Master's Thesis

Generating Web Applications with Abstract Pageflow Models

carried out at the

Information Systems Institute
Distributed Systems Group
Vienna University of Technology

under the guidance of
o.Univ.Prof. Dr. Dustdar Schahram
and
Univ.Ass. Dipl.Ing. Vasko Martin
as the contributing advisor responsible

by

Ernst Oberortner
Poststrasse 4
9551 Bodensdorf
Matr.Nr. 0027144

Bodensdorf, 03. May 2007 _____

Acknowledgements

First and mostly important I thank my parents, Christine and Ernst, who always supported me during my whole studies.

Furthermore I want to thank my advisors Schahram Dustdar and Martin Vasko for their help, guidance and standby during the realization of this work.

Finally I thank all my colleagues and friends that I met during my university life for their great teamwork and collaboration in order to graduate studies.

Abstract

Due to the fact of platform-independence and worldwide accessibility, the popularity of the Web leads to countless Web applications. In the course of this work, a possibility for an automated generation of Web applications based on a MDA (Model-Driven Architecture) is introduced.

This work is concentrated on the definition of a meta-model for modeling Web applications based on the principles of the MVC (Model-View-Controller) pattern. The main task is the automated generation of modeled Web applications. Models of Web applications contain information about graphical user interfaces on Web pages and the pageflow within the Web application. Hence the defined meta-model specifies the syntax and structure of Models. Furthermore the meta-model serves for validating models.

The primary aim is the automated generation of the layout and structure of Web pages, hence the View of the MVC pattern. Furthermore an XML file is generated that serves as input for the Controller that controls the pageflow. For the time being the developer is responsible for the Model of the MVC pattern. Therefore a methodology for dealing with generated and manually written code is developed in order that the manually written code does not become overwritten in a subsequent generator run.

To prove our approach of generating Web applications based on a MDA, a prototype is introduced. This prototype is a JSF (JavaServer Faces) Web application for the Apache Tomcat Web server.

Zusammenfassung

Die Beliebtheit von Web Applikationen ist aus Gründen der Plattformunabhängigkeit und des weltweiten Zugriffs stark gestiegen. Aus diesem Grund beschäfigt sich diese Arbeit mit der automatischen Generierung von Web Applikationen basierend auf einer MDA (Model-Driven Architecture).

Insbesondere konzentriert sich diese Arbeit auf die Definition eines Meta-Modells für die Modellierung von Web Applikationen welche auf dem MVC (Model-View-Controller) Pattern aufbauen. Die Hauptaufgabe besteht in der automatischen Generierung der modellierten Web Applikationen. Die Modelle der Web Applikationen beinhalten Informationen über die Webseiten und den Page-Flow innerhalb der zu generierenden Web Applikation. Daher wurde ein Meta-Modell entwickelt welches die Syntax und die Struktur solcher Modelle beschreibt. Weiters dient das Meta-Modell zur Validierung der Modelle.

Das Hauptaugenmerk liegt in der Generierung der grafischen Oberflächen von Webseiten, also im View des MVC Patterns. Der Controller steuert den Page-Flow. Eine XML Datei wird vom Generator erzeugt welche die Informationen über den Page-Flow beinhaltet und als Input für den Controller dient. Zurzeit ist der Programmierer für die Programmierung des Modells des MVC Patterns zuständig. Aus diesem Grund muss bei der Codegenerierung zwischen generiertem und dem vom Programmierer geschriebenen Code unterschieden werden um ein Überschreiben des handgeschriebenen Codes bei einer erneuten Codegenerierung zu verhindern.

Um die Arbeitsweise unseres Ansatzes zu demonstrieren haben wir einen Prototyp erstellt. Dieser Prototyp ist eine JSF (JavaServer Faces) Web Applikation welche auf dem Apache Tomcat Web Server ausgeführt werden kann.

Contents

1 Introduction

The main purpose of this project is the development of a Model-Driven Architecture (MDA) ([9]) for an automatic generation of Web applications that are based on the Model-View-Controller (MVC) ([8]) pattern. To demonstrate the functioning of our approach, JavaServer Faces (JSF) ([6]) are used because they are popular Web applications based on the MVC pattern. For the realization of this project the Eclipse Modeling Framework (EMF) ([15]) in companion with the openArchitectureWare (oAW) ([18]) plug-in for Eclipse ([13]) were used because they provide helpful and strong facilities for modeling software systems and code generation.

