, the `quantity` is updated, otherwise a new object instance is created and saved. Finally a redirect to the `index` action is performed. The content of the `executeAddToCart` is shown in Code Snippet 38.

The `index` action queries all cart items that belong to the current session and passes them to the template as the `cart_list` variable. The template displays the content of the cart in a table and renders a link to the `show` action of the `product` module, which allows the user to change the amount of the product. Furthermore a link to the `displayForm` action of the `shippingInfo`

```
$this->products = ProductsPeer::retrieveByPk(
  $request->getParameter(’id’));
```

Code Snippet 36: The Types index action

```
<table>
  <tbody>
    <tr>
      <th>Id:</th>
      <td><?php echo $products->getId() ?></td>
    </tr>
    <!-- More table rows for type_id, name, added date
    and description -->
  </tbody>
</table>
<form action="<?php echo url_for(’cart/addToCart’) ?>"
  method="post">
  <input type="text" name="amount" value="0" />
  <input type="hidden" name="product_id"
    value="<?php echo $products->getId() ?>" />
  <input type="hidden" name="name"
    value="<?php echo $products->getName() ?>" />
  <input type="submit" value="In den Warenkorb" />
</form>
```

Code Snippet 37: The Show template

```
$criteria = new Criteria();
$criteria->add(CartPeer::PRODUCT_ID,
  $request->getParameter(’product_id’), Criteria::EQUAL);
$criteria->add(
  CartPeer::SESSION_ID, session_id(), Criteria::EQUAL);
$cart = CartPeer::doSelectOne($criteria);
if($cart){
  $cart->setQuantity($request->getParameter(’amount’));
  $cart->save();
} else {
  $cart = new Cart();
  $cart->setProductId($request->getParameter(’product_id’));
  // more setters
  $cart->save();
}
$this->redirect(’cart/index’);
```

Code Snippet 38: The addToCart action

module is rendered.

`executeDisplayForm` is an empty method. The corresponding template renders a form to enter the shipping information. The form points to the `addShippingInfo` action of the `shippingInfo` module. The `executeAddShippingInfo` method queries the `ShippingInfo` table that corresponds to the current session. If a record is found, the fields are updated, otherwise a new record is created. Finally a redirect to the `show` action of the `shippingInfo` module is performed and the `id` of the record is passed in the query string. The `executeShow` method selects the `ShippingInfo` record that corresponds to the id passed in the request. The template simply displays the data entered by the user and renders two links: One link to the `addShippingInfo` action of the `shippingInfo` module, which allows the user to change the data entered and another link to the `processOrder` action of the `order` module.

The `executeProcessOrder` method of the `orders` module first queries the `ShippingInfo` table for the record that belongs to the current session. A new `Order` object is created and populated with the data of the `ShippingInfo` object. Then the `Cart` table is queried for the items that belong to the current session and it is iterated over all the resulting collection. For each `Cart` object a new `OrdersProduct` object is created and populated with the data of the `Cart` object. The `order_id`, which is the foreign key to link the ordered products to the order, is also set for each `OrderProduct` object. Finally the session is destroyed. The template only displays a message to the user that the order has been saved successfully.

4.5 A Comparison between the MVC Framework and Symfony

Symfony is a much more elaborated framework than the MVC framework, although the basic principles are similar. This Section gives an overview of the most significant similarities and differences.

The MVC framework organizes the code into modules. Within a module there are only two types of code artifacts: Model classes and templates. For each model class there is always exactly one template. Each request is handled by one function of a model class. The request parameters are directly retrieved from the `_GET` or from the `_POST` array. The database access is done by executing plain SQL statements. The framework provides a method to pass variables to the template that belongs to the model class. Redirects are done by returning a string with the path to the target event.

Symfony organizes the code into applications and modules. Within a module there is an action class and one or more templates. Each request is handled by an action method in the action class. As opposed to the MVC framework the functions handling a request are not considered to be part of the model but rather an extension to the controller, called frontend controller. The model is stored separately and it is shared amongst all applications. The database access is not done using SQL statements but via ORM. The business logic is supposed to be placed inside the model classes. Request parameters are retrieved from the request object passed to the action method. The framework also provides a method to pass variables to a template. Other than in the MVC framework in Symfony there is one template for each action method. Redirects are done via a redirect function.

As shown before in Subsection 4.4 it is possible to implement the example application using almost the same patterns as for the MVC framework. The most important difference is that the database access is done using an ORM mapping instead of SQL queries and that the functions and objects used are named differently. The patterns presented in Chapter 6 and the transformation program presented in Chapter 8 could be easily transformed to work with the Symfony version of the example application by simply slightly modifying the visitors used. Still, a real world Symfony application might be written using different patterns so a different transformation program would be necessary.

Chapter 5

Ingredients for the Conceptual Design

In this chapter the ingredients for the conceptual design of the reverse engineering process are presented. The most important elements of the Web Modeling Language (WebML), used for the target model, are described. Furthermore the XML and HTML processing tools used to implement the intermediate MVC meta model are introduced.

5.1 WebML

The Web Modeling Language (WebML) is a graphical language with a formal specification for modeling data intensive web applications. A complete WebML Model can be subdivided into the Data Model, the Hypertext Model and the content management model. There is also a commercial tool called Web Ratio [58] available, which supports modeling of WebML models and automatic code generation for Apache Struts [44].

5.1.1 The Data Model

According to Ceri et al. [5] The Data Model aims to provide a conceptual schema of the data used by the application. Therefore the Entity-Relationship model (ER model) [6] is used. The central concept of the ER model are entities.

Entities

An *entity* describes common properties of similar objects in the real world. The actual *objects* (or *instances*) described by the entity are called *population*. The ER model uses a graphical notation for all its concepts. An entity is represented as a rectangle with the entity name at the top.

Attributes

The properties of an entity are modeled via *attributes*. They are graphically represented inside the rectangle of the entity, below the name. In order to distinguish certain instances of an entity one or more attributes must be denoted as part of the *primary key*. If this is only one attribute, its value must be unique for each instance. If the primary key is made up of several attributes, the combination of those attributes must be unique. In WebML it is common practice to model a certain attribute called the *object identifier* (OID) whose only purpose is to serve as a unique identifier for each instance of the entity.

Furthermore attributes may be *typed* meaning that they assume values form well defined domains. WebML supports the following well known data types, common to many programming languages and database systems: `blob`, `boolean`, `date`, `decimal`, `float`, `integer`, `password`, `string`, `text`, `time`, `timestamp` and `url`.

Relationships

Relationships are named semantic connections between entities. A connection between two entities is called a *binary relationship*. A relationship with more than two entities involved is called *N-ary relationship*. However it is possible and encouraged to equivalently express an N-ary relationships by the means of several binary relationships.

A binary relationship has two *relationship roles*, expressing the role each entity plays in the relationship. It can be seen as a directed association from the source entity to the target entity. For example the entity *Book* and the entity *Author* could be connected via a Relationship named *Publication*. The relationship role from book to author could be named *Published_by* and the relationship role from author to book could be named *Publishes*.

Relationship roles can be annotated with minimum and maximum *cardinality constraints*, expressing the minimum and the maximum number of objects of the target entity to which any object of the source entity can be related. Possible values for the minimum cardinality are zero or one. Zero denotes the relationship as optional whereas one expresses a mandatory relationship, meaning that an object of the source entity can not exist without at last an object of the target entity. Possible values for the maximum cardinality are one ore many, the latter depicted as `N`.

Figure 5.1 is an example for a relationship between the two entities Category and Product, modeled with Web Ratio. Each entity has several attributes with certain data types, written after the name of each attribute and the : character. Both entities have an attribute named `OID`, denoted as the primary key, symbolized by a little key symbol on the left side of the attribute's name. The relationship role from product to category is annotated with the maximum cardinality `1` and the relationship role from category to product is annotated with the maximum cardinality `N`.

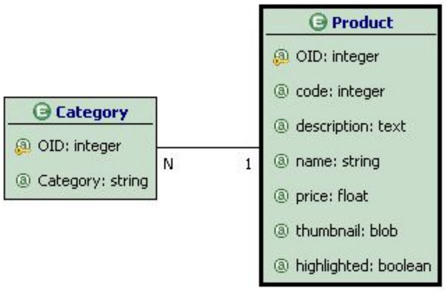


Figure 5.1: A part of an ER diagram with two entities

5.1.2 The Hypertext Model

”The goal of Hypertext Modeling is to specify the organization of the front-end interfaces of a Web application (...) the specification of the hypertext should be maintained at the conceptual level, which means that it should not commit too much to design and implementation details, such as the actual distribution of functionality between the various tiers of a Web application” [5] (pages 77,78).

For the Hypertext Model WebML uses the concepts of Pages, Units and Links, which are organized into areas and site views. Units represent pieces of publishable content that can be placed on a Page. From a user’s perspective they could be seen as extensions to the Data Model. Pages and Units can be connected amongst each other via Links.

Units

There are five types basic types of Units available in WebML:

- *DataUnit*: Refers to a single object of the Data Model.
- *Multidata Unit*: Refers multiple objects of the Data Model.
- *IndexUnit*: Shows a list of objects without showing detailed information.
- *EntryUnit*: Represents a form for the user to enter data.
- *ScrollerUnit*: Provides functionality to browse through lists of objects.

The DataUnit, the MultidataUnit, the IndexUnit and the ScrollerUnit are used to publish content, whereas the EntryUnit is used for content acquisition. The DataUnit and the multidata unit represent the actual content of an object, the IndexUnit’s and the ScrollerUnit’s purpose is to select objects. The DataUnit shows the content of one object, the MultidataUnit shows the content of a set of objects.

The content published by the units is extracted from the Data Model. There are two concepts used in WebML for selecting the content presented by a unit: the Source and the Selector.

- The *Source* is the name of the entity where the content of a unit comes from. The source entity determines the *object type* to be presented.
- The *Selector* is a conjunctions of elementary conditions taken from the entity attributes and the relationship roles in which the entity might be involved. It is used for selecting the *Actual Objects* to be presented.

In Web Ratio units are displayed as rectangles containing the symbol of the unit and some textual information. Each instance of a unit typically has a name which is shown at the top of the rectangle. The symbol of the unit is shown in the middle. In the lower part of the rectangle the name of the entity to which the unit is assigned to is displayed, followed by the Selector conditions displayed in square brackets. Figure 5.2 shows the graphical representation of the five basic units.

The current version of Web Ratio supports some more units for special purposes, but they will not be used in the example application.

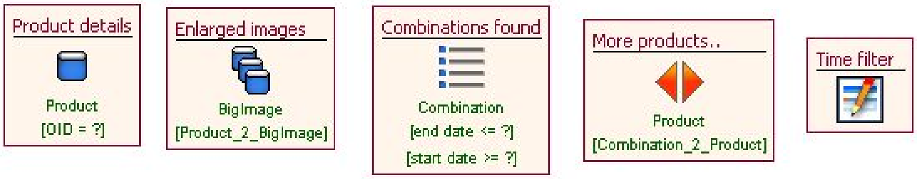


Figure 5.2: The five basic WebML Units. From left to right: DataUnit, MultidataUnit, IndexUnit, ScrollerUnit and EntryUnit

Pages

Pages are the elements delivered to the user who browses the hypertext. Units with a related communication purpose are typically grouped together into Pages. A unit may not be placed outside a Page. There must be one special Page marked as *HomePage* which is the entry point to the application for the user. Figure 5.3 shows the graphical representation of a Page containing two IndexUnits in Web Ratio.

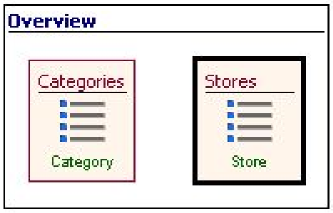


Figure 5.3: A WebML Page

Links

Links are the connection between Pages and units which facilitate the navigation in the hypertext. A Link may contain certain *LinkParameters* in order to transport information from the source to the target of the Link. A unit may have a *Parametric Selector* whose predicates refer to the LinkParameters.

In HTML a Link is either an anchor tag with a href attribute or a submit button of a form. WebML distinguishes between *Inter-Page Links* which connect two different Pages and *Intra-Page Links* that have their source and their target located on the same Page. Links with LinkParameters that transport information are called *contextual* Links, whereas *non-contextual* just trigger navigation but do not transport any information.

Site Views

Large and complex hypertext can be organized into *site views*. A site view is a container for Pages, units and Links.

5.1.3 The Content Management Model

Many web applications perform operations on data. Modeling operations in WebML requires two extensions to the Hypertext Model presented so far. The first extension is the notion of *operation units* which denote either data manipulation operations or the executions of external services. Operation units are triggered via Links coming from different hypertext elements. The second extension applies to the outgoing Links of operation units, which are subdivided into *OKLinks* and *KOLinks*. OK Links are followed after the successful execution of an operation whereas KO Links are followed if the operation fails.

Predefined Operations

There are six basic units for operations on data available in WebML:

- *CreateUnit*: Creates a new instance of an entity.
- *DeleteUnit*: Deletes an instance of an entity.
- *ModifyUnit*: Changes certain attribute values of an entity instance.
- *ConnectUnit*: Creates a new instance of a relationship between two entities.
- *DisconnectUnit*: Deletes an instance of a relationship between two entities.
- *ReconnectUnit*: Changes the source or the target instance of a relationship between two entities.

In order to perform their task of object manipulation, the CreateUnit, the DeleteUnit and the update unit have to be related to a source entity. A Selector is only needed for the DeleteUnit and the update unit as the set of objects to which these operations apply has to be selected.

The ConnectUnit, the DisconnectUnit and the ReconnectUnit do not operate on objects of entities but on relationships between objects of different entities. Therefore they need to be provided with the source relationship role where the operation applies to, a Selector for objects of the source entity and a Selector for the objects of the target entity.

Beside the six operation units mentioned so far there are two more important types of operation units: The SelectorUnit and the IsNotNullUnit.

- The *SelectorUnit* is used to preselect entity objects to be used by other units.
- The *IsNotNullUnit* checks if a certain input parameter has a value or not.

Operation units are placed on a site view, always outside Pages.

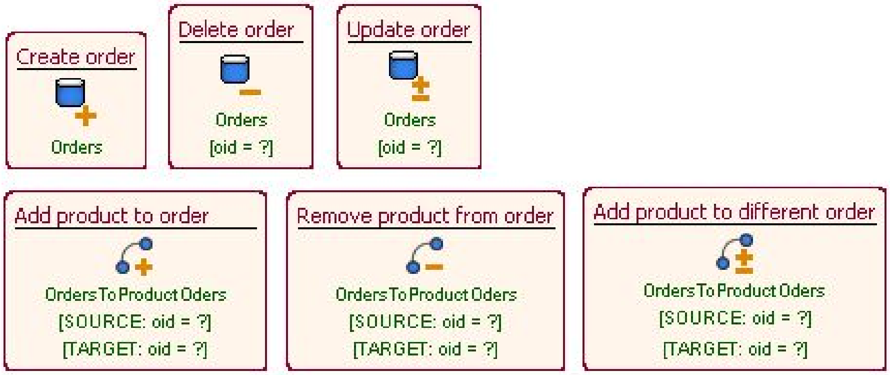


Figure 5.4: WebML operation units. From left to right: CreateUnit, DeleteUnit, ModifyUnit, ConnectUnit, DisconnectUnit, ReconnectUnit.

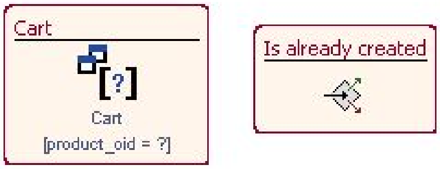


Figure 5.5: A SelectorUnit and an IsNotNullUnit

Module View

I is also possible to separate operation units from the site view by putting them into a special *ModuleView* with certain LinkParameters for encapsulating flows of operation executions. Therefore the content management model is extended with three additional modeling elements.

- *OperationModule*: A container for encapsulation operation units.
- *InputCollectorUnit*: Collects incoming LinkParameters.
- *OutputCollectorUnit*: Collects outgoing LinkParameters.
- *OKCollectorUnit*: Collects outgoing OK LinkParameters.
- *KOCollectorUnit*: Collects outgoing KO LinkParameters.

An OperationModule could be compared to a function in a programming language. InputCollectorUnits in OperationModules represent a similar concept as function parameters. OutputCollectorUnit, OKCollectorUnit and KOCollectorUnit can be compared to return values of functions.

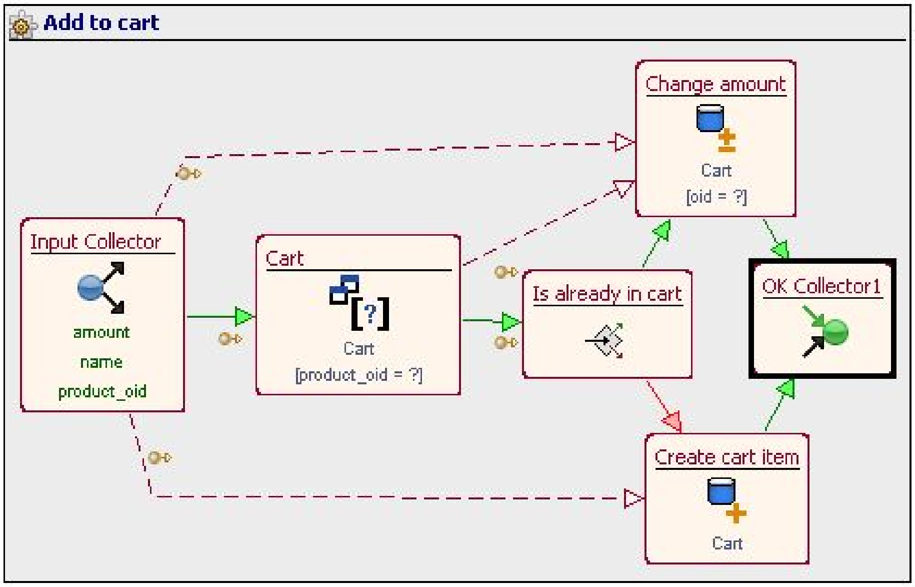


Figure 5.6: An OperationModule with an InputCollectorUnit, a OKCollectorUnit and several operation units.

5.2 XML and HTML Processing Tools

The input sources used for the transformation process described in this thesis are a mixture of SQL, PHP and HTML code. The generated output artifact is an XML file. Hence it is necessary to be able to process four different languages within one process. The processing of the PHP source code, the SQL DDL and DML code is done by JavaCC. For processing the HTML sources the Jericho HTML Parser [16] is used. To create the XML output file the Java Architecture for XML Binding (JAXB) [33] is used.

5.2.1 Jericho

Most of the HTML parsers available are either tree based, such as the Document Object Model (DOM) [4] or event based, such as the Simple API for XML (SAX) [3]. According to the Jericho website [16] Jericho is non of both but ”rather uses a combination of simple text search, efficient tag recognition and a tag position cache”. The main reason why it is used for this thesis is that it is able to recognize different kinds of server tags, including PHP server tags. None of the other parsers tested were able to work with PHP tags as desired. Another advantage of Jericho is that compared to other parsers the interface to query and manipulated tags and elements is easier and more intuitive to use.

In this thesis Jericho is used to transform template files that contain a mixture of HTML and PHP code into a custom XML representation. Therefore a short overview of the classes and methods used is given here.

- An HTML document is represented via an instance of the `Source` class. The constructor takes an `InputStream` delivering the HTML document as an argument.

- A call to the static `register` method of the class `PHPTagTypes` tells the parser to recognize PHP tags.

- An instance of the class `OutputDocument` represents the document resulting from a transformation performed on the `Source` object. The constructor of the `OutputDocument` class takes a `Source` as an argument.

- The `Source` object provides a `getNodeIterator` method to get an iterator over all nodes of the document. All instances returned by the node iterator are of the type `Segment`, which is the superclass for `Element`, `Tag` and `Attribute`.

- An `Element` object represents an HTML element which consists of a start tag, an optional end tag and all the content in between.

- The abstract `Tag` class is the superclass for `StartTag`, which represents a tag such as `<p>`, and for `EndTag`, which represents a tag such as `</p>`.

- The `OutputDocument` class provides a `replace` method with two parameters. The first parameter is the `Segement` to be replaced, the second parameter is the character sequence that replaces the `Segement` passed as the first parameter.

- The `OutputDocument` class also provides a `remove` method that simply removes the `Segment` passed as an argument.

- Another useful method of the Segment class is findAllElements, which returns all Elements of the StartTag type passed as an argument.

With the classes and methods described in this Section it is very easy to transform an HTML file containing PHP tags into any desired representation.

5.2.2 JAXB

The Java Architecture for XML Binding (JAXB) is an Interface used to create mappings between Java objects and XML documents without the need to directly process the XML code. The mapping works in two directions:

1. The creation of an XML document that represents a Java object structure is called *marshalling*.
2. The process of creating a Java object structure based on a XML document is called *unmarshalling*.

JAXB 2.0 is part of the JDK 6 [37]. In this version the rules for the mapping of object states to XML documents can be defined via annotations. Another way to define the mapping is via an XML schema definition. This is not discussed in this thesis.

Mapping and serializing an Object Structure to XML

Code Snippet 39 is an example for a simple Java Bean class as described in the JavaBeans 1.01 specification [42], called Person, annotated with the JAXB annotation @XmlRootElement and another simple Java Bean called Address. The @XmlRootElement annotation is required if the class represents the root element of an XML tree.

The core class of JAXB is JAXBContext. It is used to create either a Marshaller object for writing or an Unmarshaller object for reading. The newInstance method takes the class that represents the root element of the XML tree as an argument. The marshall method takes the root object of the object tree and an OutputStream to which the object structure should be written to as arguments. Code Snippet 40 shows a little test program for creating and marshalling a Person with an Address and the XML output it creates. By default all attributes of a bean object are serialized to XML.

The way of accessing bean attributes for serialization can be configured with the @XmlAccessorType annotation, which has to placed on class level. There are three values of interest.

- @XmlAccessorType(XmlAccessType.FIELD): All non-static attributes.
- @XmlAccessorType(XmlAccessType.PROPERTY): Each JavaBean property.
- @XmlAccessorType(XmlAccessType.PUBLIC_MEMBER): Only public JavaBean properties or public attributes.

```java
import javax.xml.bind.annotation.XmlRootElement;
@XmlRootElement(namespace = "http://tuwien.ac.at/")
public class Person {
  private String name;
  private Address address;
  public String getName(){return name;}
  public void setName(String name){this.name = name;}
  @XmlElement(name="homeAddress")
  public Address getAddress(){return address;}
  public void setAddress(Address address){
    this.address = address;}
}

public class Address {
  private String street;
  private String ZIP;
  public String getStreet(){return street;}
  public void setStreet(String street){this.street = street;}
  public String getZIP(){return ZIP;}
  public void setZIP(String zip){ZIP = zip;}
}
```

Code Snippet 39: Two Java Beans with JAXB annotations

```java
public static void main(String[] args) throws JAXBException {
  Person person = new Person();
  person.setName("Tom Turbo");
  Address address = new Address();
  address.setStreet("Operngasse 22");
  address.setZIP("1010");
  person.setAddress(address);
  JAXBContext context = JAXBContext.newInstance(Person.class);
  Marshaller m =
    context.createMarshaller().marshall(person,System.out);
}

<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<ns2:person xmlns:ns2="http://tuwien.ac.at/">
  <homeAddress>
    <street>Operngasse 22</street>
    <ZIP>1010</ZIP>
  </homeAddress>
  <name>Tom Turbo</name>
</ns2:person>
```

Code Snippet 40: A test program to create the XML document

```
class Book {
  public String name;
  public @XmlAttribute int id;
}
```

```
<book id="123">
  <name>Blackmoor</name>
</book>
```

Code Snippet 41: Using the `@XmlAttribute` annotation

The `@Transient` annotation is used to exclude an attribute from serialization. Attributes annotated with `@XmlAttribute` are serialized as XML attributes and not as XML elements as shown in Code Snippet 41.