The MVC pattern separates data and their graphical appearance. The pageflow is controlled by the Controller. The Model is responsible for fetching data (e.g. by accessing a database) and the View is responsible for the graphical appearance of the data within Web pages. The JSF framework provides an implemented Controller that needs an XML file that describes the pageflow as input. JavaServer Pages (JSP) represent the View and Enterprise Java Beans (EJB) represent the Model.

Our MDA is built-up on models that describe the structure of a Web application that should be generated. These models contain information about graphical user interfaces on Web pages as well as the pageflow. To define the syntax of models, the so-called Domain Specific Language (DSL), a meta-model was introduced. The meta-model is also responsible for the validation of models.

The modeling of Web applications can be done in any CASE tool that provides an exportation of the model to the eXtensible Metadata Interchange (XMI) file format. This XMI file is passed to our generator and if the model is valid, the Web application is generated.

One benefit of applying a MDA is to reduce the time-consuming process of developing Web applications. Furthermore the quality of the generated code as well as the maintenance are improved.

1.1 Motivation

The popularity of the Web and its advantages as a client-server platform led to countless Web applications. Web applications are global environments for delivering all kinds of applications. One reason for the popularity of Web applications is the facility to access an application from all over the world on any platform. Furthermore the maintenance of Web applications is arranged in a centralized way at minimal costs [4].

We base the necessity of developing Web applications in a rapid and high-quality way on the following statements:

Web Applications are becoming the first choice for most business application development [23].

Designing and maintaining Web applications is one of the major challenges for the software industry [22].

The maintenance of a Web application is getting difficult due to the inherent complexity of the system [24].

A MDA improves the quality and speed of developing Web applications as well as their maintenance. Furthermore a MDA unifies the steps of development and the integration from business modeling through architectural and application modeling [12].

To prove our approach of generating Web applications based on a MDA, the JSF framework is used because it is currently a popular Web oriented MVC implementation. Most Web applications are based on the MVC pattern for separating data and their presentation. The JSF framework provides server side user interface components for Web applications based on the Java technology. The main reason for our choice of the JSF framework lies in the ready-made Controller that controls the pageflow.

1.2 Problem Definition

The main purpose of this project lies in an automated generation of Web applications based on a MDA. A MDA achieves a certain level of abstraction that leads in an improvement of automation, productivity, quality and maintenance in the field of developing Web applications. Moreover MDA aims for interoperability, portability and standardization of models for popular application areas [3].

Web applications are described in models that define the user interfaces of Web pages as well as the pageflow. The pageflow describes all kinds of subsequent Web pages of a certain page. Our approach aims to a process-oriented definition of the pageflow to improve the clarity and understanding. For the definition of the structure and syntax of models, a meta-model has to be defined. Hence, the meta-model defines the DSL. Each model is an instance of the meta-model. The meta-model is also used for validating models.

Our approach describes a MDA for generating Web applications based on the MVC pattern. To demonstrate the functioning of our approach we are concentrated in an automated generation of JSF Web applications that are deployed especially on the Apache Tomcat Web server. Therefor a prototype is introduced that is described later in this work. Further works can be done to generate Web applications that can be deployed not only for a certain Web server.

Our defined meta-model supports the modeling of Web pages and the pageflow. The Controller is given by the JSF framework that controls the pageflow. Our code generator generates an XML file that serves as input for this Controller. This XML is built-on the given model that describes the Web application. Presently the developer is responsible for the Model of the MVC pattern, hence for fetching data, e.g. accessing a database. This results in the problem of handling generated and manually written code.

At present our generator generates the Web pages, a configuration file for the Tomcat Web server, an XML file that contains the information about the pageflow that serves as input for the JSF Controller, Java Beans to store data as well as an XML file for Apache Ant that eases the deployment of the generated Web application for the Tomcat Web server.

1.3 Organization of this thesis

This section gives an overview of the sections in this report and their contents.

- **Section 2 - Theory**
  This section describes the main technologies used in this work. The characteristics of the MVC pattern are specified. Furthermore we introduce the JSF framework. Then an overview of MDA is given where the main part lies in the Metamodeling section. Moreover the EMF as well as the oAW plug-in for Eclipse are described. The Theory section ends with an overview of possibilities how generated and handwritten code can be separated.

- **Section 3 - Related Work**
  The Related Work section gives an overview about research works for modeling Web applications. We specify the characteristics of other approaches introduced by researching teams on other universities or laboratories.

- **Section 4 - Description of our Approach**
  Section 4 specifies our defined approach how to generate a given modeled Web application. First we introduce our defined meta-model. Afterwards we describe how the Web application is generated with the oAW plug-in for Eclipse. Subsequently we introduce a prototype that demonstrates the functioning of our approach. The prototype is a JSF Web application that can be deployed on the Apache Tomcat Web server.

- **Section 5 - Evaluation**
  This section serves for an evaluation of our approach. We evaluate our defined meta-model, the defined templates for generating the Web application, the prototype as well as our utilized solution for the problem of handling generated and manually written code.

- **Section 6 - Further Work**
  Section 6 qualifies how our approach can be extended and improved in future work.

- **Section 7 - Summary and Conclusion**
  The Summary and Conclusion section repeats all important aspects of this work and serves as conclusion of our work.

2 Theory

2.1 The Model-View-Controller (MVC) Pattern

Web applications present contents in numerous pages containing various data to users. Developer teams are responsible for the design, implementation and maintenance of such Web applications. Therein rests the problem that multiple types of user interfaces must be supported, e.g. HTML Web pages for users and Java Web pages for developers. This problem forces that the same data needs to be accessed in different views. Furthermore an update of the same data can be done through different user interfaces. Supporting multiple types of views and interactions should not impact the components that provide the core functionality of the Web application [7].

The MVC pattern is used to put this things right by separating the core business functionality from the presentation and control logic. Such separation allows multiple views of the same data. This eases to implement, test and maintan multiple clients.

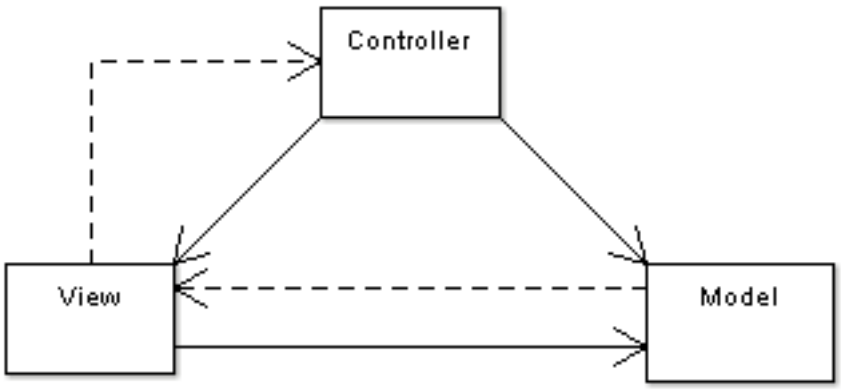

Figure 1: Model-View-Controller

Figure 1 demonstrates the division of the MVC pattern into the Model, View and Controller components and their relationships. Solid lines indicate direct associations and dashed lines indirect associations [8]. The MVC pattern is divided into three elements:

- **Model**
  The Model encapsulates the functional core of an application. It represents the data and grants access to the data.

- **View**
  The View is responsible for the rendering of the data of the Model, typically user interface elements. It specifies exactly how the data should be presented. The View must update the presentation of the data if the data in the Model has been changed [8]. Generally, the View has read-only access to the Model because the View should not change the state of the Model.

- **Controller**
  The Controller is responsible for calling methods of the Model that change the data. The Controller and the View have equal opportunity to access the Model. The Controller does not copy data values from the Model to the View, although it may place values in the Model and tell the View that the data of the Model has been changed. The Controller may also select a new View that should be presented to the user, e.g. a Web page of results [8].

In Web applications based on the MVC pattern the View is the actual HTML document and the Controller is responsible for the content within the HTML code as well as the control of the pageflow. The Model is represented by the actual content stored in a database or XML-files.

The following scenario shows actions that occur if a user interacts with the View [8]:

1. The user interacts with an user interface
2. The View recognizes that an action has occurred and calls a method that is registered to be called when such an action occurs
3. The View calls the appropriate method on the Controller
4. The Controller accesses the Model

5. The Model notifies all Views that the data has been changed. In Java technology-based applications the Controller may also be responsible for updating the View.