Collections are by default serialized as shown in Code Snippet 42. When using the `@XmlElementWrapper` annotation the output looks as shown in Code Snippet 43.

```
class Person {
  public List<String> emails;
}
```

```
<person>
  <emails>abc@def.com</emails>
  <emails>xy@big.at</emails>
</person>
```

Code Snippet 42: Marshaling collections

```
class Person {
  @XmlElementWrapper(name = "
      emails")
  @XmlElement(name = "email")
  public List<String> emails;
}
```

```
<person>
  <emails>
    <email>abc@def.com</email>
    <email>xy@big.at</email>
  </emails>
</person>
```

Code Snippet 43: Marshaling collections using the `@XmlElementWrapper` annotation

Sometimes when a Java object references another object the referenced object should not be a nested child element of the referencing object in the XML representation. The reference should rather be expressed via an attribute or element in the referencing object that contains the id of the referenced object. This can be achieved by using the `@XmlID` and the `@XmlIDREF` annotations. The `@XmlID` annotation is placed on the field that stores the id of an element and the `@XmlIDREF` is placed on the field that references this element.

Chapter 6

Conceptual Design

In this Chapter a conceptual design for the mapping of the MVC web application, created in Chapter 4, to a WebML model is defined. This includes several steps.

- A meta model for the WebML target data structure is defined.
- A meta model for the intermediate MVC data structure is defined.
- Patterns used in the view and in the model are identified and mapped to WebML concepts.

6.1 Defining the Target Data Structure

The target data structure is an XML document that can be viewed and modified with WebRatio. It is convenient to build a graph of Java objects that reflect the target data structure and marshall them to XML using JAXB. In this Section the target data structure is presented using UML class diagrams. References between classes are depicted as directed associations, properties of primitive or built in Java types are modeled as class attributes. Unfortunately the class diagram does not give any information about the JAXB annotations used to map the Java objects to XML, so this information can be found in the descriptions of the class diagrams. If it does not say differently the associations between classes are mapped as elements using the `@XmlElement` annotation. If they are mapped as attributes, using the `@XmlAttribute` annotation, it is mentioned in the description. The class attributes are always mapped as XML element attributes.

Each model element in a WebML model except `WebProject`, `WebModel` and `DataModel` has an `id` attribute and a `name` attribute. Thus an abstract class `WebMLElement` is defined, which contains an `id` and a `name` property, both of type `String`. In the constructor of `WebMLElement` the `id` is initialized with a default universally unique identifier (UUID) provided by the `java.util.UUID` class, via the statement `id = UUID.randomUUID().toString();`. The `id` and the `name` are mapped as attributes and the `id` also has the `@XmlID` annotation. All classes that are used in the target data structure, except the three mentioned above, extend `WebMLElement`. Some of those classes override the `id` with a more meaningful value then the generated UUID.

The root element of a WebML model instance is WebProject which consists of a WebModel and a DataModel as shown in Figure 6.1. The showUnitContent and the showTooltip attributes are not part of the actual WebML model, but they are needed for the graphical representation in WebRatio. Therefore they are part of a different namespace which is http://-www.webratio.com/2006/WebML/Graph.

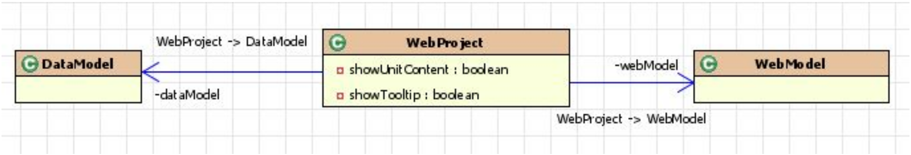


Figure 6.1: The WebProject element

6.1.1 The Data Model

The DataModel is shown in Figure 6.2. It contains a collection of Entity elements and a collection of Relationship elements. An Entity has several EntityAttributes with a key to indicate whether the attribute is part of the primary key. A Relationship has two references to Entity objects, one for the source entity and one for the target entity of the relationship. Those two properties are annotated with @XmlIDREF. A Relationship has two RelationshipRoles, one for the source and one for the target of the relationship. The Relationship role has a single attribute called maxCard which is not shown in the diagram. It is an enum type that can take the values 1 or N and it represents the cardinality of a relationship role.

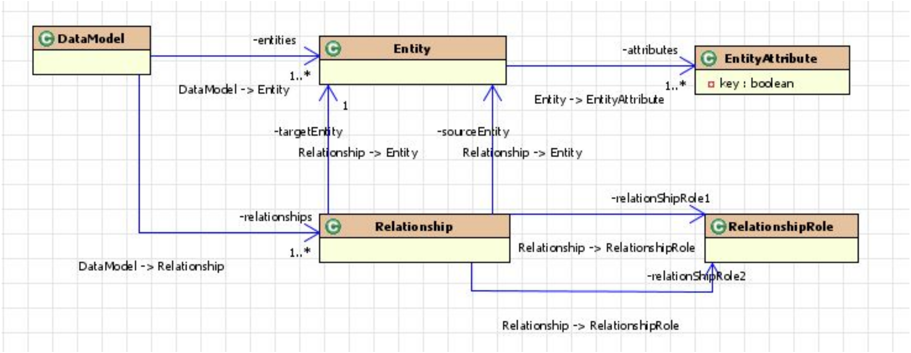


Figure 6.2: The DataModel element

6.1.2 The Web Model

The web model is more complex than the data model. Therefore it is described by several class diagrams. In Figure 6.3 it is shown that a WebModel contains SiteViews and ModuleViews. One of the site views has to be the home site view.

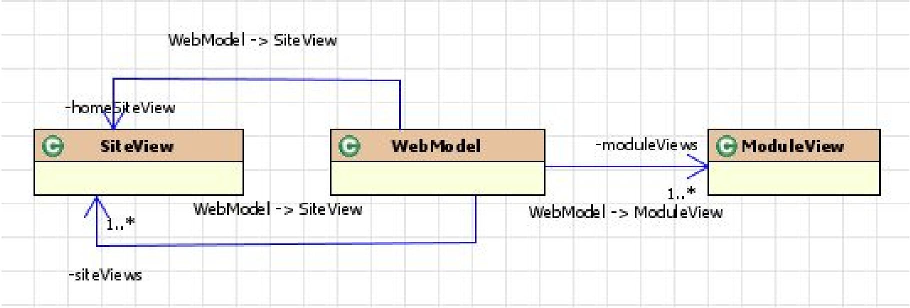


Figure 6.3: The `WebModel` element

SiteView

A `SiteView` contains `Pages` and `OperationUnits`. One `Page` has to be the HomePage. The `OperationUnit` class is declared to be abstract as there are different concrete types of OperationUnit extending from `OperationUnit`.

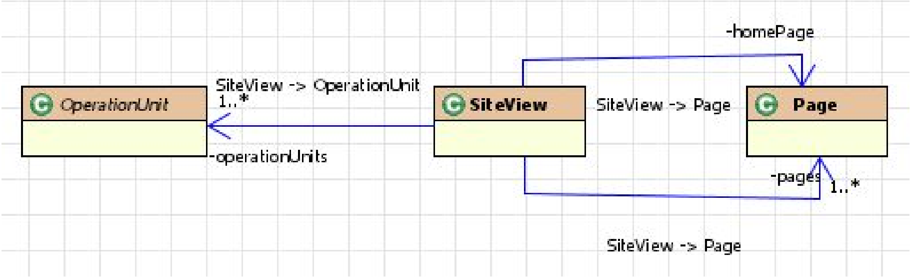


Figure 6.4: The `SiteView` element

OperationUnits

The type hierarchy and the classes related to the OperationUnits are shown in Figure 6.5. The `EntityOperationUnit` class has a reference to `Entity`, which is not shown in the diagram, as The `CreateUnit`, the `ModifyUnit` and the `SelectorUnit` operate on a certain entity. The `ModifyUnit` and the `SelectorUnit` may also have a `Selector`. The `InputCollectorUnit` has several `InputCollectorParameters` and the `OKCollectorUnit` has several `OutputCollectorParameters`.

ContentUnits

In Figure 6.6 it is shown that a `Page` has several `ContentUnits`. `ContentUnit` is an abstract class which is extended by `EntryUnit` and `EntityContentUnit`. As the `DataUnit`, the `MultiDataUnit` and the `IndexUnit` all display the fields of a certain entity instance, `EntityContentUnit` has a reference to `Entity`, which is not shown in the diagram. The `EntryU-`

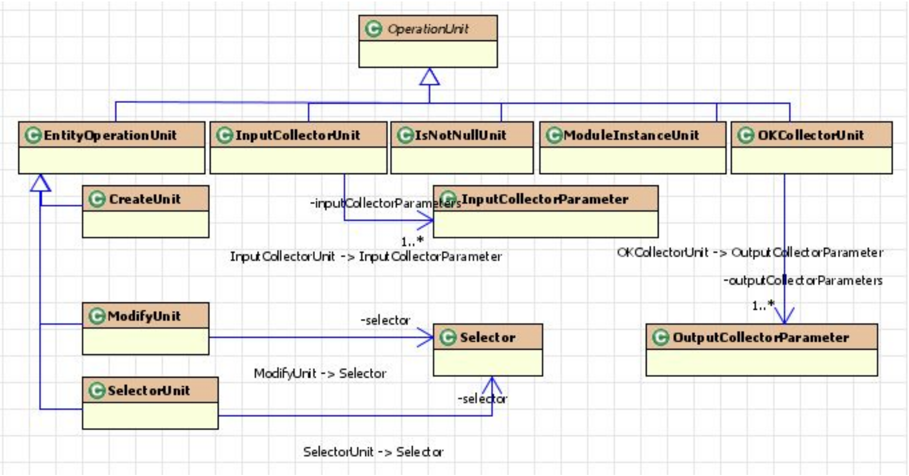


Figure 6.5: The `OperationUnit` element

`nit` has several `Fields`, which represent HTML text input fields. A field can be `modifiable` or not.

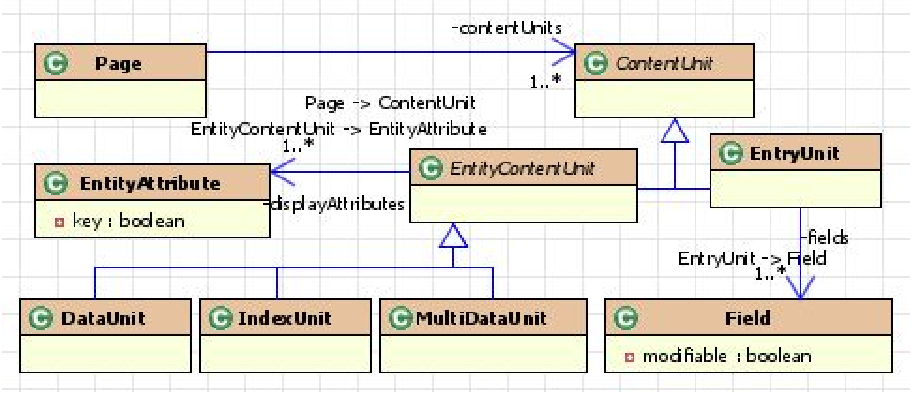


Figure 6.6: The `ContentUnit` element

ModulView

Figure 6.7 shows that the `ModuleView` can have several `OperationModules`. There are other types of Modules beside the `OperationModules` available in WebML, but they are not relevant for this thesis.

Links

A `Link` has a target that can be any `WebMLElement` as shown in Figure 6.8. A `Link` has a `type` which is either `normal` or `transport` (this is not shown in the diagram). If `automaticCou-`

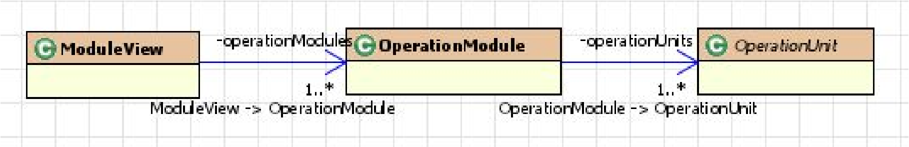


Figure 6.7: The ModuleView element

pling is set to true, it means that no LinkParameters are explicitly modeled but that the parameter coupling is assumed implicitly. A LinkParameter has a source and a target, which is the id of the corresponding elements. The OKLink and the KOLink are extended from Link.

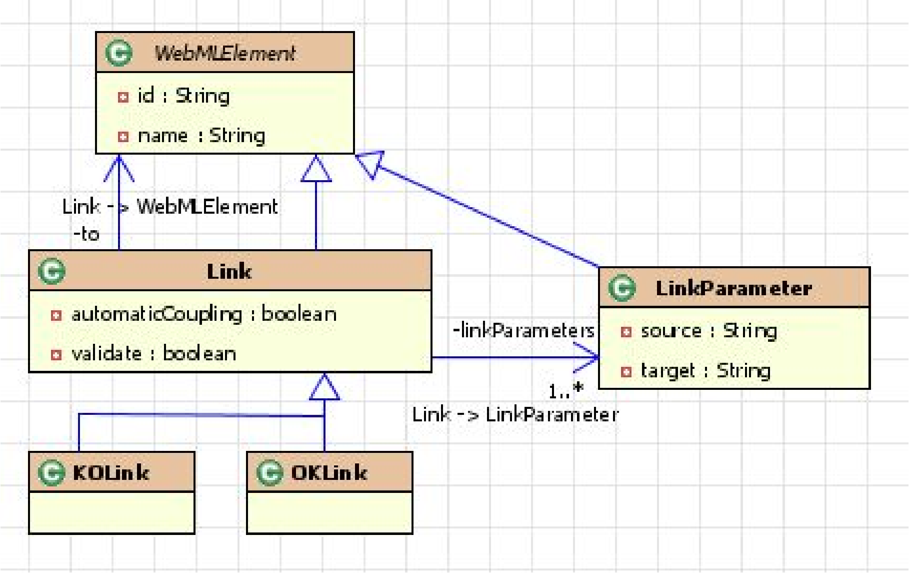


Figure 6.8: The Link element

Conditions

Conditions are used by a Selector to restrict the number of the selected entity instances to those that fulfill the condition. Condition is an abstract class and has to be an AttributesCondition, KeyCondition or RelationshipRoleCondition. The AttributesCondition references the EntityAttributes to be evaluated in the selection. An AttributesCondition also has an enum value for the predicate which has to be either eq for equal, neq for not equal, gt for greater than or lt for lower than. The KeyCondition is used to select entity instances if they either have the primary key passed, using the predicate value in or if they don't have the primary key passed, using the predicate value not in. The RelationshipRoleCondition references a RelationshipRole to either select all entity instances in this role, using the predicate value in, or to select all entity instances not in this role, using the `predicate` value `not in`. The class diagram is shown in Figure 6.9.

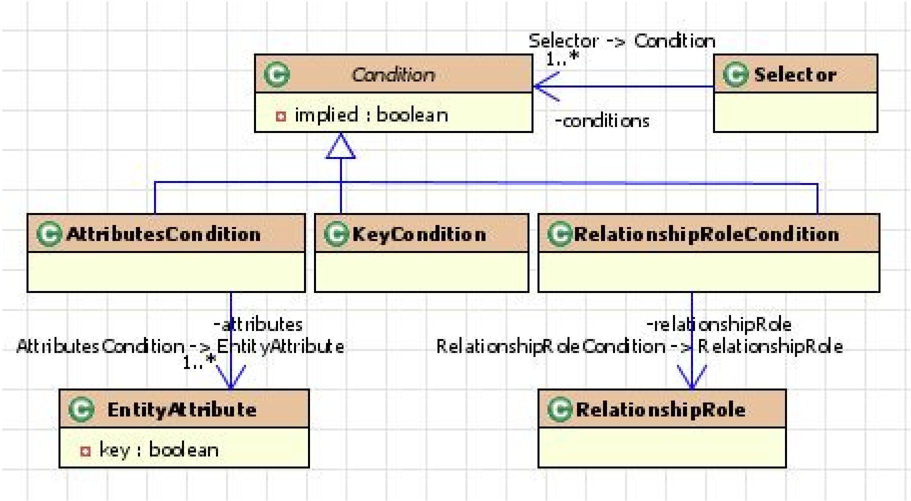


Figure 6.9: The `Condition` element

6.2 Defining a Mapping between the Web Application and the Web Model

After the target data structure is defined the next step is to identify typical patterns in the web application's code and to find suitable patterns in WebML for them.

A challenging aspect of the source platform is that it is made up of two languages: PHP and HTML. The model part of the source application is pure PHP whereas the view part mostly consists of HTML but contains some PHP statement that are crucial for the functionality of the web application. This raises a practical problem in the parsing process of the view: Two different parsers written for two different grammars have to be combined. The approach discussed here is to merge the important elements of the view templates into a third language: XML. This XML representation can easily be mapped to a Java data structure using JAXB.

The model part of the application is pure PHP code that follows certain patterns specified by the MVC framework. When parsing the model code it is suitable to put it into an intermediate Java data structure that reflects those framework patterns. This model data structure and the view data structure can be easily combined to form a data structure that represents the whole web application. The final step is to transform this structure to a WebML model.

6.2.1 Identifying View Patterns

In this Section it is described how certain template patterns can be mapped to WebML elements and how the corresponding intermediate XML representation looks like. The structure of the XML format is depicted in Figure 6.10 and described in the paragraphs below. Elements are shown as classes, element attributes are shown as class attributes.

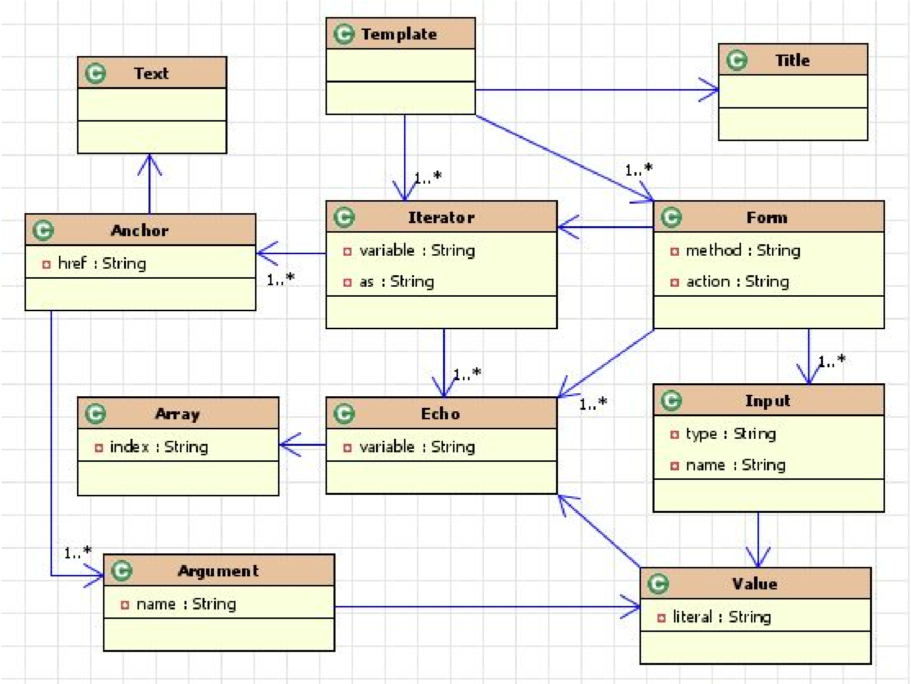


Figure 6.10: The XML data structure for view templates

One thing that most types of templates have in common is that they have a `title` element in the HTML `head` section, containing the title of the web page. This can be directly mapped to the name of a WebML Page and is used for all the patterns described below.

Pattern 1: Index Unit - Version 1

The first pattern to be considered can be found in the categories template presented in Code Snippet 14. The purpose of this template is to display a list of categories that were selected from the `category` table. Each category is rendered inside a `foreach` loop, which is nested inside a `form`. For each category, every attribute is printed out using an `echo` statement. There is also one `input` of type `radio` per category. The `type` and the `name` attributes of the `input` have hardcoded values, but the `value` attribute is dynamically set, using an echo statement. This is a common pattern to represent a WebML IndexUnit.

The first element of interest is the `form`. The action is important to create a Link in the target WebML model. The element is also directly mapped to the XML document. Inside the `form` element there is a `foreach` loop and an `input` element. The `input` element in HTML has a `type`, a `name` and a `value` attribute. A common pattern used with `input` elements is that the `type` and the `name` attributes are hardcoded values, but the value of the `value` attribute is dynamically printed, using an echo statement. Thus the HTML `input` is mapped to an `input` element that only contains a `name` and a `type` attribute to take the hardcoded values. The `value` attribute is mapped to a `value` element nested inside the `input` element. A `value` can either be a hardcoded value that is mapped to a literal element or an echo statement mapped to an echo element.

The other element inside the form element is a foreach loop which is mapped to an iterator element. The name of the array variable over which it is iterated in the loop is mapped to the variable attribute, food_categories in the example. The as attribute holds the name of the iteration variable, which is category. The category variable itself is an array. Inside the loop there are two echo statements, echoing the name and the id values of the category variable. Each echo statement is mapped to an echo element. The variable name is mapped to the variable attribute. In the case of an array value, the echo element has a nested array element with an index attribute that holds the name or number of the array index. There is also an input of type radio rendered for each category. The category's id is echoed as the value of the input. This is mapped to an echo element nested inside the value element. The XML representation of the categories template is shown in Code Snippet 44.

```
<template>
  <title>Categories</title>
  <form method="get" action="/webshop/types">
    <input type="submit" name="selectCategory">
       <value>
         <literal>Kategorie waehlen</literal>
       </value>
     </input>
     <iterator variable="food_categories"
       as="category">
       <echo variable="category">
         <array index="name"/>
       </echo>
       <echo variable="category">
         <array index="id"/>
       </echo>
       <input type="radio" name="category_id">
         <value>
           <echo variable="category">
             <array index="id"/>
           </echo>
         </value>
       </input>
     </iterator>
  </form>
</template>
```

Code Snippet 44: The categories template XML representation

Pattern 2: Index Unit - Version 2

Pattern 2 is applied in the products template presented in Code Snippet 16. It is similar to pattern 1 as it also renders a list of elements that can be selected, but this time instead of using a form, a Link is rendered for each element.

Again, the foreach loop is mapped to an iterator element and the echo statements in the

HTML table row inside the loop are mapped to echo elements. The new and interesting part is the HTML a element. Its href attribute contains the URL path to the target model class and query string containing an echo statement. In order to be able to reasonably work with the information provided, the URL has to be split into several components. The HTML a element is mapped to an anchor element with a href attribute. Other than the href attribute of the a element, the href attribute of the anchor only contains the URL path without the query string. The query string starts after the ? character. It is a list of name-value pairs, separated by the & character. Each name-value pair is mapped to an argument element. The name is mapped to to the nested name element, the value is mapped to a value element. Finally the text contained in an a element is mapped to the text element, nested inside anchor. The iterator element and its content is outlined in Code Snippet 45.

```
<iterator variable="products" as="product">
  <anchor href="/webshop/productDetails">
    <text>Auswaehlen</text>
    <queryArguments>
      <argument>
        <name>product</name>
        <value>
          <echo variable="product">
            <array index="product_id"/>
          </echo>
        </value>
      </argument>
    </queryArguments>
  </anchor>
  <echo variable="product">
    <array index="product_id"/>
  </echo>
  <!-- more echo elements follow here -->
</iterator>
```

Code Snippet 45: The products template XML representation

Pattern 3: Data Unit

Pattern 3 describes a set of echo statements that are not nested inside an iterator. This pattern is used to render the content of a single table row in the database and can be found in the product details template presented in Code Snippet 18 and in the summary template. In WebML it is represented as a DataUnit. In the intermediate XML representation the echo statement might be mapped as echo elements that are directly nested inside the template element or they might be mapped as children of a form element. In the example the echo statements are all children of the form element as shown in Code Snippet 46.

Pattern 4: Multidata Unit

The pattern that describes a MultidataUnit is very similar to the IndexUnit. It consists of an iterator with nested echo elements. The difference to the IndexUnit is that is neither nested inside a form nor does it have any outgoing Links.

Pattern 5: Entry Unit

In a template a WebML EntryUnit is implemented as a HTML form containing input elements of the type text. In the intermediate XML representation this is mapped to input elements that have the type attribute set to text. This pattern is applied in the product details and the shipping info template. The XML representation of the product details template is outlined in Code Snippet 46.

```
<form method="post"
  action="/webshop/productDetails/addToCart">
  <!-- echo elements follow here -->
  <input type="text" name="amount">
    <value>
      <literal>0</literal>
    </value>
  </input>
  <input type="hidden" name="product_id">
    <value>
      <echo variable="product">
        <array index="product_id"/>
      </echo>
    </value>
  </input>
  <!-- another hidden field for the name follows here -->
  <input type="submit" name="null">
    <value>
      <literal>In den Warenkorb</literal>
    </value>
  </input>
</form>
```

Code Snippet 46: The products template XML representation

6.2.2 Identifying Model Patterns

After having defined mappings between the view templates and the WebML ContentUnits the next step is to define mappings between certain patterns found in the business logic of the model classes and the WebML OperationUnits.

Pattern 1: Operation Module

The first thing to strike is that a public function in a model class is a self-contained block of operations that handles a request triggered by an event in the view or by another function. This matches quite well to the WebML concept of an OperationModule. In WebML models it is possible to place OperationUnits either directly on a site view or inside OperationModules. As all the code that makes the business logic has to be inside a model function, all the OperationModules to be reverse engineered are always situated inside an OperationModule and never directly on a site view.

An OperationModule has incoming and outgoing Links that transfer certain parameters. The parameters passed via incoming Links are gathered in an InputCollectorUnit that dispatches the parameters to the OperationUnits inside the OperationModule. Parameters that should leave the OperationModule are passed via OKLinks or KOLinks and can be gathered via OKCollectorUnits or in KOCollectorUnits respectively. In the reverse engineering process it is assumed that all operations terminate successfully so the KOCollectorUnit is not used.

The mapping of the source code to the InputCollectorUnit is straightforward. There is exactly one InputCollectorUnit created for each function. Each parameter taken from the _GET or from the _POST array is mapped to a parameter of the InputCollectorUnit. For the outgoing Links there is exactly one OKCollectorUnit created per function. The outgoing parameters can be identified by statements of the type $this->set(’somename’,$somevariable);. Those are the values passe to the view template. The simplest example can be found in the __default functions of the shippingInfo and the processOrder classes as those functions are empty. In this case the operation modul only contains the InputCollectorUnit and the OKCollectorUnit and no other OperationUnits. An OKLink generated that points from the InputCollectorUnit to the OKCollectorUnit. This pattern is shown in Figure 6.11.

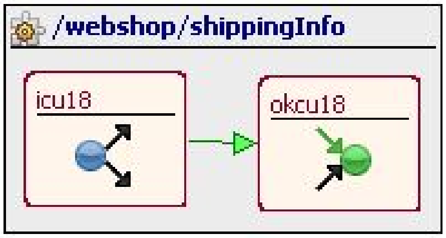


Figure 6.11: An OperationUnit with the default InputCollectorUnit and OKCollectorUnit

Pattern 2: Selector Unit without Input Parameters

One of the simplest patterns used in the example application can be found in the __default method of the categories class presented in Code Snippet 13. In the first line of the function an SQL select statement that selects all entries in the category table is assigned to a variable. This statement can be directly mapped to a SelectorUnit. A SelectorUnit has a reference to an Entity object that can be mapped by analyzing the identifier in the FROM part of the query, which is category in this case. In line two the statement is executed and in the while loop that follows the content of the result is written food_categories array. Finally the food_categories array is passed to the template under the name food_categories. This represents a Link from the SelectorUnit to the OKCollectorUnit. This time the parameter coupled is the primary key of the objects that were selected, which is the id attribute in the example. The primary key attribute is used by the

ContentUnits that were reverse engineered from the templates to determine which object or which objects to display. The OperationModule that represents this example function is depicted in Figure 6.12.

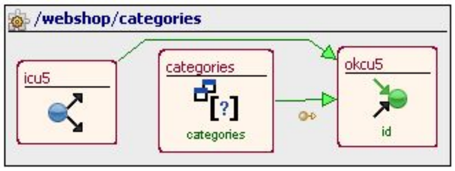


Figure 6.12: A SelectorUnit without input parameters

Pattern 3: Selector Unit with Input Parameters

An extended version of pattern 2 can be found by analyzing the source code of the __default function in the types model class, presented in Code Snippet 15. Again, in the first line of the function an SQL select statement is assigned to a variable but this time the query contains a WHERE part. The whole WHERE clause is mapped to a Selector and each comparison operation between column values in the WHERE clause is mapped to a Condition. Depending on the column that is compared this could be either a RelationshipRoleCondition, a KeyCondition or an AttributeCondition. In the example the value of the category_id column is checked, which is a foreign key attribute for the category table. Therefor the condition to be created is a RelationshipRoleCondition. If the attribute to be evaluated would have been part of the primary key the condition to be used would have been a KeyCondition. The third possibility is that an attribute is evaluated, which is neither part of the primary key nor part of a foreign key. For those cases the AttributeCondition is used.

The SQL statement is a prepared statement as it contains a ? character in the comparison operation with the category_id column. In the next line the statement is transformed into a statement object. In line three the category_id is retrieved from the _GET array, which is mapped to the parameter category_id of the InputCollectorUnit. In line four the statement is executed with the category_id. This is mapped to a Link between the OKCollectorUnit and the SelectorUnit. The category_id parameter is mapped to a LinkParameter of the Link and coupled to the categories2types relationship role via a RelationshipRoleCondition.

Finally all the result rows are written into the food_types array which is than passed to the template under the name food_types. Again this represents a Link from the SelectorUnit to the OKCollectorUnit. This time the parameter coupled is the primary key of the objects that were selected, which is the id attribute in the example. The primary key attribute is used by the ContentUnits that were reverse engineered from the templates to determine which object or which objects to display. OperationModule that represents this example function is depicted in Figure 6.13.

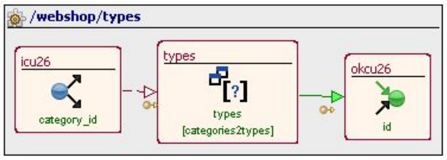


Figure 6.13: A SelectorUnit with input parameters

Pattern 4: Two Selector Units

The __default function of the summary class presented in Code Snippet 21 is an example for a pattern with two SelectorUnits. The function contains two SQL select statements that are prepared and executed. The first statement queries the shippingInfo table, the second statement queries the cart table. Both queries compare the the session_id field with the current session id. Finally the query results are passed to the template. As the session id is not passed as a request parameter there is no reasonable way to model it as WebML LinkParameter. The code is simply mapped to two SelectorUnits. There is a Link created from the InputCollectorUnit to each one of the two SelectorUnits and there is a OKLink created from each one of the SelectorUnits to the OKCollectorUnit. The pattern is shown in Figure 6.14.

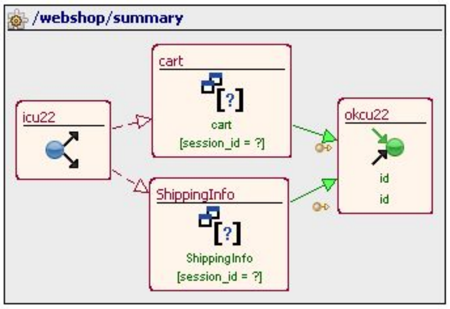


Figure 6.14: Two SelectorUnits

Pattern 5: Selector Unit, IsNotNullUnit, CreateUnit and ModifyUnit

The next pattern is a bit more complicated than the patterns before. It can be found in the addToCart function in the productDetails class, presented in Code Snippet 19 and in the add function in the shippingInfo class. The mapping is described by means of the addToCart function.

The first statement retrieves the current session id and stores it into the session_id variable. In line two, three and four, the values of the amount, name and product_id parameters are retrieved from the _POST array and stored into equally named variables. Again these parameters are mapped to parameters of the InputCollectorUnit. In line five an SQL select statement is directly passed as a parameter to the prepare function. The SQL statement is mapped to a SelectorUnit for the cart entity with a Selector. This time the Selector has two conditions. Both are comparisons with columns that are neither part of the primary key nor part of a foreign key, so they are both mapped to an AttributeCondition. The mapping of the correct variables to parameters of the prepared statement can be done by comparing the order of the comparison operations in the SQL statement with the order of the variables passed to the array that is the second argument for the execute function.

The query returns a single row, which is fetched in line seven and put into the row variable. In an if statement it is checked whether the row already exists or not. This is mapped to an IsNotNullUnit. A OKLink is created, that points from the SelectorUnit to the IsNotNullUnit.

The if part is executed when the row already exists. In this case an SQL update statement is prepared and executed. The update statement is mapped to a ModifyUnit and an OKLink from the IsNotNullUnit to the ModifyUnit. The Entity to be updated is taken from the identifier after the UPDATE keyword. The WHERE part is mapped to a Selector in the same way as it is done for the SelectorUnit. To map the values of a selector a TransportLink from the SelectorUnit to the ModifyUnit is created and for each value a coupled parameter is defined. For the values that are taken from the request a TransportLink from the InputCollectorUnit to the UpdateUnit is created and the parameters are coupled to the Link.

The else part is executed when the row does not exist yet. In this case an SQL insert statement is prepared and executed. The insert statement is mapped to a CreateUnit and a KOLink from the IsNotNullUnit to the CreateUnit is created. The Entity is taken from the identifier after the INSERT keyword. The parameters to be inserted are taken from the product_id, the name, the quantity, and the session_id. The parameter that are taken from the request are coupled to the TransportLink between the InputCollectorUnit and the CreateUnit.

There are no parameters passed to the template but the function returns a value instead. The value determines the next navigation goal. This is needed later for mapping Links between OperationModules. Finally there is an OKLink created between the CreateUnit and the OKCollectorUnit and one OKLink is created between the ModifyUnit and the OKCollectorUnit.The pattern is shown in Figure 6.15.

Pattern 6: Two Selector Units and two Create Units

There is one more pattern left that can be found in the example application. It is used in the processOrder function of the summary class presented in Code Snippet 22 and is an extension of pattern 4. Again the shippingInfo table and the cart table are queried. The results of the query of the shippingInfo table is inserted into the order table. Then the cart table is queried and in a while loop it is iterated over all the cart items. For each item an insert statement into the order_product table is executed using the same order_id as for the insert into the order table. Finally the function returns a Link to /webshop/processOrder. This pattern is mapped to two SelectorUnits, one for the cart entity and one for the shipping_info entity. Then there are two CreateUntis created, one for the order entity and one for the order_product entity. An OKLink is is created that goes from the cart SelectorUnit to the order_product CreateUnit. Another OKLink is is created that goes from the shippingInfo SelectorUnit to the order CreateUnit. The InputCollectorUnit has four outgoing TransportLinks that go to the four Units. Finally there is an OKLink from each one

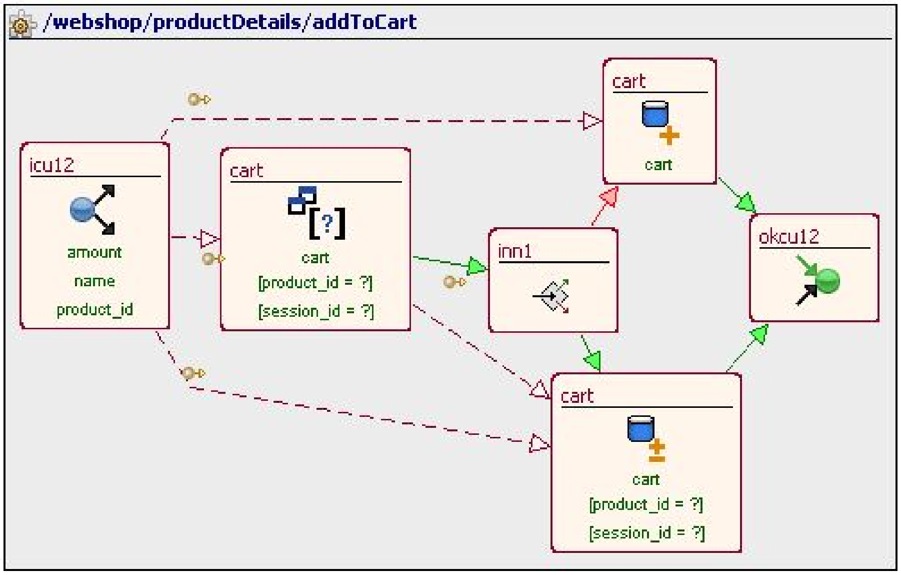


Figure 6.15: A SelectorUnit, an IsNotNullUnit, a CreateUnit and a ModifyUnit

of the `CreateUnits` to the `OKCollectorUnit`. The pattern is shown in Figure 6.16.

6.2.3 Defining an intermediate Data Structure for the Web Application

After having identified the patterns used in the model and in the view of the web application the next step is to define an intermediate data structure that helps implementing the transformation into a WebML model. The goal is to define a data structure that represents the user provided functionality of the web application in a structure that combines the platform and language elements of the MVC framework and the corresponding elements of the WebML target language. This includes also the template data structure presented in Subsection 6.2.1.

The whole web application is represented by the `WebApp` class. Each web application written for the MVC framework consists of several model class - template pairs. To reflect this specification the `Template` class presented in Subsection 6.2.1 is augmented with a property of type `ModelClass`. A `ModelClass` represents a model class file of the MVC framework. The business logic that makes up a model class is contained in its functions. So a `Function` class is introduced that maps to a model class function and each `ModelClass` references a list of `Function` objects. To build the bridge to WebML the data structure also needs references to WebML model elements. A `WebApp` object references the `DataModel` of the web application, the `SiteView` and a `ModuleView` object where the WebML elements are placed. A `ModelClass` object also references the `DataModel` and a `Template` references a `Page`. Each function has a reference to an `OperationModule` and a `ModuleInstanceUnit`. The reason for this is described in Chapter 8. The class diagram of the data structure is shown in Figure 6.17.

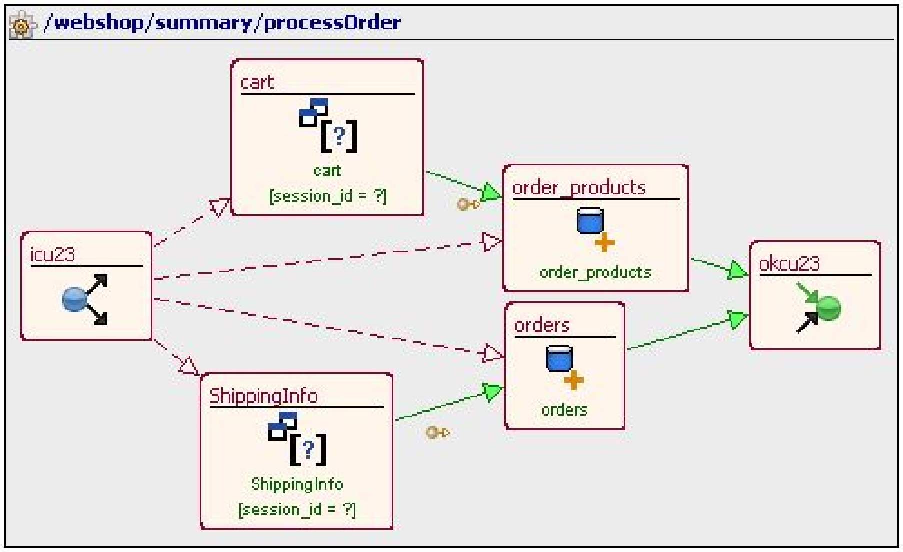


Figure 6.16: Two SelectorUnits and two CreateUnits

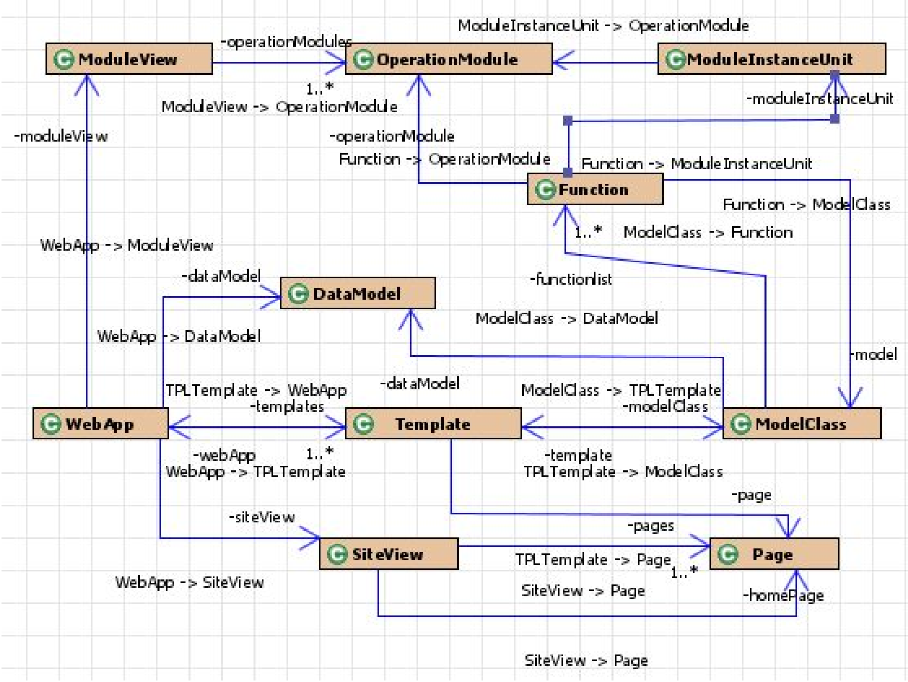


Figure 6.17: The intermediate data structure for describing the MVC application

Tracing Variable Values

To be able to create Links between the OperationUnits and to set the parameter passed it is necessary to trace the values of the variables used in the statements. For example in the `__default` function of the `types` class presented in Code Snippet 15, all statements except the `while` statement and the last statement are assignments of values to variables. For the compilation process it is necessary to subdivide the values in certain types. But other than typical data types used in many programming language such as Integer or String, the types used here should reflect a different aspect that is related to the domain of web applications. For example the first statement is an assignment of a String value. But the interesting thing here is not that the value is a String but rather that the value represents an SQL select query. The third statement on the other hand is an assignment of a value submitted with a get request. For the transformation process, the actual String representation of the value is not important, but it is necessary to know that this value is a request parameter called `category_id`.

The `Function` class has to keep track of all variables and their values defined in the function. Therefor a `Variable` class is created, that has a `name` property and a reference to an object of type `IValue`. The `IValue` interface represents the value of a variable and is implemented by the types `AbstractStringValue`, `RequestParameterValue` and `QueryResultColumnValue`.

The abstract class `AbstractStringValue` represents a string value and has a `value` property to store the corresponding string. It is extended by `SQLStatementValue` which represents a string that is a statement of the SQL data manipulation language. Any other string is considered as a value of type `ArbitraryStringValue`. `SQLStatementValue` is also declared to be abstract as it is further extended by the classes `InsertValue`, `UpdateValue` and `QueryValue`. `InsertValue` represents an SQL insert statement, `UpdateValue` represents an SQL update statement and `QueryValue` represents an SQL select statement.

The `RequestParameterValue` class represents a value passed via a get or a post request.

The `QueryResultColumnValue` class represents the value of a single field of a selected table row.

Each `Function` object references a map with a `String` representing the variable name as key and an `IValue` object as value. The class diagram in Figure 6.18 shows this structure.

Mapping the Selector Unit via the QueryValue Class

In the compilation process the `QueryValue` class is used as value object that helps to build a SelectorUnit. The abstract `SQLStatementValue` class has a reference to the `Entity` object that represents the table that is effected by the SQL statement. A `QueryValue` object references a list of `EntityAttribute` objects that are selected, a `Selector` object and a `SelectorUnit` object. The structure is shown in Figure 6.19.