The benefits of MVC are:

- Substitutable user interfaces
  Different Views and Controllers can be substituted to provide alternate user interfaces for the same Model.
- Multiple simultaneous views of the same Model
- Synchronized Views
- Easier user interface changes
- Easier testing

On the other hand the drawbacks of MVC are:

- Increased complexity
- Close coupling of View and Controller to the Model:
  Changes of the data require changes in the View and may require additional changes in the Controller
- Close coupling between View and Controller
  A strict separation between View and Controller is difficult

2.2 The JavaServer Faces (JSF) Framework

The JSF framework is a server-side framework for user interface components for Web applications based on the MVC pattern.

- **Model**
  The Model is represented by Enterprise Java Beans (EJB).
- **View**
  The View is represented by JavaServer Pages (JSP).
- **Controller**
  The Controller is represented by Java Servlets.

First we'll take a closer look to EJB and JSP:

- **Enterprise Java Beans (EJB)**
  An EJB is a pieces of code that has properties and methods to implement modules of the business logic [5]. There are three kinds of EJB:
  - **Session Beans**
    A Session Bean represents a transient conversation with the client. They are valid until the client finishes its execution. [5].
  - **Entity Beans**
    Entity Beans are distributed objects that have a persistent state where the persistency may or may not be managed by the bean itself. Hence, they are divided into Container-Managed Persistence (CMP) and Bean-Managed Persistence (BMP).
- **JavaServer Pages (JSP)**
  JSP allow the programmer to put Java code and certain pre-defined actions into HTML documents. A JSP page is a text-based document that contains static data and JSP tags that construct dynamic content [5]. The JSP tag libraries act as extensions to the standard HTML or XML. Tag libraries provide a platform independent extension of a Web server. Static data can be expressed in any text-based formats such as HTML [5]. JSP pages are compiled into Java Servlets by a JSP compiler that generates a Java Servlet in Java code or directly in byte-code. JSP can be seen as an abstraction of Java Servlets. Sun

recommends that the MVC pattern should be used with the JSP files in order to split the presentation from request processing and data storage. Either regular Java Servlets or separate JSP files are used to process the request. After the request processing has finished, control is passed to a JSP page for creating the output.

2.2.1 JSF Technology

Like mentioned above, the JSF framework provides server-side components for Java based Web applications. The framework consists of two main components [5]:

- An API for representing user interface components and managing their state, handling events, server-side validation, defining page navigation and supporting internationalization and accessibility
- Two JSP custom tag libraries for expressing user interface components within a JSP page

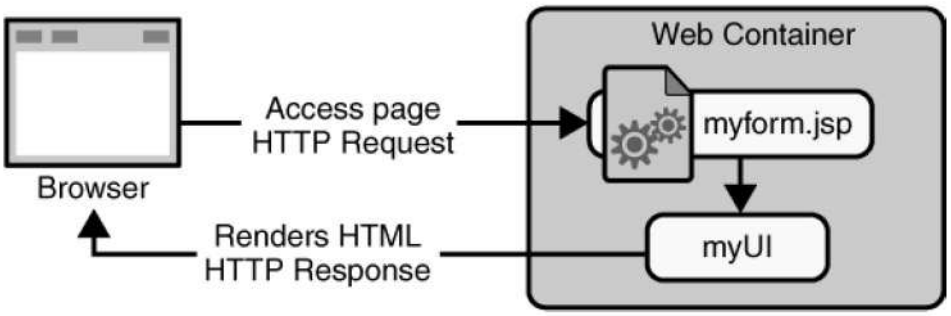

Figure 2: JavaServer Faces [5]

Figure 2 shows the organization of JSF Web applications. The client (*Browser*) requests the JSP page *myform.jsp* that contains JSF tags. The user interface created by the JSF framework, that is represented by *myUI* in the figure, runs on the server and renders back to the client [5]. The user interface manages the objects referenced by the JSP page. These objects include:

- User interface component objects that are mapped by the tags in the JSP page

- Event listeners, validators and converters
- Java Bean components that contain the data and application specific functionalities.

2.2.2 JSF Web Applications

A typical JSF Web application consists of [5]:

- A set of JSP pages
- A set *Java Beans* that define the properties and functionality for user interfaces
- An application configuration file that defines page navigation rules and configures the used Beans (*faces-config.xml*)
- A deployment descriptor (*web.xml*)
- Possibly a set of custom objects that include validators, converters or listeners
- A set of custom tags for representing custom objects

The next paragraph describes the steps of developing a JSF Web applications that can be deployed on the Apache Tomcat Web server.