Mapping the CreateUnit via the InsertValue Class

The `InsertValue` class is used for building a CreateUnit. The effected entity is inherited from `SQLStatementValue` and the effected attributes are referenced in a list of `EntityAttribute` objects. The structure is shown in Figure 6.20

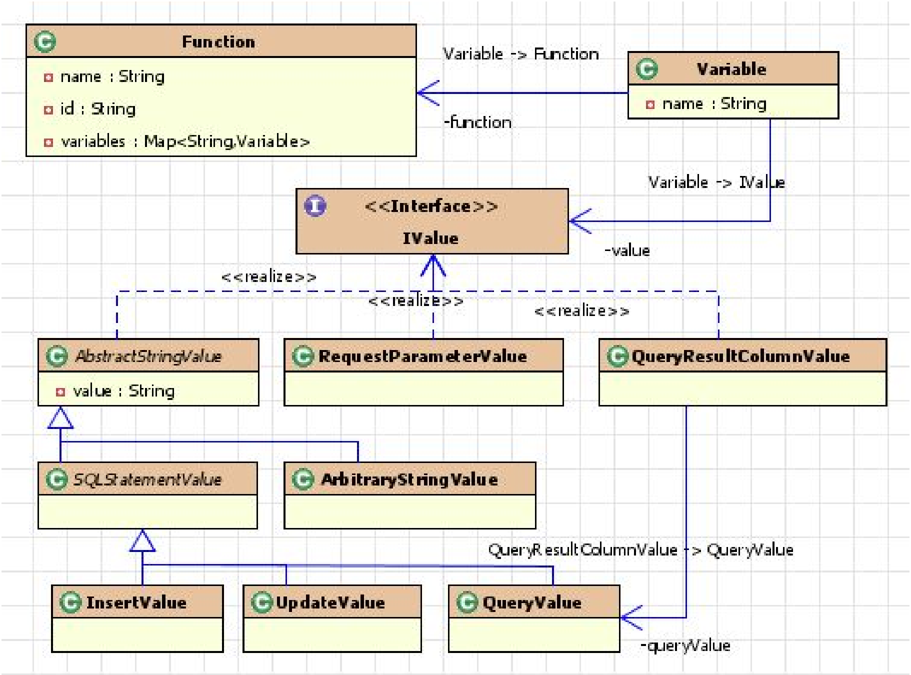


Figure 6.18: The `Function` class and its `Variable` map

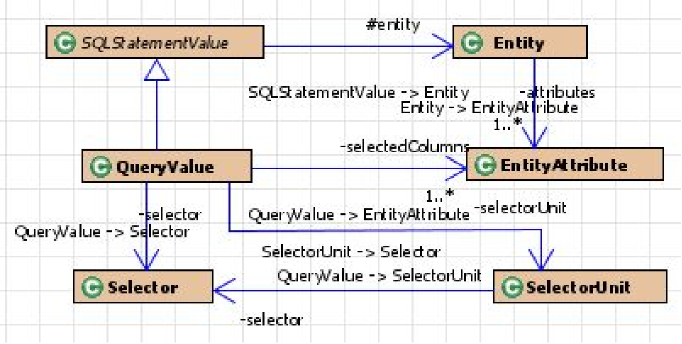


Figure 6.19: The `QueryValue` class

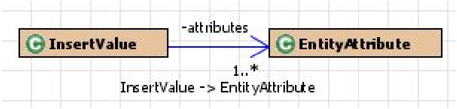


Figure 6.20: The `InsertValue` class

Mapping the Modify Unit to the UpdateValue Class

The ModifyUnit is represented by the `UpdateValue` class. Again the effected entity is inherited from `SQLStatementValue` and the attributes to be updated are referenced in a list of `EntityAttribute` objects. The `UpdateValue` also has a reference to a `Selector`. The structure is shown in Figure 6.21.

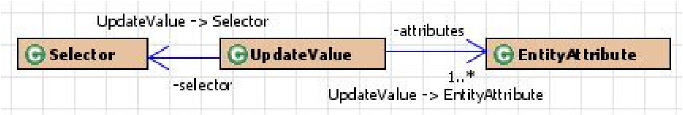


Figure 6.21: The `UpdateValue` class

Chapter 7

Ingredients for the Implementation

In this Chapter the ingredients for the automatic transformation steps in the reverse engineering process are presented. This requires the understanding of the basic principles of how a compiler works, the writing of a parser for the source code and the building of an abstract syntax tree.

7.1 Introduction on how a Compiler works

The major work of a compiler can be subdivided into three major phases:

1. Lexical analysis
2. Syntactic analysis
3. Code generation or execution

7.1.1 The lexical Analysis Phase

In the lexical analysis phase, which is performed by the scanner, the source code is split into atomic pieces called tokens. Examples for tokens are keywords, numbers, strings or operators. Nontokens, e.g. whitespaces, are character sequences that are ignored but that are often used to separate tokens.

7.1.2 The syntactic Analysis Phase

The parser is responsible for the syntactical analysis phase. It is checked whether the source code is syntactical correct meaning that it conforms to the grammar of the source language. The input is transformed into a syntax tree representation. If a grammar rule is violated the parser raises a syntax error.

The grammar of a programming language can be specified unambiguously with the Extended Backus-Naur-Form (EBNF) notation. An overview of how EBNF works is given by M. Garshol [15]. In the EBNF the atomic elements of a text are called *terminal symbols*. A *production rule* assigns a sequence of terminal symbols to a *non terminal* via the = operator. The non terminal is written on the left side of the =. The symbols on the right side can be either terminal symbols that are enclosed by quotation marks or non terminals that can be further evaluated via another production rule. The | operator represents alternatives. Symbols can be grouped to `compound expressions` by putting them into parentheses. An expression may have the quantifiers + or ?. + means that the expression must occur at least once or that it might occur several times. ? means that the expression might occur zero or several times. An expression inside square brackets is optional. Code Snippet 47 describes a language to specify the basic arithmetic expressions in EBNF.

```
expr = number | expr ′+′ expr | expr ′-′ expr  |
  expr ′*′ expr  | expr ′/′ expr  |  ′(′ expr ′)′ | ′-′ expr
  number = digit+ (′.′ digit+)?
digit =
  ′0′ | ′1′ | ′2′ | ′3′ | ′4′ | ′5′ | ′6′ | ′7′ | ′8′ | ′9′
```

Code Snippet 47: The basic arithmetic expressions described in EBNF (Source: [12])

The example defines three production rules: `expr`, `number` and `digit`. An `expr` can be a `number`, two `exprs` connected with an infix operator, an `expr` enclosed by parentheses or an `expr` preceded by a negation operator. A `number` has to consist of at least one digit and can be followed by a decimal point and one or more digits. A `digit` is one of the decimal digit symbols.

7.1.3 The Code Generation or Execution Phase

After the syntactic analysis phase is finished an internal representation of the program exists. The compiler can now generate the code that conforms to the target language or, in case of interpreters, execute the internal representation directly.

7.2 Working with JavaCC

The Java Compiler Compiler (JavaCC) [34] is a popular parser/scanner generator for Java that enables the writing of compilers or interpreters. A compiler performs a transformation from a program written in a source language into a semantically equivalent program in a target language. O. Enseling [12] gives a good overview on how to write a compiler with JavaCC.

JavaCC allows the programmer to define grammars in a way similar to EBNF mixed with language elements of the Java programming language. Code Snippet 48 is an example of how to write a simple calculator program with JavaCC. A JavaCC grammar file has the file extension `jj`.

The `options` section at the beginning contains different configuration options for the grammar. In this example the `LOOKAHEAD` option is set to `2`, telling the parser to always look ahead two tokens further in the input stream. This is necessary if the choice which rule to evaluate next is not clear to the parse by only considering the next token.

The block between `PARSER_BEGIN` and `PARSER_END` contains the definition of the parser class which is called `Calculator` in this example. The parser class contains a `main` method to initiate the parsing process by creating an instance of the parser class. The argument passed to the constructor must be an `InputStream`. By passing `System.in` the parser takes its input from

```
options{LOOKAHEAD=2;}
PARSER_BEGIN(Calculator)
public class Calculator {
  public static void main(String args[])
    throws ParseException {
    Calculator parser = new Calculator(System.in);
    while (true){ parser.parseOneLine(); }
  }
}
PARSER_END(Calculator)
SKIP : { " " | "\r" | "\t" }
TOKEN:
{ < NUMBER: (<DIGIT>)+ ( "." (<DIGIT>)+ )? >
  | < DIGIT: ["0"-"9"] > | < EOL: "\n" > }
void parseOneLine():
{ double a; }
{ a=expr() <EOL> { System.out.println(a); }
  | <EOL>  | <EOF> { System.exit(-1); } }
double expr(): { double a; double b; }{
  a=term()
  ( "+" b=expr() { a += b; }
    | "-" b=expr() { a -= b; } )*
  { return a; }
}
double term(): {double a; double b;}{
  a=unary()
  ( "*" b=term() { a *= b; }
    | "/" b=term() { a /= b; } )*
  { return a; }
}
double unary():{double a;}{
  "-" a=element() { return -a; }
  | a=element() { return a; }
}
double element():{Token t; double a;}{
  t=<NUMBER> {return Double.parseDouble(t.toString());}
  | "(" a=expr() ")" {return a;}
}
```

Code Snippet 48: A simple calculator written in JavaCC [12] the command line. All the Java code needed for the scanning and the parsing process is generated out of the grammar definition that follows

The SKIP section contains all nontokens that should be ignored by the parser which are whitespaces in this case. The TOKEN section contains the tokens of the language which are digits and numbers, as well as the end of line character.

The rest of the file contains the production rules. There are more production rules in Code Snippet 48 then in the EBNF grammar in Code Snippet 47. This is because the grammar in Code Snippet 47 is ambiguous. For instance the expression 1+2*3 could be either matched as expr*3 or as 1+expr. As JavaCC grammars have to be unambiguous the expr rule is split into four rules called expr, term, unary, and element.

All the production rules are transformed into methods of the parser class by JavaCC. Therefore production rules have return values and might also have parameters. They can also contain Java arbitrary statements. All Java statements have to be written inside curly braces. All variables used have to be declared inside the curly braces coming after the colon.

The first production rule is called parseOneLine. It matches three possible input sequences:

1. An expr followed by an end of line character. In this case the return value of expr is stored into the variable a and its value is written to System.out.

2. An empty line. In this case nothing happens.

3. An end of file character. In this case the program terminates.

The calculation of the expressions is directly performed inside the production rules.

7.2.1 Lexical States

The lexical analysis phase in JavaCC is done by the token manager. The lexical specification of JavaCC is organized into lexical states and the token manager is in one lexical state at any moment. Each lexical state has a name and defines different tokens with different regular expressions. There is a standard lexical state called DEFAULT and the token manager is initially in this state. After consuming a token the token manager might be switched to another state. The benefit of different lexical states is that special parts in a source document that follow different grammar rules than the rest of the document can be handled differently. A good example for the use of lexical states are comments inside program code. Comments usually contain arbitrary text, that does not have to match to any specific grammar rules. In Java and PHP a block comment is everything between the characters /* and */. A lexical state called COMMENT could be defined. If the token manager consumes the token /* it switches to from the DEFAULT lexical state to the COMMENT state. In this state it matches any text that does not contain the character sequence /* or */ as one token. If it consumes the token */ it switches back to the DEFAULT state. A detailed description about lexical states and the token manager can be found in the TokenManager MiniTutorial on the JavaCC website [36].

7.3 Working with JJTree

In the example presented in Subsection 7.1 any input sequence is directly interpreted during the parsing process. Very often it is necessary to build an abstract syntax tree (AST) first and then generate the target data structure out of it. In JavaCC this is supported via JJTree [35].

7.3.1 Building an Abstract Syntax Tree

JJTree works as a pre-processor for the actual JavaCC processor. The grammar file is written as usual but put into a file with the extension `jjt`. This file is passed to the JJTree preprocessor, which produces a JavaCC file, enhanced with the code to build a syntax tree. For each non terminal symbol an instance of the class `SimpleNode` is created by default when parsing the source code. The result is an object tree reflecting the structure of the parsed document, but this tree does not contain any information about the actual values that where parsed.

To get a useful tree structure it is necessary to have custom node objects. JJTree is able to generate and use the necessary classes automatically by enhancing the production rules with the class name to be generated. Furthermore the option `MULTI` in the `options` block has to be set to `true`. For example in Code Snippet 49 the `#ColumnDeclaration` part triggers the generation of a `ASTColumnDeclaration` class that extends from `SimpleNode`. The prefix `AST` can be changed via an option. Often it is not necessary to create a node in the AST for every production rule. To avoid the creation of an AST object for production rules that do not declare a custom AST class the option `NODE_DEFAULT_VOID` has to be set to `true`.

```
void ColumnDeclaration() #ColumnDeclaration : {Token t;}{
  t = <NAME> {jjtThis.setColumnName(t.image);}
}
```

Code Snippet 49: A production rule that defines a custom class for the AST

The generated class can no be supplemented with custom methods and properties as shown in Code Snippet 50.

```
public class ASTColumnDeclaration extends SimpleNode {
  ...
  private String columnName;
  public String getColumnName() {
    return columnName;
  }
  public void setColumnName(String columnName) {
    this.columnName = columnName;
  }
}
```

Code Snippet 50: A production rule that defines a custom class for the AST

The production rule in Code Snippet 49 declares the variable `t` of the type `Token`. The line `t =`

<NAME> assigns the actual value of the token <NAME> to t. The string representation of the token is retrieved via t.image. The jjtThis variable references the current node object, ASTColumnDeclaration in this case. So the statement jjtThis.setColumnName(t.image); stores the value of the current token in the current node object.

7.3.2 The Visitor Pattern

Although it would be possible to use the node objects directly to create the target data structure, this option often comes with some drawbacks:

- Very often it is quite complex to write the grammar in a way that it is very similar to the target data structure.
- Changes in the target data structure influence the grammar.
- The syntax tree should be as lightweight as possible to speed up the parsing process.

Another problem is that the objects in the syntax tree are of different types. For instance a node object representing a column in a database table definition might have a DataType property and a corresponding getDataType method. Another node object representing a database table does not have a DataType property but might have a TableName property. When traversing the syntax tree it would be necessary to cast the objects to their actual type before being able to call their special methods, which is not a good programming style.

A solution to this problem is to apply the visitor pattern presented by Gamma et al. [14]. First of all each class used in the syntax tree needs an accept method that takes an object of a visitor class as an argument. This visitor class has to contain a visit method to which the visited object passes itself as an argument. JJTree supports the visitor pattern. To generate the required classes automatically the VISITOR option in the options block has to be set to true. This triggers the JJTree pre-processor to add the method jjtAccept to the SimpleNode class and to generate an interface for the visitor called <NameOfTheParser>Visitor. This interface declares a visit method for each node class. An example is given in Code Snippet 51.

```
public interface DDLVisitor {
  public Object visit(SimpleNode node, Object data);
  public Object visit(DDLTable node, Object data);
  public Object visit(DDLColumnDeclaration node, Object data);
  ...
}
```

Code Snippet 51: A visitor interface

It is good practice to create an abstract class that provides a default implementation of all visit methods by returning null. The next step is to implement one or more concrete visitor classes that perform the actual compiler work. The AST has to be traversed recursively and for each node the jjtAccept method has to be called. A comprehensive example of how to apply the visitor pattern is given in Subsection 8.1.3.

Chapter 8

Implementation

In this Chapter the implementation of the reverse engineering program is presented. The program is able to recognize certain patterns in a web application written for the MVC framework described in Section 4.1 and to transform them into a WebML model. It utilizes the technologies and libraries presented so far. The whole transformation process is implemented within one program. The tasks performed by this program can be roughly divided into four steps.

1. Loading the source artifacts.
2. Building a WebML Data Model using the source artifacts.
3. Building a WebML Web Model using the Data Model and the source artifacts.
4. Putting together the Data Model and the Web Model into a WebML Web Project and serialize it to XML.

Those steps are initialized from the `main` method of a class called `PHP2WebML`.

Step one is the creation of an in-memory representation of the web application's file system structure, together with the sql dump file that contains the `create table` statements for the database of the MVC application and the XML file containing the relationship mappings for the database tables as described in Subsection 8.1.2). This is done in a helper class, which is using a modified version of the algorithm to create an in memory file system tree presented in an article on the Java Boutique website [18]. The helper class has a `build` method that takes a `java.io.File` object [37] representing the root directory with the source artifacts. The method recursively traverses all the directories and files contained in the root directory and returns a tree of `FS_Directory` and `FS_File` objects. Those objects provide methods to access meta information and the content of each directory or file in the file system tree.

Step two is the creation of the Data Model out of the sql dump. This is done in the `compile` method of a class called `DDLCompiler`. How the `DDLCompiler` is implemented is described in section 8.1.

Step three is to create the Web Model using the Data Model together with the model and view files of the MVC application. This is done in the `compile` method of a class called `PHPCompiler`. How the `PHPCompiler` is implemented is described in section 8.2.

Step four is to put together the Data Model and the Web Model into a WebML Web Project and serialize it to XML, which is described at the end of section 8.2.

8.1 Transforming the Database Schema into a WebML Data Model

The Data Model is the base for a WebML model and it can be reverse engineered quite easily out of the database schema of the web application. Three steps are necessary to transform the MySQL dump file with the table create statements of the web application into a WebML Data Model:

1. Writing a grammar file for JJTree and enhance the generated node classes with custom properties.
2. Providing additional information about the relationships between the tables. This is necessary as no foreign key constraints are used in the example application.
3. Traversing the AST created by JJTree and compiling the data into the target data structure.

8.1.1 A Grammar for Create Table Statements

As the database technology of choice is MySQL a grammar of the MySQL data definition language (DDL) is necessary. For the example it sufficient to provide a grammar that supports the syntax of the create table statement as described in the MySQL Reference Manual [39]. The compiler for the Data Model is using JavaCC and JJTree. First of all a JJTree file called `DDL2WebML.jtt` is created.

The prefix for the node classes generated by JJTree are is set to `DDL` via the line `NODE_PREFIX = "DDL";` in the `options` section. The next thing of interest are the tokens used in the grammar. As there are quite a lot of them only some of the most common ones are listed in Code Snippet 52.

```
SKIP : { " " | "\n" | "\r" | "\r\n" }
TOKEN : {
<CREATE: "CREATE"> | <TEMPORARY: "TEMPORARY">
| <TABLE: "TABLE"> | <IF_NOT_EXISTS: "IF NOT EXISTS">
| <BIT: "BIT">  // more MySQL datatypes follow here
| <USING: "USING">  | <ENGINE: "ENGINE">
| <SEMCOL: ";"> | <CONSTRAINT: "CONSTRAINT">
| <NOT_NULL: "NOT NULL">  | <NULL: "NULL">
| <DFLT: "DEFAULT">| <AUTO_INCREMENT: "AUTO_INCREMENT">
| <UNIQUE: "UNIQUE"> | <PRIMARY: "PRIMARY">
| <KEY: "KEY"> | <FOREIGN_KEY: "FOREIGN KEY">
| <REFERENCES: "REFERENCES"> | <CASCADE: "CASCADE">
| <LB: "("> | <RB: ")"> | <COMMA: ",">
| <DIGITS: (["0"-"9"])+ >
| <NAME : (["0"-"9","a"-"z","A"-"Z","-","_","\"","′","`"])+ >
| <STRING : (˜[])>
}
```

Code Snippet 52: The Tokens used in `DDL2WebML.jtt`

The first production rule is `Start()` which consists of zero or more `CreateTable()` statements followed by an end of file character. It returns an object of `DDLStart` which has been generated by JJTree and which is used by the compiler as a starting point for processing the tree. The two production rules are shown in Code Snippet 53.

```
DDLStart Start() #Start : {}{
  (CreateTable())* <EOF> { return jjtThis; }
}
void CreateTable() #Table : {Token t;}{
  <CREATE> (<TEMPORARY>)? <TABLE> (<IF_NOT_EXISTS>)? t=<NAME>
  {jjtThis.setName(t.image.replace("`",""));}
  <LB>CreateDefinition()(<COMMA>CreateDefinition())*<RB>
  (TableOption())* (<SEMCOL>)?
}
```

Code Snippet 53: The `Start` and the `CreateTable` production rules

The only token of interest for the reverse engineering process in the `CreateTable()` production rule is the name of the table. The `DDLTable` class which is generated by JJTree is enhanced with a `name` property. The value of the `<NAME>` token is stored in each `DDLTable` via the line `jjtThis.setName(t.image.replace("`",""));`. The string replace method which removes all occurrences of ` is called because the MySQL Dump tool adds the ` around all character values but they are not part of the actual name.

The table name has to be followed by one or more create definitions which have to be separated by commas. All create definitions have to be between an opening an a closing brace. The `CreateDefinition()` is shown in Code Snippet 54.

```
void CreateDefinition(): {}{
  ColumnDeclaration() | LOOKAHEAD(2) PrimaryKeyConstraint()
  | IndexDeclaration() | LOOKAHEAD(2) UniqueConstraint()
  | SpatialOrFulltextDeclaration() | ForeignKeyConstraint()
  | CheckExpression()
}
```

Code Snippet 54: The `CreateDefinition` production rules

A `CreateDefinition` contains a choice of production rules which are either `ColumnDeclaration`, `PrimaryKeyConstraint`, `IndexDeclaration`, `UniqueConstraint`, `SpatialOrFulltextDeclaration`, `ForeignKeyConstraint` or `CheckExpression`. The most important production rule is `ColumnDeclaration()` which is outlined in Code Snippet 55.

A `ColumnDeclaration` consists of the column name, represented by the `<NAME>` token, a data type and several keywords to further specify the properties of the column (such as if the column is nullable, if its value is auto-incremented etc.) but those specifiers are not important for the reverse engineering process. The `DDLColumnDeclaration` class is enhanced with a

```
void ColumnDeclaration() #ColumnDeclaration : {Token t;}{
  t = <NAME>
  {jjtThis.setColumnName(t.image.replace("`",""));}
  DataType() (<NOT_NULL> | <NULL>)?
  // more keyword tokens follow here
}
```

Code Snippet 55: The `ColumnDeclaration` production rule

`columnName` property and set to the value of the current column name. Again, all ` characters are removed first. The next production rule of interest is `DataType()` which represents the MySQL data type declaration of the column. The production rule simply contains a choice of tokens representing the data types. The value of the token matched is stored in the `dataType` property of a `DDLDataType` object.

Returning to the `ColumnDeclaration()` production rule the next production rule in the choice after `ColumnDeclaration()` is `PrimaryKeyConstraint()`. A primary key constraint has to contain the keywords `primary key` followed by an optional `IndexType` which is not important for the reverse engineering process. It is followed by a comma separated list of type `IndexColumnName()`. The `IndexColumnName()` production rule represents a column that is part of the primary key and it returns the value of the column name. The `DDLPrimaryKeyConstraint` class maintains a list of all columns that are part of the primary key and the String returned by the `IndexColumnName()` production rule is added to this list. The `IndexOption()` and the `IndexType()` are not relevant for the reverse engineering process. The `PrimaryKeyConstraint()` and the `IndexColumnName()` production rules shown in Code Snippet 56.

```
void PrimaryKeyConstraint() #PrimaryKeyConstraint :
{String key;}{
  (<CONSTRAINT> (<STRING>)?)? <PRIMARY> <KEY> (IndexType())?
  <LB>key = IndexColumnName(){jjtThis.getKeys().add(key);}
  (<COMMA>key = IndexColumnName(){jjtThis.getKeys().add(key);})*
  <RB> (IndexOption())*
}

String IndexColumnName(): {Token t;}{
  t = <NAME> (<LB><DIGITS><RB>)? (<ASC> | <DESC>)?
  {return t.image.replace("`","");}
}
```

Code Snippet 56: The `PrimaryKeyConstraint` production rule

Finally it is important to know, how the developer provided part of the parser class looks like. The class is called `DDL2WebML` and it contains a `compile` that takes and `InputStream` as an argument which is used to build the AST. The section containing the parser class definition is outlined in Code Snippet 57.

```
PARSER_BEGIN(DDL2WebML)
// package declaration and import statements follow here
public class DDL2WebML {
  public static DDLStart compile(InputStream in) throws ParseException {
    DDL2WebML p = new DDL2WebML(new DataInputStream(in)) ;
    return p.Start();
  }
}
PARSER_END(DDL2WebML)
```

Code Snippet 57: The `PrimaryKeyConstraint` production rule

8.1.2 Defining the Relationships between the Tables

One way to reverse engineer the relationships between the table would be to analyze the foreign key constraints defined in the create table statements. But in the example application there are no foreign key constraints defined. Therefore the user has to provide this information. This is done via a very simple XML format.

The root element of this XML format is the `references` element. It contains an arbitrary number of `reference` elements, each one representing the relationship between two tables. A `reference` element has the following child elements in any order:

- `sourceTable` contains the name of the table, which represents the source end of the relationship.

- `targetTable` contains the name of the table, which represents the target end of the relationship.

- `sourceToTargetCardinality` represents the cardinality at the target side of the relationship. The element value has to be either `1` or `N`.

- `targetToSourceCardinality` represents the cardinality at the source side of the relationship. The element value has to be either `1` or `N`.

- `targetJoinColumn` contains the name of a column in the target table which is part of the foreign key in this relationship.

To define all references in the database schema of the example application, four `reference` elements for the following references are necessary: `category` to `type`, `type` to `product`, `product` to `orer_product` and `orer_product` to `order`. The relationship definition between the `type` and the `product` table is shown in Code Snippet 58. The XML document is stored in a file called `relationships.xml`.

The file containing the references XML document can be simply mapped to objects of the two classes `References` and `Reference` using JAXB. The `References` class is shown in Code Snippet 59 and the `Reference` class is outlined in Code Snippet 60.

```
<reference>
  <sourceTable>type</sourceTable>
  <sourceToTargetCardinality>N</sourceToTargetCardinality>
  <targetTable>product</targetTable>
  <targetToSourceCardinality>1</targetToSourceCardinality>
  <targetJoinColumn>type_id</targetJoinColumn>
</reference>
```

Code Snippet 58: Defining a relationship between the `type` and the `product` table

```
@XmlRootElement(name = "references")
@XmlAccessorType(XmlAccessType.PROPERTY)
public class References {
  private List<Reference> references = new ArrayList<Reference>();
  @XmlElement(name = "reference")
  public List<Reference> getReferences() {
    return references;
  }
  public void setRelationships(List<Reference> references) {
    this.references = references;
  }
}
```

Code Snippet 59: The References class with JAXB annotations

```
@XmlAccessorType(XmlAccessType.PROPERTY)
@XmlRootElement
public class Reference {
  private String sourceTable;
  private String targetTable;
  private String sourceToTargetCardinality;
  private String targetToSourceCardinality;
  private String targetJoinColumn;
  // getters and setters follow here
}
```

Code Snippet 60: The References class with JAXB annotations

8.1.3 Implementing the Compiler

The next step is to write a compiler that takes the JJTree parser, the MySQL dump file and the user provided XML file defining the relationships in the database schema and transforms it into a WebML Data Model. This is done in the `DDLCompiler` class. The class has a public static `compile` method that takes an `InputStream` providing the MySQL Dump file as an argument. As shown in Code Snippet 61, first of all the `InputStream` object is passed to the static `compile` method of the `DDL2WebML` parser class that has been presented in Code Snippet 57. The method returns a `DDLStart` object which is the root of the AST representing the create table statements. Then a new `DDLVisitor` is created and passed to the `traverseNodes` method together with the AST. The `traverseNodes` method recursively goes through all the levels of the AST and calls the `jjtAccept` method with the visitor as an argument on each node, as shown in Code Snippet 62. The implementation of `DDLVisitor` is described below.

```
public static DataModel compile(InputStream in)
  throws ParseException, JAXBException, FileNotFoundException{
  DDLStart dataModelAST = DDL2WebML.compile(in);
  DDLVisitor visitor = new DDLVisitor();
  traverseNodes(dataModelAST, visitor);
  \\...
```

Code Snippet 61: The `DDLCompiler`

```
private static void traverseNodes(SimpleNode node, DDLVisitor visitor) {
  for(int i=0;i<node.jjtGetNumChildren();i++) {
    SimpleNode sn = (SimpleNode)node.jjtGetChild(i);
    sn.jjtAccept(visitor, null);
    traverseNodes(sn, visitor);
  }
}
```

Code Snippet 62: The `traverseNodes` method

The DDLVisitor

To extract the interesting information out of the AST, the visitor pattern, presented in Subsection 7.3.2, is applied. The visitor class is called `DDLVisitor` and it extends `AbstractDDLVisitor`. The `AbstractDDLVisitor` provides default implementations of all `visit` methods that simply return `null`.

As shown in Code Snippet 63, the `DDLVisitor` contains a static map that maps the MySQL data types to WebML data types. The keys of the map are the names of the MySQL data types as `String` and the value of the map is an `enum` called `DataType`. This `enum` contains all WebML data types annotated with the JAXB annotation `@XmlEnumValue`. This annotation takes a String as an argument, whose value is later used to marshall the WebML data types to XML.

The `DDLVisitor` contains a list of all `Entities` that were created when parsing the AST. The `currentEntity` property references the last `Entity` created and the `currentEntityAttribute` references the last `EntityAttribute` created.

```
public class DDLVisitor extends AbstractDDLVisitor {
  private static final Map<String, DataType> TYPES_MYSQL_2_FRAMEWORK;
  static {
    TYPES_MYSQL_2_FRAMEWORK = new HashMap<String, DataType>();
    TYPES_MYSQL_2_FRAMEWORK.put("BIT", DataType.INTEGER);
    // More data type mapping follow here
  }
  private List<Entity> entities = new ArrayList<Entity>();
  private Entity currentEntity;
  private EntityAttribute currentEntityAttribute;
  \\ visit methods follow here
```

Code Snippet 63: The `DDLVisitor`

The next step is to implement the `visit` methods for each node type. This is shown in Code Snippet 64. The first node type of interest in the AST is `DDLTable`. A create table statement can be directly mapped to a WebML entity, so a new `Entity` is created. The constructor takes the entity name as an argument which is the name of the table. The `currentEntity` reference is set to the newly created instance and the duration of the `Entity` is set to persistent. Finally the entity is added to the `entities` list.

The next level in the node hierarchy are the `DDLColumnDeclaration` nodes. They can be directly mapped to a WebML `EntityAttribute`, so a new instance of `EntityAtribute` is created and the column name is used as the name of the attribute. The `currentEntityAttribute` reference is set and the attribute is added to the `attributes` list of the current entity.

The `visit` method for `DDLDataType` nodes is used to set the WebML data type of the current `EntityAttribute`. The correct WebML type for the MySQL datatype is retrieved from the map defined earlier.

Finally the `EntityAttributes` that are part of the primary key have to be marked. Therefore the `visit` method for `DDLPrimaryKeyConstraint` nodes is used. It is iterated over all the column names that are part of the primary key and the names are compared with the names of the `EntityAttributes` created. If the name is the same, the `key` property of the `EntityAttribute` is set to `true`.

Setting the Relationships

After traversing the AST the list of `Entity` objects is fully initialized and can be retrieved from the visitor by calling `getEntities`. A new `DataModel` instance is created and the `Entities` are set in the `compile` method. What is missing now is to set the relationships between the Entities. Therefore the user provided `relationships.xml` has to be unmarshalled into a Java object representation. This is shown in Code Snippet 65.

An empty list of `Relationship` objects is created and it is iterated over all `Reference` objects, as shown in Code Snippet 66. Inside the loop for each `Reference` a corresponding

```
public Object visit(DDLTable node, Object data) {
  currentEntity = new Entity(node.getName());
  currentEntity.setDuration(Duration.PERSISTENT);
  entities.add(currentEntity);
  return null;
}
public Object visit(DDLColumnDeclaration node, Object data) {
  currentEntityAttribute = new EntityAttribute(node.getColumnName());
  currentEntity.getAttributes().add(currentEntityAttribute);
  return null;
}
public Object visit(DDLPrimaryKeyConstraint node, Object data) {
  for(String key : node.getKeys()){
    for(EntityAttribute a : currentEntity.getAttributes()){
      if(a.getName().equalsIgnoreCase(key)){a.setKey(true);}
    }}
  return null;
}
public Object visit(DDLDataType node, Object data) {
  currentEntityAttribute.setType(TYPES_MYSQL_2_FRAMEWORK.get(
    node.getDataType()));
  return null;
}
```

Code Snippet 64: The `DDLVisitor`

```
DataModel dataModel = new DataModel();
dataModel.setEntities(visitor.getEntities());
JAXBContext context = JAXBContext.newInstance(References.class);
Unmarshaller u = context.createUnmarshaller();
References references = (References) u.unmarshall(
  new FileReader("webapp/relationships.xml"));
```

Code Snippet 65: Setting the entities and unmarshalling the `relationships.xml` file

Relationship is created. The getEntityByName method is a convenience method of the DataModel to find an entity by its name. The Relationship is initialized with the source and the target entity in the constructor.

```
List<Relationship> relationships = new ArrayList<Relationship>();
for(Reference reference : references.getReferences()){
  Entity source = dataModel.getEntityByName(reference.getSourceTable());
  Entity target = dataModel.getEntityByName(reference.getTargetTable());
  Relationship relationship = new Relationship(source,target);
  relationship.setName(source.getName() + "_" + target.getName());
  \\...
```

Code Snippet 66: Creating the relationships between the entities

A Relationship has two RelationshipRoles that store the cardinality for each end of the Relationship, as shown in Code Snippet 67. relationShipRole1 represents the end of the source entity, relationShipRole2 represents the end of the target entity.

```
  // Creating the relationship role 1 (rr1)
  RelationshipRole rr1 = new RelationshipRole();
  if(reference.getSourceToTargetCardinality().equals("1")){
    rr1.setMaxCard(Cardinality.ONE);
  } else {
    rr1.setMaxCard(Cardinality.MANY);
  }
  rr1.setName(source.getName() + "2" + target.getName());
  // the same for relationship role 2 (rr2)
  relationship.setRelationShipRole1(rr1);
  relationship.setRelationShipRole2(rr2);
```

Code Snippet 67: Creating the relationships between the entities

Finally the targetJoinColumn is set on the Relationship, which represents the foreign key in the target Entity, as shown in Code Snippet 68. This is not a part of the DataModel, but it is needed later in the reverse engineering process. Therefore this property is annotated with @XmlTransient. The getEntityAttributeByName of the Entity class is a convenience method to find an EntityAttribute by its name.

8.2 Implementing the PHP to WebML Compiler

After the DDLCompiler has finished its work the next step is to create the Web Model. This is done in a class called PHPCompiler. In this Section the implementation of this class and other classes used by PHPCompiler is explained. The PHPCompiler has a static compile Method that takes an FS_Directory object containing the file system tree of the web application and a DataModel object that has been reverse engineered before as arguments. In the compilation process a WebApp object is created and manipulated several times until it finally contains the data necessary to create the target WebML model. The compile method performs the following steps:

```
  //...
  EntityAttribute ea = target.getEntityAttributeByName(
    reference.getTargetJoinColumn());
  if(ea != null){relationship.setTargetJoinColumn(ea);}
  relationships.add(relationship);
}
dataModel.setRelationships(relationships);
return dataModel;
```

Code Snippet 68: Creating the relationships between the entities

1. Step one is the reverse engineering of the OperationUnits, which are encapsulated inside OperationModules.This is done by the `createOperationModules` method, which is responsible for preparing the sources, for creating the intermediate data structure and for doing the transformation of the model code to WebML operation modules. The method takes the `FS_Directory` object and the `DataModel` passed to the `compile` method as arguments and returns a `WebApp` object.

2. Step two is the creation of the Pages and their nested ContentUnits. This is done by the `createPages` method, which transforms the `Template` objects created before into a WebML site view with Pages. The method takes a `WebApp` object as an argument.

3. Step three is the creation of the Links between the Pages and the ModuleInstanceUnits. This is done by the `createLinksFromPages` method. The method takes a `WebApp` object as an argument.

4. Step four is the creation of the Links between the ModuleInstanceUnits and the Pages or other ModuleInstanceUnits respectively. This is done by the `createLinksFromOperationModules` method. The method takes a `WebApp` object as an argument.

5. The final step is to put the data collected by the `WebApp` object into a WebML `WebModel` object which is returned by the `compile` method.

8.2.1 Building the AST

The first step in the process is to get an AST of the PHP source code. Therefore a JavaCC grammar file is used which can be found in the grammar repository on the JavaCC website [41]. The grammar file does not create an AST so it has to be transformed into a JJTree file first. This file is quite long as it covers almost all of the PHP language elements, so only the parts relevant for the reverse engineering process are discussed in this thesis.

The parser class `PHPParser` has a static `buildAST` method that takes an InputStream as an argument an returns a `PHPStart` Object, which is the source of the AST. The source code is shown in Code Snippet 69. The `DEFAULT` lexical state is used for parsing PHP code but there is also a state for parsing HTML called `HTML_STATE`. As templates usually start with HTML code, it is initially switched to the `HTML_STATE` by calling `parser.token_source.SwitchTo(HTML_STATE);`.

```
public class PHPParser {
  private static PHPParser parser;
  public static PHPStart buildAST(InputStream in)
    throws ParseException {
    parser = new PHPParser(in);
    parser.token_source.SwitchTo(HTML_STATE);
    return parser.PhpPage();
  }
}
```

Code Snippet 69: The `PHPParser` class

The `PhpPage` production rule represents a PHP script and starts with zero or many `HtmlBlocks`. In the `HtmlBlock` one of three possible choices is expected:

- A token of type `<HTML>`, which is defined as `<HTML: ( ["<"] | "<" ["?"])+ >`
- A token of type `<HTML_OTHER>`, which is defined as `<HTML_OTHER: "<" [] >`
- An `Expression` enclosed by `<PHP_EXPR>` and `<PHP_END>`. `<PHP_EXPR>` is defined as `<PHP_EXPR: "<?=" > : DEFAULT` and causes a switch to the `DEFAULT` state. `<PHP_END>` is defined as
  `<PHP_END: "?>" > : HTML_STATE` and causes a switch back to the `HTML` state.

The String parsed as `<HTML>` or `<HTML_OTHER>` represent an arbitrary sequence of text or HTML code and is stored in the `PHPHtmlBlock` object as it is needed later.

The `HtmlBlock()` production rule is followed by a `<PHP_BEGIN>` token which is defined as `<PHP_BEGIN: "<?" ("php")?> : DEFAULT` and which causes a switch to the `DEFAULT` state and zero or many `Statement()` production rules.

```
PHPStart PhpPage() #Start : {}{
  (HtmlBlock())* (
  <PHP_BEGIN> (Statement())*
  | <EOF> ) { return jjtThis; }
}
void HtmlBlock() #HtmlBlock :
{Token t;}{
  t=<HTML> {jjtThis.setHtml(t.image);}
  | t=<HTML_OTHER> {jjtThis.setHtml(t.image);}
  | <PHP_EXPR> Expression() <PHP_END>
}
```

Code Snippet 70: The `PhpPage` and the `HtmlBlock` production rule

As the grammar file is quite long the rest of the production rules is not shown here. A table with the production rules relevant to the reverse engineering process, a short description of what language elements are matched by the rule and the node type created is given in table 8.1.

Production rule	Node type	Description
PhpPage	Start	The whole PHP script
HtmlBlock	HtmlBlock	HTML code that comes before the first PHP statement
EmbeddedHtml	EmbeddedHtml	All HTML code blocks after the first PHP statement
Statement	Statement	Any type of PHP statement
ForeachStatement	Foreach	A foreach statement followed by a ForeachArgument and a ForeachBody
ForeachArgument	ForeachArgument	The head of the foreach statement
ForeachBody	ForeachArgument	The body of a foreach statement
Variable	Variable	A PHP variable
ArrayAccess	ArrayAccess	A child node of Variable that matches the an expression inside [] characters
StringLiteral	StringLiteral	Any Text inside "" or '' characters
EchoStatement	Echo	An echo or print statement followed by a list of expressions
CompoundStatementBegin	CompoundStatementBegin	The { character
CompoundStatementEnd	CompoundStatementEnd	The } character
MemberFunction	MemberFunction	Matches a member function of a class
AssignementOperator	AssignementOperator	Matches the = symbol. The first child of the node has to be the Variable node to which a value is assigned. The second child can be any expression.
IfStatement	If	Matches an if statement followed by an IfArgument argument and a IfBody. The else part is optionally matched by ElseBody
IfArgument	IfArgument	Matches the expression in the head of the if statement.
IfBody	IfBody	Matches the statements in the body of an if block
ElseBody	ElseBody	Matches the statements in the body of an else block
ExpressionStatement	ExpressionStatement	Matches a statement other that a variable assignment such as a call to a function
ReturnStatement	Return	Matches the return keyword followed by an arbitrary statement.
MemberAccess	MemberAccess	Matches a call to either a member function or a member variable of an object.

Table 8.1: Important production rules of the PHPParser

8.2.2 Transforming the Templates to XML

To transform the templates into the XML representation presented in Subsection 6.2.1 several steps are necessary that are performed by the `createOperationModules` method:

1. Creating pairs of model classes and the corresponding templates.
2. Removing all HTML elements that are not important for the transformation.
3. Replacing the interesting HTML elements by the corresponding XML elements.
4. Transforming the PHP statements into XML elements.

Creating Pairs of Model Classes and Templates

A new class called `ModelViewFilePair` is introduced which has a `templateFile` and a `modelFile` property, both of type `FS_File`. This class represents a file containing a model class and a file containing the corresponding template file. Those files are extracted from the `FS_Directory` object representing the files system tree of the whole web application. It is searched for files that are placed in a module directory which is a child directory of `modules`. For each `<model_class_name>.php` model file in this directory the subdirectory `phptpl` is searched for a file with the same and the ending `tpl.php` and a `ModelViewFilePair` object is created and initialized with the two files. All `ModelViewFilePair` objects are stored in a list.

Removing unnecessary HTML Elements

The next step is to iterate over all the `ModelViewFilePair` objects and transform the content of the template files to XML. This is done using the Jericho HTML parser. A Jericho `Source` object is created with the content of each template file and a corresponding `OutputDocument` object is created for each `Source` object. A call to `PHPTagTypes.register()` makes sure that the PHP tags are recognized by the parser. Then it is iterated over all the `Segments` of the `Source` object. The starting and closing tag of the `html` element is replaced with the `template` tag using the `replace` method of the `OutputDocument`. All other HTML tags are removed except `title`, `a`, `value`, `form`, `input` and `PHPTagTypes.PHP_STANDARD`.

Replacing the HTML Elements with XML

Now it is necessary to bring the `input` tag and the `a` tag into the correct format. The `input` element in `HTML` has the format `<input name="..." type="..." value="..."/>`. The goal is to extract the content of the `type` attribute and to transform it into the format shown in Code Snippet 71, if the content of the value attribute is a String literal or into the format shown in Code Snippet 72, if the content of the value attribute is a PHP statement. This is done using the `findAllElements` method on the `Source` object to find all elements of type `Tag.INPUT`. Then it is iterated over all the `input` elements. The new XML element is created for each `input` element using a `StringBuffer` and is replaced in the `OutputDocument`. The `anchor` element is create out of the `a` element in the same manner.

```
<input name="..." type="..." >
  <literal>...</literal>
<input>
```

Code Snippet 71: The `input` element with a nested `literal` element

```
<input name="..." type="..." >
  <?php ... ?>
<input>
```

Code Snippet 72: The `input` element with a nested PHP statement

Transforming the PHP Statements to XML

The last step in the transformation process of the templates is to convert the PHP tags to XML. The `PHPParser` presented in Section 8.2 is used to build an AST of the `OutputDocument` created before. The visitor class `TemplateVisitor` is created and used when traversing the AST. The `visit` Method is implemented for the following node types: `Foreach`, `HtmlBlock`, `EmbeddedHtml`, `Echo` and `CompoundStatementEnd`. When the template visitor is created it is initialized with a `StringBuffer` used to build the target XML document.

The first production rule to be matched is `HTMLBlock`. The `HTMLBlock` node type has an `html` property that contains the whole block of HTML code matched in one String. The fact that the code matched here is XML and not HTML does not matter because any text that comes before the first PHP statement is matched in this rule. In the `visit` method for `HTMLBlock` the String value of the `html` property is appended to the `StringBuffer` that is still empty so far.

Like the `HTMLBlock` production rule the `EmbeddedHTML` production rule matches any text outside a block of PHP code. The only difference is that `HTMLBlock` matches the text before the first block of PHP code and `EmbeddedHTML` matches any subsequent block of text. In the `visit` method for `EmbeddedHTML` the String value of the `html` property is appended to the `StringBuffer` as it is.

In the `visit` method for the `Foreach` node the `Variable` node that comes on the left side of the `as` keyword and the `Variable` node that comes on the right side of the `as` keyword are extracted. The `Variable` node has a `name` property that contains the name of the variable. A String that has the format `<iterator variable="..." as="...">` is created and appended to the `StringBuffer`. The `variable="..."` part is filled with the name of the variable that is on the left side of the `as` keyword and the `as="..."` is filled with the name of the variable that is on the right side of the `as` keyword.

The `EchoStatement` production rule matches any occurrence of an `echo` or `print` statement. In the `visit` method for the `Echo` node the first thing to happen is that the String `<echo ` is appended to the `StringBuffer`. The `Echo` node has a child node of type `Variable`. This node is retrieved and the string `variable="`, followed by the variable name followed by the string `">` is appended to the `StringBuffer`. The `Variable` node can have a child node of type `ArrayAccess` that matches an expression enclosed by square brackets characters appended to a variable.

The ArrayAccess node can have a child of type StringLiteral that matches the index of the array. The StringLiteral node has a value property that contains the string value that has been matched. This value is retrieved and appended to the StringBuffer enclosed by the strings <array index=" and "/>.

A foreach statement ends with a } character, which is matched by the the PHPCompoundStatementEnd production rule. In the visit method a closing </iterator> tag is appended to the StringBuffer. This concludes the implementation of the TemplateVisitors visit methods.

Now that the XML document has been built it is unmarshalled to a Java object structure using JAXB and added to the list of Template objects of the WebApp.

8.2.3 Transforming the Model Classes

After a template has successfully been transformed into a Java object structure the compiler program is still in the loop that iterates over all the ModelViewFilePair objects. The next task is to transform the model class that belongs to the template into the intermediate data structure. It starts with the creation of a new ModelClass object. The object is initialized with the DataModel and added to the Template. The Template and the ModelClass are both initialized with an id which has the structure /<module_name>/<class_name>. The transformation into the intermediate data structure is done in several visitor classes. Again, the AST of the model class is built first using the PHPParser.

Transforming the Member Functions

The top level visitor class is called MemberFunctionVisitor. It has a single visit method for MemberFunction nodes. The traverseNodes method is called with the AST of the model class, the MemberFunctionVisitor object and the DataModel. In the visit method a new Function object is created and added to the Function list of the Model object. An OperationModule and a ModuleInstanceUnit is created. The OperationModule contains the OperationUnits and the ModuleInstanceUnit represents an instance of the OperationModule, which is placed on the site view, so a reference from the ModuleInstanceUnit to the OperationModule is set. The OperationModule is also initialized with an InputCollectorUnit and an OKCollectorUnit.