2.2.3 Guidance for developing JSF Web Applications

The development of JSF Web applications consists of the following steps [5]:

- Mapping the `FacesServlet` instance in the *web.xml* file
- Creation of the JSP Web pages using the user interface components and core tags
- Defining the pageflow in the *faces-config.xml* file
- Development of the *Java Beans*
- Adding managed bean declarations in the *faces-config.xml* file

2.2.3.1 Mapping the `FacesServlet` instance

The `FacesServlet` is the controller for the entire Web application. It receives and processes incoming requests and has to be included by every JSF application. The following snippet shows the binding of the `FacesServlet` in the deployment descriptor *web.xml* [5].

```
<servlet>
  <display-name>FacesServlet</display-name>
  <servlet-name>FacesServlet</servlet-name>
  <servlet-class>javax.faces.webapp.FacesServlet</servlet-class>
  <load-on-startup>1</load-on-startup>
</servlet>

<servlet-mapping>
  <servlet-name>FacesServlet</servlet-name>
  <url-pattern>/appname/*</url-pattern>
</servlet-mapping>
```

Listing 1: Mapping the `FacesServlet` instance

The `<servlet-mapping>` tag specifies that any requests made through the `<url-pattern>/appname/</url-pattern>` will be processed by the `FacesServlet` specified through the `<servlet-name>` tag. By specifying any (`*`) request after `/appname/` only those with the .jsp or .jsf suffix will be processed by the `FacesServlet`.

That is all what has to be included in the deployment descriptor *web.xml* of the JSF Web application. Subsequently the creation of the Web pages can start.

2.2.3.2 Creation of JSP Web pages

Every JSP page needs access to the two standard JSF tag libraries, the HTML component tag library and the core tag library using `taglib` declarations [5]:

```
<%@ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>
<%@ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>
```

Listing 2: Loading the standard JSF tag libraries

Each tag library has to be assigned with a prefix. The HTML component tag library is declared with the prefix `h` and the core tag library with the prefix `f`.

Afterwards the creation of the *view* can take place. All JSF component tags must be inside of a `<f:view>` tag.

```
<f:view>
  <h:form id="formID">
  ...
  </h:form>
</f:view>
```

Listing 3: Creation of the Web pages

The `<h:form>` tag represents a set of input components, such as textfields or menues, that allow the user to input data and submit it to the server [5].

Component	Declaration
Output Text	`<h:outputText id="outputID" value="{beanName.attribute}" />`
Input text	`<h:inputText id="inputID" label="input label" value="{beanName.attribute}" />`
Button	`<h:commandButton id="buttonID" action="buttonAction" value="Submit" />`
Hyperlink	`<h:commandLink id="linkID" action="linkAction"> <h:outputText value="linkValue" /> </h:commandLink>`

Table 1: JSF user interface component tags

Table 1 shows the main JSF user interface component tags that are used for the definition of user interfaces for user interaction. Each component can get an ID by setting the *id* attribute. The *value* attribute for input and output components (`<h:inputText>` and `<h:outputText>`) binds the component to the given attributes of the given bean. The `<h:commandButton>` tag is utilized for sending the entered data in the textfields of the form to the server. Each `commandButton` contains an *action* attribute that helps the navigation mechanism decide which page to open next [5]. A `<h:commandLink>` consists beside the *id* attribute of an *action* attribute that defines the outcome of the link. The `<h:commandLink>` tag must include a `<h:outputText>` tag that defines the caption of the link.

After the creation of the Web pages is done the pageflow definition can start. The next paragraph describes how the pageflow of a JSF Web application is described.

2.2.3.3 Defining the Pageflow

The page navigation is defined in the application configuration file *faces-config.xml*. Each page of the application can have so-called outcomings. The navigation rules in the application configuration file define which page has to be displayed when the current page delivers a certain outcome. A page delivers outcomings when the user clicks a button or a hyperlink.