Visiting the Statements of a Function

To create the OperationUnits it is necessary to analyze the statements inside a function. This is done by the StatementVisitor. The subtree that follows under a Function node is traversed with this visitor together with the Function object.

The entry point to the StatementVisitor is the visit method for the Statement node. The current Function object is set as a member variable here.

Many of the statements in the web application are assignment operations where a variable is initialized with the value of another variable an array or the result of a call to a function. Those assignments are handled in the visit method for the AssignementOperator node. The left side of an assignment is the variable to which a value is assigned. So the first child of the AssignementOperator node is fetched and casted to a Variable node object. As mentioned before the Function object maintains a map of Variable objects. It is checked whether the variable already exists in this map and if not it is added under its name. The value assigned to the variable depends on the right side of the assignment statement so this subtree has to be analyzed. For this purpose a new visitor class called AssignmentRightSideVisitor is created. The subtree is traversed using this visitor. The functionality of the AssignmentRightSideVisitor is discussed later in this Subsection.

An if statement is matched by the IfStatement production rule but the interesting part is the expression in the head of the if block, so the visit method is implemented for the IfArgument node type. In the patterns presented in Subsection 6.2.2 an if statement always checks if the value of a variable containing the result of a database query is set. This is mapped to an IsNotNullUnit. The Variable node, which is the first child of the IfArgument node is fetched and the corresponding Variable object from the current Function is retrieved. The value of the Variable is casted to QueryValue, a new IsNotNullUnit is created and the stored as an instance variable of the StatementVisitor as it is needed again later. According to the pattern presented in Subsection 6.2.2 the SelectorUnit that has been reverse engineered from the variable value has an outgoing OKLink that points to the IsNotNullUnit. So a new OKLink is created and added to the SelectorUnit object of the QueryValue. The to property of the Link is set to the IsNotNullUnit and the IsNotNullUnit is added to the OperationUnit property of the Function object. How the QueryValue object is created and initialized is described later in this Subsection.

The else part of an if statement is handled in the visit method for the ElseBody node. The only thing that happens here is that a flag called isInElse is set to true, to indicate that the current statement is inside an else block.

The next type of statements that can be mapped to the patterns presented in Subsection 6.2.2 is the call to the set function of a model class, where a variable is passed to the template. This kind of statement is handled in the visit method for the ExpressionStatement node type. The statement has the format
$this->set('variable_name',$variable);. The first child of the ExpressionStatement node is casted to a Variable node that represents the this variable. The interesting parts of the statement are the arguments passed to the template. The first argument is a string that is the variable name under which the variable passed as the second argument is available in the template. The Function object is enhanced with a second map that keeps track of the variables used in the template. To handle the two arguments a new visitor class called SetTemplateVariableVisitor is created. The Variable node representing the this variable, the SetTemplateVariableVisitor and the current Function object is passed to the traverseNodes method. The implementation of the SetTemplateVariableVisitor is described in later in this Subsection.

The last node type of interest is Return, which represents a variable or string literal value after a return keyword. It is used to trigger a redirect to the path that is returned. The first child of the Return node is retrieved and casted to a StringLiteral node. The value of the node is the path of the module to which the redirect is made. The Function class is enhanced with a returnLinks property which is a list of String objects to store this path and the path is added to this list.

Visiting the set Functions for Template Variables

The SetTemplateVariableVisitor is used to handle the arguments passed to the set method of the model class. The first argument is represented by a StringLiteral node. The visit method for this node gets its value and stores it in a member variable of the visitor. In the visit method for the Variable node the corresponding Variable object is retrieved from the Function object. As the variables presented in the templates contain query results they have to be of type QueryValue so the value is casted to a QueryValue object. According to the patterns presented in Subsection 6.2.2 this represents an OKLink from the SelectorUnit that has been reverse engineer from this query to the OkCollectorUnit. Thus an OKLink is created and added to the SelectorUnit object of the QueryValue. The to property is set to the OKCollectorUnit of the Function. Furthermore the key attributes of the Entity that is queried with this SelectorUnit has to be coupled with LinkParameters. The Entity object has a keys property which is a list of EntityAttribute objects that are part of the primary key. Those keys are retrieved from the entity property of the Query object and for each key EntityAttribute a new LinkParameter is created and added to the OKLink. Now the LinkParameter has to be coupled an OutputCollectorParameter of the OKCollectorUnit so a new OutputCollectorParameter is created and added to the OKCollectorUnit. To couple the two parameters the target attribute of the LinkParameter is set to the id of the OutputCollectorParameter. The name of both parameters is set to the name of the EntityAttribute. Finally the Variable object created before is added to the map of template variables under the name that has been extracted before in the visit method of the StringLiteral node.

Visiting the right Side of Assignment Statements

In the example application the right side of a variable assignment is either a call to the prepare function of the db object, a call to the execute function of the db object, an assignment of a request parameter or an assignment of any other variable. All these cases have to be handled properly by the AssignmentRightSideVisitor. The visitor has two visit methods, one for StringLiteral nodes and one for Variable nodes. The constructor of the visitor takes a Function object, a IsNotNullUnit object and a boolean value that indicates whether the current statement is inside an else block or not as arguments. Those values are set as member variables of the visitor. The visitor also has a value property of type IValue which is initialized with the correct value that after evaluating the right side of the assignment. The visit method for the StringLiteral nodes is responsible to transform the string value into an IValue object. In the example application all string literals are SQL statements so they are transformed to one of the types that implement SQLStatementValue. How this transformation works is described at the end of this Subsection. The visit method for the Variable node type has to decide what to do with the variable.

The first and simplest possibility is that the value of the variable is directly assigned to the variable on the left side of the assignment operation. This is the case if the Variable node does not have any children. The value of the variable is retrieved from the Variable map of the Function object and assigned to the value property of the visitor.

The second possibility is that the name of the Variable node is either _GET or _POST. In this case the variable represents a request parameter and it is interesting to know which parameter is accessed here. This information can be found by inspecting the array index that is accessed on the _GET or _POST variable. A new RequestParameterValue object and a new InputCollectorParameter is created. The name of the InputCollectorParameter is set to the value of the StringLiteral used as array index and the InputCollectorParameter object is added to the RequestParameterValue object. Finally the InputCollectorParameter object is added to the InputCollectorUnit of the Function object and the RequestParameterValue object is set to the value property of the visitor.

The third possibility is that the the variable represents a query result stored in an array where the column names are used as indexes. In this case the Variable object represented by the Variable node is retrieved form the Function and its value is casted to QueryValue. The EntityAttribute that has the same name as the index used for accessing the array is retrieved, a new QueryResultColumnValue is created an initialized with the attribute. Finally the QueryResultColumnValue object is set to the value property of the visitor.

The fourth possibility is that the expression on the right side is a call to one of the functions of the $this->db object. This case is handled by a separate visitor called MethodCallVisitor. The constructor of the visitor takes the Variable, the current IsNotNullUnit object and the boolean flag indicating whether the current statement is inside an else block as arguments. The current Variable node is traversed with the MethodCallVisitor and the current Function object. The implementation of the MethodCallVisitor is described later in this Subsection.

Visiting Function Calls for Database Access

The MethodCallVisitor has a single visit Method for nodes of type MemberAccess. This node type represents a call to a member function or to a field of an object an it has a name property to get the function or variable name. The function calls used in the example application are either on the db object of a model class or on a query result object. Calls on the db object are either to query, prepare or execute. A call on a query result object is always to fetchRow.

The first and simplest possibility is a call to fetchRow of a query result object. In this case the result object is already of type QueryResult so the type of the variable passed to the MethodCallVisitor is simply returned.

The second possibility is a call to the query function of the db object. This case is handled by a separate visitor called DBQueryVisitor. The current MemberAccess node is traversed with a new instance of the DBQueryVisitor and the current Function object. The visitor implements the visit methods for the StringLiteral node type and for the Variable node type. The visit method for StringLiteral handles the case that an SQL query string literal is directly passed as an argument to the query function. As before the query string has to be transformed into an SQLStatementValue object. The visitor has a value property of type IValue which is initialized with the SQLStatementValue. At this point the SelectorUnit object is created, using the data collected in the QueryValue object, and added to the OperationModule of the current Function. The visit method for the Variable node does almost the same except that the value is retrieved directly from the Variable object that maps to the Variable node which is passed.

The third possibility is a call to the prepare function of the db object. This case is handled by a separate visitor called DBPrepareVisitor. The current MemberAccess node is traversed with a new instance of the DBPrepareVisitor and the current Function object. The visitor implements the visit methods for the StringLiteral node type and for the Variable node type. The two methods are implemented in the same way as in the DBQueryVisitor.

The fourth possibility is a call the execute function of the db object. This case is pretty complex and therefore describe separately later in this Subsection.

Visiting the execute Function

The analysis of the execute function is crucial to the creation of the patterns described in Subsection 6.2.2. The transformation to WebML is implemented using two visitors: The DBExecuteVisitor and the DBExecuteQueryArgumentsVisitor. The execute function has two arguments that have to be handled separately. The first argument is a statement object representing a prepared statement. The second argument is an array containing the values to be set for the prepared statement. The values are set in the order as they occur in the array.

From the MethodCallVisitor the current MemberAccess node is traversed with a new instance of the DBExecuteVisitor and the current Function object. The first argument is a node of type Variable. The coresponding variable object is retrieved from the Function object and its value is stored in the value property of the visitor. The DBExecuteVisitor has a single visit method for the ArgumentList node type which is a child of the MemberAccess node type and which represents a list of arguments.

The next step is the analysis of the variables passed in the array as the second argument of the execute function. This is handled by the DBExecuteQueryArgumentsVisitor. A new instance of this visitor is created and initialized with the IValue object of the variable passed as the first argument of the execute method and the current Function is also passed to the visitor via the constructor. The visitor has a property called currentOperationUnit which is of the abstract type OperationUnit. In the constructor the actual type of the SQLStatementValue object is checked, the corresponding OperationUnit is created and set as the value of the currentOperationUnit property. If the SQLStatementValue is an instance of QueryValue a new SelectorUnit is created, if the SQLStatementValue is an instance of InsertValue a new CreateUnit is created and if the value is an instance of UpdateValue a new ModifyUnit is created. In the case of a QueryValue the corresponding SelectorUnit is additionally stored in a global selectorUnit property. This is necessary to implement pattern 5 presented in Subsection 6.2.2 where a transport Link is set from the SelectorUnit to the ModifyUnit. Furthermore a Link of type transport is created and added to the outgoing Links of the InputCollectorUnit that belongs to the current Function object. The to property of the Link is set to the currentOperationUnit and the Link is also stored in a global field called currentLinkFromICU (ICU stands for InputCollectorUnit). The reason for this is that the parameters for the prepared statement might be set by values that come from request parameters. If so the corresponding LinkParameter objects have to be created and set in the visit method for the Variable nodes.

The DBExecuteQueryArgumentsVisitor has a single visit method for nodes of type Variable. The method is entered for each variable in the array passed as the second argument for the execute function. It is important to keep track of the position of the variable in the array in order to map it correctly to the columns set in the query. Therefor a global counter variable called variableIndex of type int is used that is initialized with -1 and incremented each time the visit method is called. The next step is to get the value of the variable from the Variable map of the Function object.

Now there are two possibilities what the value of the variable could be. If it is a request parameter is has to be an instance of RequestParameterValue. Otherwise it is a the value column of another database query, e.g. a foreign key queried before or a value that is copied from a row column of a certain table to a row column of another table.

The first case to be considered is when the value is an instance of RequestParameterValue. A new LinkParameter object is created and the source is set to the InputCollectorParameter stored in the RequestParameterValue object. The parameter is added to the LinkParameter list of the currentLinkFromICU property. Now that the source of the LinkParameter is set the target has to be set as well. This depends on the type of the currentOperationUnit property to which the corresponding currentLinkFromICU property points to. The are three possibilities:

1. If the current OperationUnit is a SelectorUnit the target of the LinkParameter has to be set to the correct Condition of the SelectorUnit. The conditions are stored in a list of the SelectorUnit and the order in the list corresponds to the order of the variables so the correct condition is retrieved by getting the list element with the index value of the variableIndex field. Finally the target property of the LinkParameter is set to the id of the Condition.

2. If the current OperationUnit is a CreateUnit the target of the LinkParameter has to be set to the corresponding id of the EntityAttribute. The EntityAttribute the element from the attributes list of the CreateUnit with the current index of the variableIndex field.

3. If the current OperationUnit is a ModifyUnit the situation is a bit more complex as the SQL update statement has a SET part and a WHERE part which are both set dynamically in the prepared statement. The SET part comes first and is mapped to the entity attributes to be updated. The WHERE part that follows is mapped to the selector conditions of the ModifyUnit. To solve this problem another global counter of type int called numUpdateSetArgs is introduced and initialized with 0 when the visitor is created. To find out if the current Variable node represents a value for the SET part or for WHERE part of the statement, it is checked if the variableIndex is lower than the size of the attribute list of the ModifyUnit. If so, the variable represents an argument for the SET part of the query. The EntityAttribute with the index of the current variableIndex value is retrieved from the attributes list and the id of the EntityAttribute is set as the target of the LinkParameter. The

numUpdateSetArgs counter is incremented by one. If the condition evaluates to false it means that the LinkParameter has to be coupled to a Condition of the Selector. This is done in almost the same way as for the conditions of the SelectorUnit. The difference is that the index for the Condition to be retrieved from ModifyUnit is not variableIndex but variableIndex - numUpdateSetArgs.

The other case is that the value is an instance of QueryResultColumnValue. This part is only relevant for the processOrder function in the summary class, which corresponds to pattern 6 presented in Subsection 6.2.2. The goal is to create a Link from the SelectorUnit to which the value belongs to the currentOperationUnit. To determine if the OKLink has already been created a new instance field called linkFromSEUtoCurrentOperationUnit of type Link is defined. If the field is null a new OKLink is created and added to the Link list of the SelectorUnit. The to property is set to the currentOperationUnit. The parameter coupling is not implemented for this case. If the linkFromSEUtoCurrentOperationUnit property has been set already, nothing happens here. This concludes the implementation of the DBExecuteQueryArgumentsVisitor visitor.

The execution of the compiler program continues now in the DBExecuteVisitor. What is left to do is to set a Link between the SelectorUnit and the ModifyUnit, to set the OKLink between the IsNotNullUnit and the ModifyUnit and to set the KOLink between the IsNotNullUnit and the CreateUnit. The SelectorUnit object and the object of the current OperationUnit is retrieved from the DBExecuteQueryArgumentsVisitor. If the current OperationUnit is an instance of ModifyUnit a new Link is created and added to the SelectorUnit's Link list. The target of the is set to the ModifyUnit. Furthermore it is checked if the currentIsNotNull Unit is set. If so it is checked if the is isInElse flag is set to false in this case a new OKLink is created that points and added to the IsNotNullUnit. It the flag is set to false, the same is done with a KOLink instead. This concludes the description of the visitor implementations. After the AST has been traversed by all the vistors, the execution of the program returns to the createOperationModules method of the PHPCompiler class which returns the WebApp object that is now filled with the information extracted by the visitors to the compile method.

Transforming the SQL Statements to SQLStatementValues

The string literals in the source code that contain SQL statements have to be transformed into SQL-StatementValue objects. This is done in a utility class with a static getLiteralValue method. This method takes the String with the SQL statement and the DataModel as arguments. The string is parsed using a tool called ZQLParser [17]. A new instance of ZQLParser is created and initialized with the SQL statement String. The readStatement method of the ZQLParser object returns an Object of type ZStatement which is an abstract class. The implementing types are ZQuery representing a select statement, ZInsert representing an insert statement, ZInsert representing an insert statement and ZDelete representing a delete statement.

In case that the ZStatement object is an instance of ZQuery the object is casted and the select part is retrieved via the getSelect method, which returns a Vector of ZSelectItem objects. The from part of the query is retrieved by calling getFrom which returns a Vector. The first element of this `Vector` is a `ZFromItem` object the contains the table name. The `Entity` object that belongs to this name is retrieved from the `DataModel`. A new `QueryValue` object is created and the entity is set on the object. The next step is to iterate over all the elements in the `Vector` with the selected column names. The `QueryValue` object has a `selectedColumns` property which is a list of type `EntityAttribute`. The `EntityAttribute` for each selected column is retrieved and added to the `selectedColumns` list. Finally the where part of the query is retrieved by calling `getWhere`. The method returns an object of type `ZExpression`. The where part of the query is mapped to a selector so a new `Selector` object is create and set as the `selector` property of the `QueryValue` object. Now the `ZExpression` object has to be evaluated to create the correct `Condition` objects for the `Selector`. This is done in a separate method called `evalWhereExpr`. First it is checked whether the `ZExpression` is an simple expression such as `a = 1` or if it is a compound expression, such as `a = 1 AND b = 2`. In the latter case it is checked if the boolean operator between the two sub expressions is `AND` or if it is `OR` and the `booleanAttribute` property of the `Selector` object is set accordingly. Then the `evalWhereExpr` is recursively called again to evaluate the sub expressions. In the first case the expression is further analyzed to decide to what kind of `Condition` it maps to. The left side of the comparison expression is an object of type `ZConstant` and represents the column whose value is checked. The corresponding `EntityAttribute` is retrieved from the `QueryValue` object. The next step is to check what kind of `Condition` the expression represents based on the the characteristics of this `EntityAttribute`. The first possibility checked is if the expression represents a relationship role condition. Therefore it is iterated over all `Relationship` objects of the `Entity`. The name of the `EntityAttribute` object that is referenced by the `targetJoinColumn` property of each `Relationship` is compared to the name of the `EntityAttribute` in the expression. If they are the same it is obvious that the column used in the expression is the foreign key of another table. In this case a new `RelationshipRole` object is created and added to the list of `Condition` objects of the `Selector`. The second possibility is that the column in the expression is a primary key. This is true if the `key` attribute of the related `Entity` is set to true. In this case a new `KeyCondition` is created and added to the list of `Condition` objects of the `Selector`. The third possibility is that the condition to be created is neither a `RelationshipCondition` nor a `KeyCondition`. In this case an `AttributeCondition` is created and added to the list of `Condition` objects of the `Selector`.

In case that the `ZStatement` object is an instance of `ZInsert` a new `InsertValue` object is created and initialized with the `Entity` that matches the table name in the insert statement. It is iterated over all the columns that are listed in the `VALUES` part of the statement. The corresponding `EntityAttribute` for each column name is retrieved from the `Entity` and added to the list of `EntityAttribute` objects referenced by the `attributes` property of the `InsertValue` object.

In case that the `ZStatement` object is an instance of `ZUpdate` a new `UpdateValue` object is created and initialized with the `Entity` which matches the table that is updated. It is iterated over all the columns that are updated in the statement. The corresponding `EntityAttribute` for each column name is retrieved from the `Entity` and added to the list of `EntityAttribute` objects referenced by the `attributes` property of the `UpdateValue` object. As an update statement also has a where part the `UpdateValue` object has a `selector` property. The selector is created in the same way as for the QueryValue objects by calling the evalWhereExpr method.

8.2.4 Creating the Pages

After the the execution of the createOperationModules method the intermediate data structure is built completely and the statements inside the model functions has been transformed to WebML operation modules. The next step in the reverse engineering process is to build the Pages and the ContentUnits placed on the Pages. This is done in the createPages method that takes the WebApp object as an argument.

Inside the method it is iterated over all the templates in the WebApp object. In the loop a new Page is created and its name property is filled with the title of the Template. Inside this loop the code can be split into two sections. The purpose of the first section is to transform the patterns that are based on forms. The purpose of the second section is to transform the patterns that are based on anchors.

Transforming form-based Patterns

In this section of the code it is iterated over all the Form objects of the template. Inside the loop it is first of all checked if the Form has an Iterator object. If so the pattern at hand is the IndexUnit. The Variable object that belongs to the name stored in the variable property of the Iterator is retrieved. The variable mentioned in the as property of the Iterator has not been initialized yet. According to the patterns presented in Subsection 6.2.1 the variable over which it is iterated in the template contains the result of a database query and has to be of type QueryValue because of that. The iteration variable to which the as part references is simply set to the same value. An IndexUnit object is created and the entity property set to the Entity object of the QueryValue. For the creation of the Links, which is done later in the program, the Form and the Iterator needs to have a reference back to the IndexUnit so it is added to the entityContentUnit property of the Iterator and to the list of ContentUnits of the Form. In the next step a Selector has to be added to the IndexUnit. A default selector is created for this purpose which is used for all IndexUnits, DataUnits and MultidataUnits. The reason for this is that the correct selection already happens in the operation modules and that the object id of the object to be displayed in the ContentUnit is passed as a coupled parameter on a Link from the ModuleInstanceUnit to the ContentUnit. The default Selector is created and initialized with a KeyCondition. After that it has to be checked which attributes should be displayed. This is done by iterating over the Echo elements of the Iterator. The Variable referred to in the variable property is retrieved from the template variable list and casted to QueryValue. Then the EntityAttribute that maps to the index property of the array property of the Echo object is retrieved from the QueryValue and added to the displayAttributes list of the IndexUnit. Finally the IndexUnit is added to the Page

If the size of the list of Echo objects is greater than 0 the pattern that follows is the DataUnit. First of all a new DataUnit is created and added to the contentUnits list of the Form as well as to Page. As with the IndexUnit the default selector is set as selector for the DataUnit. The displayAttributes of the DataUnit are set in almost the same way as the display attributes of the IndexUnit. The only difference is that this time it is iterated over Echo elements that are children of the Form instead of the Iterator.

The last form-based pattern to be handled is the EntryUnit, which follows if the Form has children that are Input Elements whose type property is set to text. In this case a new EntryUnit is created and it is iterated over all the the Input elements of the form. For each Input with the type set to text a new Field is created. The name is set to the name of the Input and the Field is added to the fields list of the EntryUnit. Finally the EntryUnit is added to the contentUnit lists of the Page and the Form.

Transforming anchor-based Patterns

In this section of the code it is iterated over all Iterator elements of the Template. An Iterator outside a Form could be either an IndexUnit or a MultidataUnit. The decision what type of Unit to create is based on the fact if the Iterator contains Anchor elements or not. If the Iterator contains Anchor elements a new instance of IndexUnit is created otherwise a MultiDataUnit is created. The Unit just created is added to the Iterator because it is needed later in the program for the creation of the Links. The next step is to retrieve the iteration variable in the same way as describe for the IndexUnit in form based patterns, described before. The Entity for the ContentUnit is retrieved from the QueryValue of the iteration variable. The display attributes are set in the same way as with the form-based patterns described before and again the default selector is set for the selector property.

8.2.5 Creating the Links from the Pages

After returning from the createOperationUnits method the next step is to create the Links from the Pages to the ModuleInstanceUnits. This is done in the createLinksFromPages method that takes the WebApp object as an argument. Again it is iterated over all the Template objects and the code inside the loop can be split into two section: One for extracting form-based patterns and one for extracting anchor-based patterns.

Transforming form-based Patterns

In this code section it iterated over all the Form objects in the template. The action attribute of the Form is also used as the id of the Function objects, so the correct Function that handles the request triggered by the current Form is retrieved from the WebApp object. A form usually has a submit button which is mapped to an outgoing Link of the form. Therefore it is iterated over all the Input elements of the Form and searched for an Input with the type property set to submit. Now it has to be decided whether this submit input element denotes a Link that belongs directly to the Page or one or more Links that belong to a ContentUnit of the Page. In the createPages method the ContentUnit objects created for each Form have been added to the contentUnits list of the Form. If the size of this list is greater than 0 it indicates that the submit input belongs to the the ContentUnits stored in the list.

In the simpler case that the submit input belongs to the Page a new Link is created and its name is initialized with the literal value inside the value property. The to property is set to the ModuleInstanceUnit whose id is the same as the action property value of the Form. Finally the Link is added to the Page that belongs to the current Template.

In the other case it is iterated over all the ContentUnit objects of the Form. Inside the loop a Link is created and initialized in the same way as a Link that belongs to a Page but this time the Link is added to the ContentUnit. The next step is to create the LinkParameter objects for the Link and to couple them correctly. How this is done depends on the type of the ContentUnit. There are three possibilities:

1. If the current Form has an Iterator child element the current ContentUnit must be an IndexUnit. The interesting part here is the value of the radio button used in the form which is supposed to be an echo statement that prints the primary key of the database object to be selected. First of all the Variable used in the echo statement has to be retrieved. The name is found by walking down the properties of the subtree under the Iterator object until input.value.echo.variable. The QueryValue of the corresponding Variable is retrieved from the list of template variables. The correct EntityAttribute is found in the QueryValue object by its name which is the value of the input.value.echo.variable.array.index property of the Iterator. A new LinkParameter is created and added to the linkParameter list of the Link. The source of the LinkParameter is set to the id of the EntityAttribute. What is left to do is to couple the target of the LinkParameter to the correct InputCollectorParameter of the InputCollectorUnit of the target ModuleInstanceUnit. Therefore the Function with the same id as the value of the action property of the Form is retrieved from the WebApp object. The InputCollectorParameter is retrieved from the Function and it is iterated over its InputCollectorParameter elements. The target of the LinkParameter is set to the id of the InputCollectorParameter whose name is the same as the name attribute of the Input element.

2. If the Form has children of type Input the current ContentUnit must be a DataUnit. The DataUnit can be used to submit attributes of the object that it represents to a ModuleInstanceUnit via LinkParameters. This pattern is applied for the productDetails template and for the summary template. The values submitted with the Form are found in the Input elements of type hidden. The variable used in the echo statement that prints the value of the each hidden Input is retrieved and its value is casted to a QueryValue object. The EntityAttribute that is referenced by the array.index property of the Echo element is retrieved from the Entity object of the QueryValue. The name of such an EntityAttribute is compared to the name of each EntityAttribute of the DataUnits Entity. If they are the same the correct mapping for the LinkParameters is found. A new LinkParameter is created and added to the Link. Its name is set to the name of the Input. The source is of the LinkParameter is set to the id of the EntityAttribute of the DataUnit. The target of the LinkParameter is found in the same way as described for the IndexUnit.

3. If the ContentUnit does not have an entity property it must be an EntryUnit. In this case a LinkParameter is created for each Input element of type text and added

to the Link. The source of the LinkParameter is set to the id of the Field of the EntryUnit with the same name as the name of the Input element. The target of the LinkParameter is found in the same way as described for the IndexUnit.

Transforming anchor-based Patterns

What is left to do in in this section of the code is to create the Links for anchor based patterns. In this pattern the Template element has an Iterator element as a direct child. A new Link is created and the name is set to the value of the anchor.text attribute. The to attribute of the Link is set to the ModuleInstanceUnit with the same id as the value of the anchor.href attribute and the Link is added to the Link list of the ContentUnit stored in the Iterator object.

Now the LinkParameters have to be coupled correctly. This is done by analyzing the query string of the URL path. Therefore it is iterated over the Argument objects of the Anchor. For each Argument the Variable object that belongs to its value.echo.variable value is retrieved and casted to a QueryValue object. The creation and the mapping of the corresponding LinkParameter works exactly the same way as described for the IndexUnit described before. This concludes the implementation of the createLinksFromPages method.

8.2.6 Creating the Links from the Operation Modules

The final step in the transformation process for the Web Model is the creation of the Links between the ModuleInstanceUnits and the Pages. This step is done inside the createLinksFromOperationModules method. The WebApp object is passed as an argument.

In the first step it is iterated over all the Template objects of the WebApp object. In a nested loop it is iterated over all the Function objects of the ModelClass that belongs to the Template. It is checked whether the Function has any returnLinks. Now there are two possibilities:

1. If the Function has any return Links it means that the outgoing Link of the corresponding ModuleInstanceUnit points to another ModuleInstance Unit. In this case the ModuleInstanceUnit with the same id as the the value of the return Link is retrieved from the WebApp. A new OKLink is created, the target is set to this ModuleInstanceUnit and the Link is added to the ModuleInstanceUnit of the current Function.

2. If the Function does not have any return Links it means that the outgoing Link of the corresponding ModuleInstanceUnit points to the Page related to this ModuleInstanceUnit. A new OKLink is created, the to property is set accordingly and the Link is added to the ModuleInstanceUnit of the current Function. The target of the Link has been set to point to the Page by default now, but it is also possible that the Link points to a ContentUnit placed in the Page and that LinkParameters are passed to this ContentUnit. To find this out it is iterated over the OutputCollectorParameters of the OKCollectorUnit that belongs to the current OperationUnit. For each OutputCollectorParameter a LinkParameter is created and the source is set to the

`id` of the `OutputCollectorParameter`. In a nested loop it is iterated over all `ContentUnits` of the target `Page`. For each `ContentUnit` it is checked whether it has an `entity` property. If so the `Entity` is retrieved and in another nested loop it is iterated over its `EntityAttributes`. Now it is checked whether the current `EntityAttribute` is the same instance as the `EntityAttribute` related to the current `OutputCollectorParameter`. If so the `target` of the `Link` is set to the current `ContentUnit`. Finally it has to be taken care that the `LinkParameter` points to the `KeyCondition` of the target `ContentUnit`. This is done in another loop over all the `Conditions` of the `Selector` of the target `ContentUnit`.

This concludes the implementation of the `createLinksFromOperationModules` method.

8.2.7 Creating and Marshalling the WebProject

Now the transformation process is almost finished. What is left to do is to create a `WebModel` object out of the data collected in the `WebApp` object, to put it into a `WebProject` object together with the `DataModel` and to marshall it to XML. Back in the `compile` method a new `WebModel`, a new `SiteView` and a new `ModuleView` are created. The `SiteView` and the `ModuleView` are added to the `WebModel` and the `SiteView` is also set as the `homeSiteView`. Then it is iterated over all the `Template` objects stored in the `WebApp` and the `Page` that belongs to each `Template` is added to the `Pages` list of the `SiteView`. In a nested loop it is iterated over all the `Function` objects of the `ModelClass` that belongs to the current `Template`. The `OperationModule` of each `Function` is added to the `ModuleView` and the corresponding `ModuleInstanceUnit` is added to the `SiteView`. Now the work of the `PHPCompiler` class is done and the `WebModel` object is returned to the `main` method of the `PHP2WebML` class. All that is left to do is to create a new `WebProject`, to add the `DataModel` and the `WebModel` and to marshall it to XML using JAXB.

The result can be viewed in WebRatio.

Chapter 9

Related Work

In this Chapter an overview of some related work is given. Furthermore the reverse engineering process presented in this thesis is compared to another approach called WARE.

9.1 Overview

In Patel et al. [22] a summary of recent works concerning the reverse engineering of web applications by different authors is given. Some important points are:

- Many reverse engineering methodologies for web applications are based on web application development methodologies, e.g. UML based.
- S. Tilley [51] described different reverse engineering objectives including pattern abstraction, redocumentation and architecture recovery.
- Most reverse engineering activities concentrate on static analysis of source data rather then on the dynamic analysis of the operation of web applications.
- Schwabe et al. [27] discovered three areas of concern when undertaking web application forward engineering, which also apply to reverse engineering. Those are application behavior, navigation modeling and interface design.
- Two popular forms of representations for web applications are tabular and graphical representations. Ricca and Tonella [26] proposed a graph model where a web application is represented by a directed graph `W=(P,E)`, where `P` is a set of HTML pages and `E` a set of links connecting members of `P`.
- Reverse engineers of web applications must follow a disciplined structure and should use dedicated reverse engineering methodologies that provide a consistent and complete plan, making it possible to determine which process need to be carried out together with associated tools required for supporting the method. For simple/small web applications manual analysis is sufficient. For larger applications a computer aided analysis will be required.

- The majority of the reported web application reverse engineering methodologies and tools are founded on the Unified Modeling Language (UML) such as the WARE (Web Application Reverse Engineering) tool developed by Di Lucca [10], which adopts the UML extensions for web application modeling proposed by Conallen [8]. UML based method provide a stable, familiar environment for reengineers to work with for modeling components as well as the behavior of an application. Chung and Lee [7] also adopted the Conallen extension and extracted component diagrams (each WWW Page is a component) and package diagrams(reflecting the web applications directory structure).

Although WebML is a good choice for modeling web applications, as it is intuitive and expressive, it is worth taking brief a look at other web modeling languages, which is done in Section 9.2 of this Chapter. A tool of special interest is WARE, as it utilizes some approaches similar to those proposed in this thesis. An overview of the WARE tool and a comparison to the reverse engineering process presented in this thesis is given in Section 9.3 of this Chapter.

9.2 Web Modeling Languages

Beside WebML there are some other web modeling languages and approaches available. In a survey on web modeling approaches for ubiquitous web applications by Schwinger et. al. [28] a good overview is given:

- The Hera Design Methodology is based on the Resource Description Framework (Schema) - RDF(S). The content level is modeled using a proprietary graphical notation. The domain model is based on concepts, attributes, content relationships and media types. The application model and the presentation model resembles the hypermedia design method Relationship Management Methodology (RMM). The application model is mainly based on slices and slice relationships.

- The Web Site Design Method (WSDM) is one of the earliest web modeling approaches and uses the Web Ontology Language (OWL). The structural modeling of all web application levels is supported.

- The UML-based Web Engineering Approach (UWE) supports UML models for content, hypertext and presentation levels. Structural modeling is based on class diagrams using stereotypes for specific web concepts. Navigation at hypertext level is modeled using state chart diagrams. At presentation level sequence diagrams can be used for modeling presentation flows. OCL is used to describe the interface between content and hypertext levels.

- The Object-Oriented Hypermedia Method (OO-H) is based on UML and uses different models for the content, hypertext and presentation level. At content level, UML class diagrams are used for modeling content and activity diagrams are used for modeling processes. At hypertext level, so called navigation access diagrams (NAD) are used for modeling the navigation paths a user can activate. At navigation level, a default abstract presentation diagram (APD) can be generated out of the NAD.

9.3 The WARE tool

The Web Application Reverse Engineering (WARE) [9] tool follows Bendusi's general reverse engineering paradigm Goals, Tools and Models [1]. Tools aim to support the recovering process, goals focuses on reverse engineering motivations and models deals with the definition of the information to extract. The reverse engineering process must recover the static architecture, the dynamic interactions and the behavior of a web application. Thus the reverse engineering process consists of the following phases:

1. Static Analysis
2. Dynamic Analysis
3. Behavioral Analysis

The aforementioned phases recover views that can be represented by extended UML diagrams. The architecture of a web application is represented by class diagrams, the dynamic model is represented by sequence and collaboration diagrams and the behavior is represented by use case diagrams. With the results of the analysis phases a conceptual model is created. The whole process is tool supported but also requires manual actions.

9.3.1 The Analysis Phases

Static Analysis

In the static analysis phase the web application's architecture components and the static relations amongst them are recovered. This includes HTML files, the directory structure, scripting language sources, database connections, the use of applets/servlets or any other static information. HTML Pages and the relevant sub elements (e.g. forms, script blocks, database components) are mapped into classes while links are mapped into relations.

Dynamic Analysis

The dynamic analysis phase rely on the static analysis results. The web application is executed and dynamic interactions among the components in the class diagram are recorded. Any event or action is traced to the source code and to the classes in the class diagram. Events are the visualization of a page, a form submission, a link traversal, a database query, processing of data etc. the sequence of actions fired by an event deriving form the web application's code control flow or from user actions are associated to sequences of messages exchanged between the object of the web application. These sequences can be represented by sequence diagrams or collaboration diagrams.

Behavioral Analysis

The behavioral or functional analysis phase aims to detect the behavior of a web application from a user point of view. The discovered behavior is described by use case diagrams.

9.3.2 The Conceptual Model

A web application's conceptual model has to specify abstractions representing the application, its components and the relations between components. The following taxonomy concerning pages is considered:

- *Server pages* that reside on the server as opposed to *client pages* which are actually sent to the browser.
- *Static pages* whose content is fixed as opposed to *dynamic pages* whose content varies.
- *Simple pages* as opposed to *framed pages*, which consist of a frameset that includes several pages.
- *Unlinking pages* as opposed to *linking pages* that contain hypertextual links.

9.3.3 Tool Support

The architecture of the supporting tool consists of the following elements:

- The *Extractor* parses HTML, client-side and server-side scripting languages and produces an intermediate datastructure.
- The *Intermediate Form Repository* stores an intermediate form obtained from the extracted information.
- The *Abstractor* executes abstraction operations on the intermediate form and recovers UML diagrams.

9.4 A Comparison to WARE

The WARE approach applies a combination of static code analysis and dynamic program behavior analysis. The static analysis phase is done completely automatically, wheres the dynamic analysis phase requires user interaction. An intermediate datastructure repository which is subsequently enhanced is used for storing the results of the different analysis phases. Different models at different levels of granularity are created based on the data stored in the repository. The approach seems to work with arbitrary web applications that are not implemented using a certain framework.

The process presented in this thesis applies static code analysis only. The static code analysis phase is done completely automatically but it requires that the source web application is written for a certain framework. Before that the web application has to be manually rewritten to conform to this framework. As with ware, certain intermediate data structures are used to support the automatic transformation process. As opposed to ware, those data structure are not stored persistently, but only exist in-memory during the execution of the transformation program. The result of the transformation process is a model that resembles the actual implementation of the web application. Other than in WARE it is not possible to create models at different levels of granularity.

Chapter 10

Conclusion and Future Work

In this Chapter the result of the reverse engineering process is presented. Furthermore the result is evaluated and some options for future work are outlined.

10.1 Summary

The transformation program presented in Chapter 8 has been tested with the refactored source application from Chapter 4. The result is an XML document representing the WebML model recovered from the input sources. It can be viewed and modified with WebRatio. The graphical representation of the model is shown in Figure 10.1, Figure 10.2 and Figure 10.3.

Figure 10.1 and Figure 10.2 show the OperationModules that that have been recovered from the model files of the MVC application. Each OperationModule represents one model class function. As the purpose of a model function is to perform certain operations using the request parameters coming from the view, an InputCollectorUnit is created for each operation module. At the end of each model function certain parameters are usually passed either directly to the corresponding view template or to another model function by the means of a redirect operation. Both cases are represented by a OKCollectorUnit, which is created by default. All parts of the code where database queries are executed were recovered as SelectorUnits, CreateUnits or ModifyUnits (there was no delete operation in the example application). If/else statements were mapped to IsNotNullUnits. The Links amongst the OperationUnits and the coupled parameters were recovered by tracing the variable assignments in the source code.

Figure 10.3 shows the Hypertext Model that has been recovered from the view templates. Each template is mapped to a Page. The components of the view templates were mapped to the corresponding WebML elements such as forms, IndexUnits, DataUnits and Links. An important fact is that the recovered Hypertext Model reflects the actual MVC implementation and not a conceptual model that shows the functionality from a user point of view. For example there are no direct Links between Pages but always Links from Pages or Page elements to operation ModuleInstanceUnits and from OperationModuleInstanceUnits to Pages. This reflects the MVC implementation where all requests are sent to the controller first, which in turn decides, which model function is responsible for handling the request.

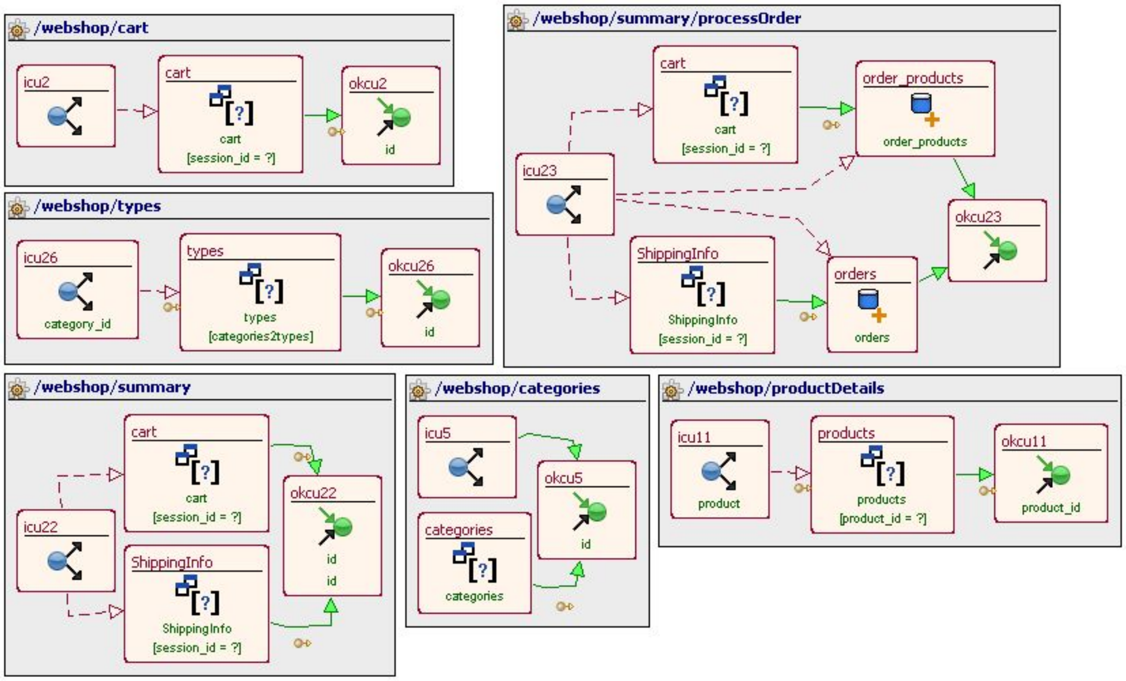


Figure 10.1: Operation modules recovered from the reverse engineering process (1/2)

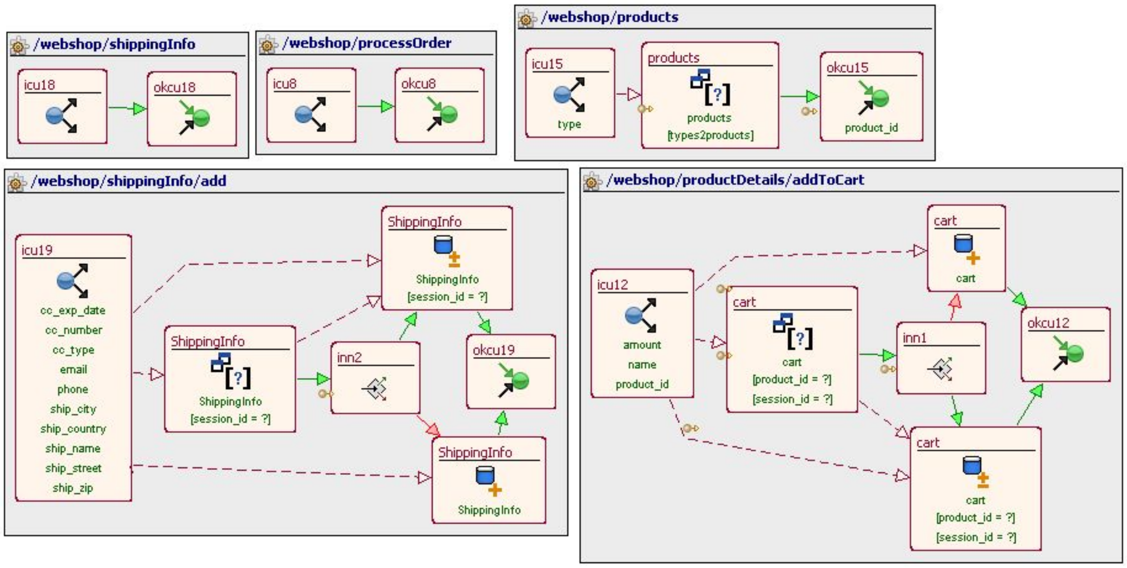


Figure 10.2: Operation modules recovered from the reverse engineering process (2/2)

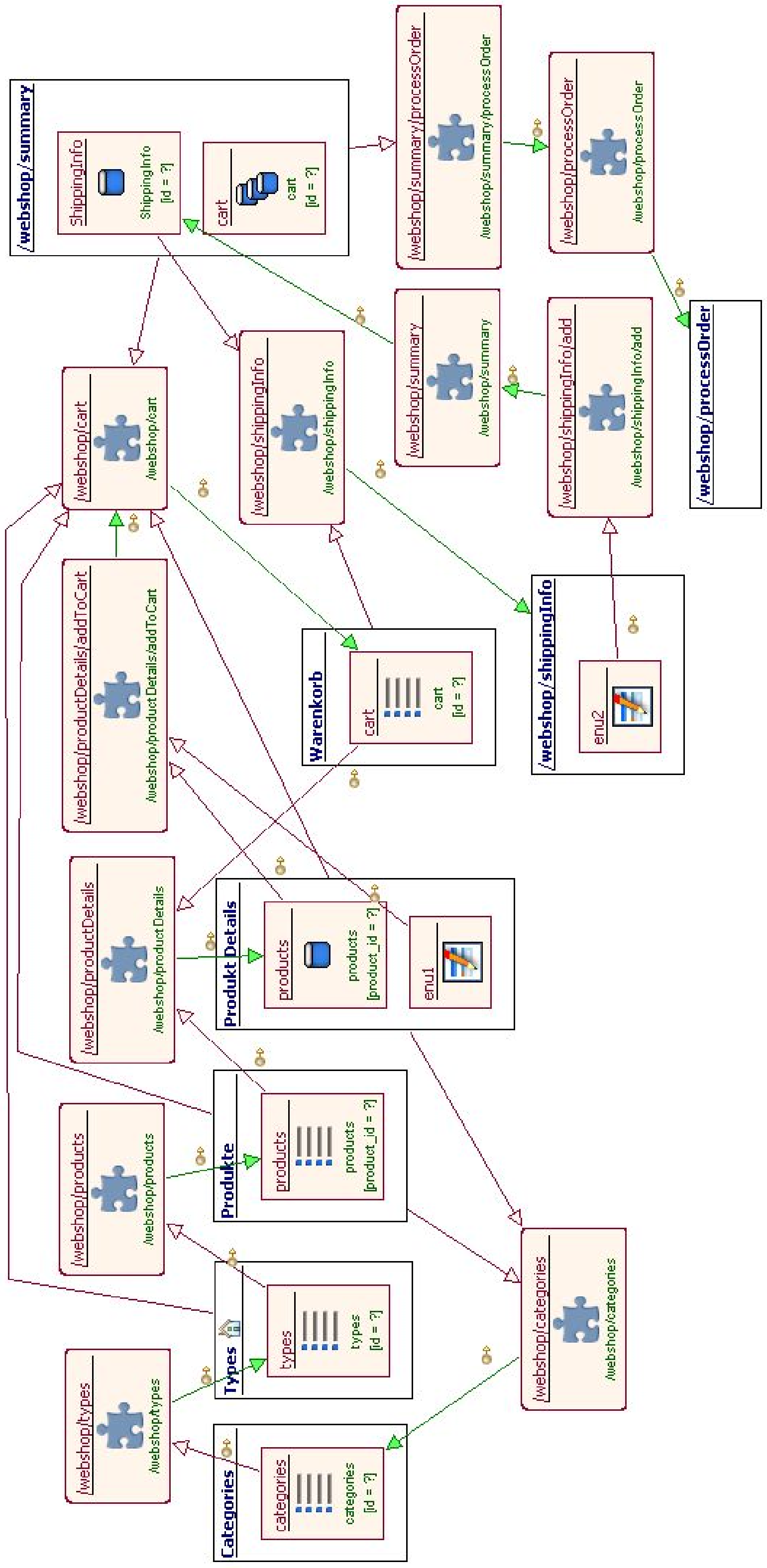


Figure 10.3: The hypertext model recovered from the reverse engineering process

10.2 Evaluation

The analysis of the source code has been rather difficult as there are hardly any tools available for processing PHP code. Also the combination of different grammars is not trivial. The solution presented in this thesis to define an intermediated data structure that is partly based on XML to Java object binding worked quite well for this purpose.

WebML is a language on a very high level of abstraction which makes it hard to define a direct mapping between the source code artifacts and WebML model elements. The use of an intermediate data structure was very helpful to close the gap between the two worlds.

10.2.1 Prerequisite for an Automatic Transformation

It was necessary to make some assumptions about the web application to find suitable mappings between source code patterns and WebML patterns.

1. The database has to be normalized in order to reasonably map tables and relationships to a WebML data model.

2. SQL create table statements can be directly mapped to entities. The relationships between the entities can either be inferred from the foreign key constraints or this information has to provided by the user.

3. The web application has to follow the MVC pattern. This is done for three reasons:

   (a) The MVC pattern is the most widely adopted pattern for PHP web applications and also for web applications written in languages other than PHP.

   (b) Much of the ”glue code” used for request handling and dispatching is predefined by the MVC framework used and does not have to be considered in the source code analysis.

   (c) It dramatically reduces the amount of possibilities for patterns used in the implementation of the web application.

4. Each request is handled in a single function of a model class. This is true for the MVC framework and also for the symfony framework and it can be mapped quite well to the concept of an operation module in WebML.

5. The input parameters used in the InputCollectorUnit can be mapped straightforward from access operations to request parameters in the source code. This works for the MVC framework and for symfony.

6. For database access operations it has been assumed that certain objects and functions are used. In the MVC framework the database access works by executing SQL query strings or prepared statements and in symfony database operations are done using an ORM layer. Both variants can be directly mapped to OperationUnits.

7. All the if/else blocks used in the model functions are checking whether a query has returned a result or not. This can be mapped quite well to the IsNotNullUnit in WebML.

8. All the variables used in the model functions either contain SQL query statements or values of request parameters.

9. The Links and the Link Parameters set between the OperationUnits are inferred from the variable values and types used for database queries or as template variables.

10. The variables passed to a template are mapped to output parameters of an OKCollectorUnit.

11. The view templates do not contain any business logic. The only PHP statements that can be found in a template are echo statements that print a variable value or foreach statements to echo all the values of an array.

The example application does fulfill all these assumptions so it can almost be perfectly mapped to a WebML model.

10.2.2 Shortcomings of the Assumptions

In real world applications the above mentioned assumptions are often not fulfilled. There are some shortcomings that have to be considered:

- Obviously not all databases in use for web applications are normalized.

- The data of a web application might not only be taken from a relational database but maybe from other datasources such as web services or semantic web data sources. Such kinds of data sources are not considered by WebML.

- There are countless different MVC frameworks for PHP available. They all follow the similar basic principles but still there are significant differences regarding the design and the libraries used.

- Although a request triggers a call to a model function that is able to access the request parameters and other information, it is not necessarily true that all of the business logic is also handled in this function. No one can prevent a developer from placing business logic into another method in the same or in another class. In symfony the function that handles a request is actually considered to be part of the controller and the preferred programing style is to put business logic into the model classes used with Propel.

- WebML is designed to model typical web related patterns such as passing parameters between Pages and OperationUnits and some basic data manipulation patterns. Although this forms a big part of many web application it usually does not cover all the functionality. All parameters passed amongst OperationUnits or between Pages and OperationUnits are related to attributes of the Data Model. There is no concept for modeling values that are calculated in the business logic of a web application or that come from data sources other than the database. The assumption that all variable values used in a web application represent database related information is hardly ever true in a real wold application.

- An if/else block does not necessarily have to check whether an SQL query has returned a result or not. If it fulfills another purpose this cannot be recognized by the reverse engineering program.
- Although it is recommended not to put any business logic into the view, no one can prevent the developer from doing so. Such cases are not recognized by the transformation program.

10.2.3 Information Loss in Reverse Engineering

Not all the information contained in the source application is recovered during the reverse engineering process. The following pieces of information are lost:

- The layout and the structure of the template files are not mapped to the target WebML model.
- The original database layout is lost, due to the normalization of the database.
- The paginating functionality of the products page is not recovered.

10.3 Future Work

10.3.1 Improving the Analysis of the PHP Code

The first prerequisite to improve the reverse engineering process would be to have a better processing tool for PHP code. The parser used in this thesis is based on a simple grammar file for JavaCC. Initially the parser only checked whether a given PHP file conformed to the grammar or not. By using JJTree it was possible to build an AST but still it was rather complex to process the AST. A lot of visitor classes where necessary to search for certain patterns in the AST. Here are some suggestions for the requirements to a more useful PHP processing tool.

- The tool should not only be able to parse single PHP script files and to check their syntactical correctness but it should be able to take a whole PHP application as input and to resolve references between classes as well as inheritance hierarchies. This would include the evaluation of several PHP language features.
  - Statements to make PHP scripts available in other scripts such as `require`, `require_once`, `include` or `include_once`.
  - Class inheritance.
  - References between classes.
- The tool should know all the built-in PHP functions (e.g. `mysqli_query`, `exit`, etc.).
- There should be a high level API to access the PHP source code artifacts and elements such as all classes or scripts of the application, all functions of a class and all statements of a function. The statements could be further classified into assignment statements, function calls, etc. There should be accessors to easily get the arguments passed to a function and it should be possible to resolve instance variables such as `this`.

A promising tool that might help to fulfill those requirements is the open source PHP compiler (PHC) [23].

10.3.2 Using intermediate Models or Data Structures

As mentioned before WebML is a very high level modeling language that is not intended to create a model that reflects the implementation details of a web application but rather gives a top down view on the structure and behavior of the application modeled. To reasonably improve the reverse engineering process from the source code level to a WebML model it would be suitable to do several transformation iterations into well define intermediate models. With each iteration the target model should further prescind from the source code. In this thesis only one intermediate data structure has been used. Probably it would be helpful to define the following levels of abstraction:

1. The source code, consisting of PHP, HTML and SQL code.

2. The AST of the source code.

3. A platform specific model for the concepts of the framework used, such as symfony or the MVC framework used in this thesis. This model should cover platform specific aspects such as model classes, templates, frontend controller, ORM mappings, etc.

4. A platform independent model covering concepts common for all MVC frameworks. This model should not consider details such as if the database access is done via ORM or via plain SQL queries or if the framework uses frontend controllers beside the backend controller or not.

5. The target WebML model.

Different approaches to standardize such kinds of models are done by the Architecture Driven Modernization Task Force [55].

10.3.3 Introducing a Refactoring Phase

The resulting model is very close to the actual implementation but does not quite look like a model that would have been created when doing forward engineering activities. Thus it would be suitable to introduce an additional phase where the implementation model resulting from phase 2 is refactored into a conceptual model. Some parts could possibly be refactored automatically, while others parts might require refactoring by a human.

Some patterns that are good candidates for automatic refactoring are:

- De facto empty Operation Modules (i.e. OperationUnits that only consist of an InputCollectorUnit and an OKCollectorUnit) could be removed, as they represent empty functions in the MVC implementation. From a user perspective this is a simple Link from one Page to another Page, that does not transport any information and it could therefore be replaced with a direct Link from one Page to the other Page.

- The pattern combination in Figure 10.4 shows how the selection made in one IndexUnit serves as the input for the selection criteria of another InputUnit. The selection is done by a SelectorUnit inside an OperationModule. This pattern combination could be simplified as shown in Figure 10.5, where a Link is directly pointing from one IndexUnit to the other.
- The current implementation of the compilation program creates one Operation Module for each model function and does not consider Operation Modules with duplicated content.

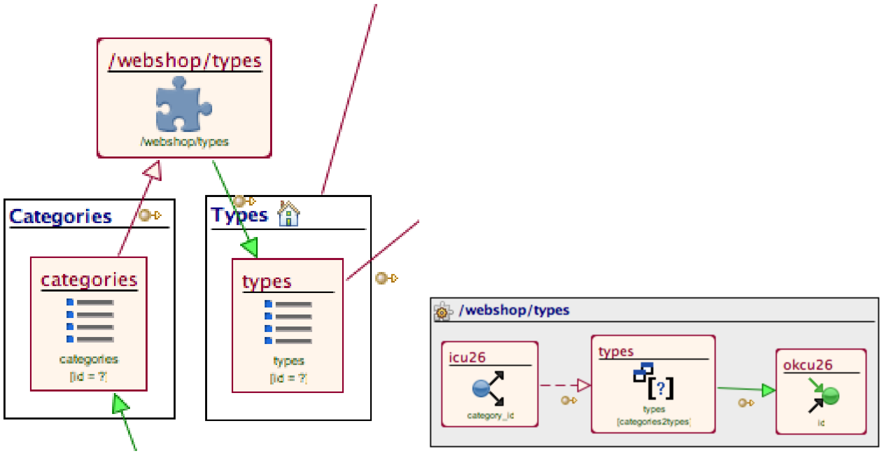


Figure 10.4: Pattern combination produced by the reverse engineering tool

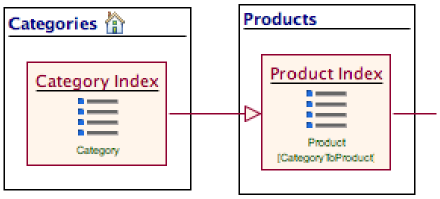


Figure 10.5: Refactored pattern

Appendix A

The Reverse Engineering Framework and Examples

The CD-Rom attached to this thesis contains the following elements:

- The reverse engineering framework.
- The example web application.
- The refactored MVC versions of the example application.
- The WebML model resulting from the reverse engineering process in the Web Ratio XML format.

Bibliography

[1] Benedusi, P. and Cimitile, A. and De Carlini, U. Reverse engineering processes, design document production, and structure charts. *J. Syst. Softw.*, 19(3):225–245, 1992.

[2] Berners-Lee, T. and Masinter, L. and McCahill, M. *RFC 1738 - Uniform Resource Locators (URL)*. The Internet Engineering Task Force, December 1994.
http://tools.ietf.org/html/rfc1738, accessed on June 9th, 2009.

[3] Brownell, D. *SAX2*. O'Reilly, 2002.

[4] Byrne, S. and Le Hors, A. and Le Hégaret, P. and Champion, M. and Nicol, G. and Robie, J. and Wood, L. Document Object Model (DOM) Level 3 Core Specification. W3C Recommendation, W3C, April 2004. http://www.w3.org/TR/2004/REC-DOM-Level-3-Core-20040407, accessed on June 9th, 2009.

[5] Ceri, S. and Fraternali, P. and Bongio, A. and Brambilla, M. and Comai, S. and Matera, M. *Designing Data-Intensive Web Applications*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2002.

[6] Chen, P. The entity-relationship model—toward a unified view of data. *ACM Trans. Database Syst.*, 1(1):9–36, 1976.

[7] Chung, S. and Lee, Y. Reverse Software Engineering with UML for Web Site Maintenance. In *Proceedings of the First International Conference on Web Information Systems Engineering (WISE'00)-Volume 2*, page 2157, Washington, DC, USA, 2000. IEEE Computer Society.

[8] Conallen, J. *Building Web Applications with Uml*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[9] Di Lucca, G. A. and Casazza, G. and Di Penta, M. and Antoniol, G. An Approach for Reverse Engineering of Web-Based Applications. In *Proceedings of the Eighth Working Conference on Reverse Engineering (WCRE'01)*, page 231, Washington, DC, USA, 2001. IEEE Computer Society.

[10] Di Lucca, G. A. and Fasolino, A. R. and Tramontana, P. Reverse Engineering Web Applications: The WARE approach. *Journal of of Software Maintenance and Evolution, Research and Practice*, 16:71 – 101, 2004.

[11] ECMA International. *ECMAScript Language Specification*, December 1999. http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-262.pdf, accessed on June 9th, 2009.

[12] Enseling, O. Build your own languages with JavaCC. *Java World*, December 2000. https://javacc.dev.java.net/, accessed on June 9th, 2009.

[13] Fielding, R. and Irvine, UC. and Gettys, J. and Mogul, J. and Frystyk, H. and Masinter, L. and Berners-Lee, T. *RFC 2616 - Hypertext Transfer Protocol – HTTP/1.1*. The Internet Engineering Task Force, June 1999. http://tools.ietf.org/html/rfc2616, accessed on June 9th, 2009.

[14] Gamma, E. and Helm, R. and Johnson, R. *Design Patterns. Elements of Reusable Object-Oriented Software.* Addison-Wesley Longman, Amsterdam, 1995.

[15] Garshol, L. BNF and EBNF: What are they and how do they work?, June 2009. http://www.garshol.priv.no/download/text/bnf.html, accessed on June 9th, 2009.

[16] Geeknet, Inc. *Jericho HTML Parser*, June 2009. http://jericho.htmlparser.net/docs/index.html, accessed on June 9th, 2009.

[17] Gibello, P. *Zql: a Java SQL parser*. gibello.com, June 2009. http://www.gibello.com/code/zql/, accessed on June 9th, 2009.

[18] Hansen, K. Working with files and directories in Java. *The Java Boutique*, June 2009. http://javaboutique.internet.com/tutorials/Files_Directories/, accessed on June 9th, 2009.

[19] Kappel, G. and Pröll, B. and Reich, S. and Retschitzegger, W. *Web Engineering. Systematische Entwicklung von Webanwendungen*. Dpunkt Verlag, 2003.

[20] Magoulas, R. Programming Language Trends. *O'Reilly Radar*, Aug 2006. http://radar.oreilly.com/archives/2006/08/programming-language-trends-1.html, accessed on May 21st, 2009.

[21] Maurer, F. and Martel, S. Extreme programming. Rapid development for Web-based applications. *Internet Computing, IEEE*, 6(1):86 – 90, January/February 2002.

[22] Patel, R. and Coenen, F. and Martin, R. and Archer, L. Reverse Engineering of Web Applications: A Technical Review. Technical report, University of Liverpool, July 2007. http://www.csc.liv.ac.uk/research/techreports/tr2007/ulcs-07-017.pdf, accessed on June 9th, 2009.

[23] phpcompiler.org. *PHC Manual*, June 2009. http://www.phpcompiler.org/doc/latest/manual.html, accessed on June 9th, 2009.

[24] Politecnico di Milano. *The Web Modeling Language*, June 2009. http://www.webml.org/webml/page1.do, accessed on June 9th, 2009.

[25] Potencier, F. and Zaninotto, F. *The Definitive Guide to symfony*. Apress, Berkeley, California, USA, May 2009.

[26] Ricca, F. and Tonella, P. Web Site Analysis: Structure and Evolution. In *Proceedings of the International Conference on Software Maintenance (ICSM'00)*, page 76, Washington, DC, USA, 2000. IEEE Computer Society.

[27] Schwabe, D. and Esmeraldo, L. and Rossi, G. and Lyardet, F. Engineering Web Applications for Reuse. *IEEE MultiMedia*, 8(1):20–31, 2001.

[28] Schwinger, W. and Retschitzegger, W. and Schauerhuber, A. and Kappel, G. and Wimmer, M. and Pröll, Birgit and Castro Cachero, C. and Casteleyn, S. and De Troyer, O. and Fraternali, P. and Garrigos, I. and Garzotto, F. and Ginige, A. and Houben, G. and Koch, N. and Moreno, N. and Pastor, O. and Paolini, P. and Ferragud Pelechano, V. and Rossi, G. and Schwabe, D. and Tisi, M. and Vallecillo, A. and van der Sluijs, K. and Zhang, G. A survey on web modeling approaches for ubiquitous web applications. *International Journal of Web Information Systems (IJWIS)*, 4(3):234–305, 2008.

[29] Sensio Labs. *symfony - Open-Source PHP Web Framework*, June 2009.
http://www.symfony-project.org/, accessed on June 9th, 2009.

[30] Sensio Labs. *The symfony Cookbook*, June 2009.
http://www.symfony-project.org/cookbook/1_2/en/, accessed on June 9th, 2009.

[31] Smarty. *Smarty - the compiling PHP template engine*, September 2007.
http://www.smarty.net/manual/en/, accessed on June 9th, 2009.

[32] Stump, J. Understanding MVC in PHP. *O'Reilly Media*, October 2005.
http://oreilly.com/php/archive/mvc-intro.html?page=1, accessed on June 9th, 2009.

[33] Sun Microsystems, Inc. *The JavaTM Architecture for XML Binding (JAXB) 2.0*, April 2006.
http://jcp.org/aboutJava/communityprocess/final/jsr222/index.html, accessed on June 9th, 2009.

[34] Sun Microsystems, Inc. *JavaCC [tm]: Documentation Index*, June 2009.
https://javacc.dev.java.net/doc/docindex.html, accessed on June 9th, 2009.

[35] Sun Microsystems, Inc. *JavaCC [tm]: JJTree Reference Documentation*, June 2009.
https://javacc.dev.java.net/doc/JJTree.html, accessed on June 9th, 2009.

[36] Sun Microsystems, Inc. *JavaCC: TokenManager MiniTutorial*, June 2009.
https://javacc.dev.java.net/doc/tokenmanager.html, accessed on June 9th, 2009.

[37] Sun Microsystems, Inc. *Java*TM *Platform, Standard Edition 6 API Specification*, June 2009.
http://java.sun.com/javase/6/docs/api/, accessed on June 9th, 2009.

[38] Sun Microsystems, Inc. *MySQL 5.1 Reference Manual - 4.5.4. mysqldump — A Database Backup Program*, June 2009.
http://dev.mysql.com/doc/refman/5.1/en/mysqldump.html, accessed on June 9th, 2009.

[39] Sun Microsystems, Inc. *MySQL 5.1 Reference Manual - CREATE TABLE Syntax*, June 2009. http://dev.mysql.com/doc/refman/5.1/en/create-table.html, accessed on June 9th, 2009.

[40] Sun Microsystems, Inc. *MySQL Documentation*, June 2009. http://dev.mysql.com/doc/, accessed on June 9th, 2009.

[41] Sun Microsystems, Inc. *PHP Grammar Defintion for use with JavaCC*, June 2009. https://javacc.dev.java.net/files/documents/17/14269/php.jj, accessed on June 9th, 2009.

[42] Sun Microsystems, Inc. *The JavaBeans 1.01 specification*, June 2009. http://java.sun.com/javase/technologies/desktop/javabeans/docs/spec.html, accessed on June 9th, 2009.

[43] The Apache Software Foundation. *Apache Module mod_rewrite*, June 2009. http://httpd.apache.org/docs/2.0/mod/mod_rewrite.html, accessed on June 9th, 2009.

[44] The Apache Software Foundation. *Apache Struts 2 Documentation - Core Developers Guide*, June 2009. http://struts.apache.org/2.1.8.1/docs/core-developers-guide.html, accessed on June 9th, 2009.

[45] The Apache Software Foundation. *URL Rewriting Guide*, June 2009. http://httpd.apache.org/docs/2.0/misc/rewriteguide.html, accessed on June 9th, 2009.

[46] The Linux Documentation Project. *Introduction to Linux - A Hands on Guide*, Nov 2009. http://tldp.org/LDP/intro-linux/intro-linux.pdf, accessed on May 21st, 2009.

[47] The PHP Group. *Classes and Objects (PHP 5)*, June 2009. http://at.php.net/manual/en/language.oop5.php, accessed on June 9th, 2009.

[48] The PHP Group. *PEAR Manual*, October 2009. http://pear.php.net/manual/en/, accessed on June 9th, 2009.

[49] The PHP Group. *PHP Function List*, June 2009. http://www.php.net/quickref.php, accessed on June 9th, 2009.

[50] The Propel Project. *Propel 1.4 Documentation*, June 2009. http://propel.phpdb.org/trac/wiki/Users/Documentation/1.4, accessed on June 9th, 2009.

[51] Tilley, S. A Reverse-Engineering Environment Framework. Technical report, Carnegie Mellon Software Engineering Institute, April 1998. http://www.sei.cmu.edu/pub/documents/98.reports/pdf/98tr005.pdf.

[52] TIOBE Software. *TIOBE Programming Community Index Definition*, May 2009. http://www.tiobe.com/index.php/content/paperinfo/tpci/tpci_definition.htm, accessed on May 21st, 2009.

[53] TIOBE Software. *TIOBE Programming Community Index for May 2009*, May 2009. http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html, accessed on May 21st, 2009.

[54] Ulrich, W. *Legacy Systems: Transformation Strategies*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2002.

[55] Ulrich, W. A Status on OMG Architecture-Driven Modernization Task Force. Technical report, The Object Management Group (OMG), 2004.
http://adm.omg.org/MELS_EDOC2004_Ulrich_Extended_Submission_Revised.pdf, accessed on June 9th, 2009.

[56] Valade, J. *PHP- und MySQL-Applikationen für Dummies*. WILEY-VCH Verlag GmbH & Co. KGaA, Weinheim, 2006.

[57] Wage, J. and Vesterinen, K. *Doctrine ORM for PHP*. Sensio SA, April 2009.

[58] Web Models s.r.l. *Web Ratio*, June 2009.
http://www.webratio.com/, accessed on June 9th, 2009.

[59] yaml.org. *YAML, Version 1.2*, June 2009.
http://yaml.org/spec/1.2/spec.pdf, accessed on June 9th, 2009.