The following snipped shows an example of defining a navigation rule:

```
<navigation-rule>
   <from-view-id>/login.jsp</from-view-id>
   <navigation-case>
       <from-outcome>success</from-outcome>
       <to-view-id>/success.jsp</to-view-id>
   </navigation-case>
   <navigation-case>
       <from-outcome>invalid</from-outcome>
       <to-view-id>/invalid.jsp</to-view-id>
   </navigation-case>
</navigation-rule>
```

Listing 4: Definition of navigation rules

This navigation rule is an example navigation rule defined for the page *login.jsp*. A navigation rule consists of multiple navigation cases. Each navigation cases defines which page has to be displayed for a specific outcome of the *login.jsp* page. In this example it is defined to go to the page *success.jsp* if the outcome equals to `success` or to go to the page *invalid.jsp* if the outcome is equal to `invalid`. The logical outcomings have to be defined in the *action* attribute of a `commandButton` or of a `commandLink`, e.g.

```
<h:commandButton id="buttonID"
                 action="success"
                 value="Submit" />

<h:commandLink id="linkID"
               action="success">
   <h:outputText value="linkValue" />
</h:commandLink>
```

The logical outcome can also come from the return value of a method of a Java Bean. For example, the method checks whether the entered values for username and password are valid or not. If they are valid it returns the String `success` and if not it returns the String `invalid`. If the outcome should be delivered by a method of a Bean, the definition of the *action* attribute of the `commandButton` or `commandLink` looks like follows:

```
<h:commandButton id="buttonID"
                 action="#{userBean.checkData}"
                 value="Submit" />

<h:commandLink id="linkID"
               action="#{userBean.checkData}">
   <h:outputText value="linkValue" />
</h:commandLink>
```

Listing 5: Binding of outcomings to Bean methods

After the defining the pageflow, the development of the Java Beans is done. The creation of Java Beans is described in the next paragraph.

2.2.3.4 Development of the *Java Beans*

Java Beans define properties and methods that are associated with user interface components. A typical JSF Web application couples each page of the application with a Bean [5].

The following example shows the binding of a textfield with the attribute *username* of a Bean called `UserBean`:

```
<h:inputText id="userName"
             label="Username"
             value="#{UserBean.username}">
```

Listing 6: Binding of a textfield with an attribute of a Bean

The declaration of the Java Bean `UserBean` is described in the following listing:

```
public class UserBean {
    private String username = null;
    ...

    public void setUsername(String username) {
         this.username=username;
    }

    public String getUsername() {
         return this.username;
    }
    ...

}
```

Listing 7: Declaration of a Bean

Every Java Bean needs `get-` and `set`-methods for the bonded attributes to user interface components, e.g. `getUsername` and `setUsername`.

The next paragraph demonstrates the definition of the declared Java Beans in the JSF Web applications configuration file *faces-config.xml*.

2.2.3.5 Adding managed bean declarations

Every managed Bean has to be defined in the application configuration file *faces-config.xml*. The following snipped demonstrates the definition of the `UserBean` in the *faces-config.xml* file:

```
<managed-bean>
   <managed-bean-name>UserBean</managed-bean-name>
   <managed-bean-class>UserBean</managed-bean-class>
   <managed-bean-scope>session</managed-bean-scope>
   <managed-property>
      <property-name>username</property-name>
      <property-class>String</property-class>
      <value>null</value>
   </managed-property>
</managed-bean>
```

Listing 8: Adding managed Bean declarations

Each Bean is defined within the `<managed-bean>` tags. The `<managed-bean-name>` tag defines the name of the Bean and the `<managed-bean-class>` tag depicts the name of the Java class where the Bean is defined.

The `<managed-bean-scope>` defines the duration of the availability of the Beans and accepts four scopes:

- **none**
  The Bean is instantiated new when an item is referenced. A possible reason to use this kind of scope is when a managed Bean references another managed Bean.

- **request**
  A Bean with a `request` scope will have values that are stored only for the duration of a single request.

- **session**
  Bean properties will stay alive during multiple requests. Objects stored

in a session will expire if the session times out or if they are cleaned by the application explicitly.

- **application**
  Beans created in an `application` scope will exist during the entire lifetime of the Web server.

A very important point about managed Beans referencing other managed Beans is that a managed Bean can only refer to Beans that do not have a scope that is equal nor longer.

Initialization values of the properties of the Bean can be set within the `<managed-property>` tags. The name of the property of the Bean is defined within the `<property-name>` tags. The type is specified through the `<property-class>` tag. The initialization value is given by the `<value>` tag.

After all parts of the development process of a JSF Web application are finished the JSF Web application can be deployed on a Web server, e.g. Apache Tomcat. The advantages of JSF Web applications are described in the next section.

2.2.4 Benefits of JSF Web applications

The benefits of JSF Web applications are [5]:

- Clean separation between behavior and presentation
  JSF can map HTTP requests to component specific event handling and manage elements of user interfaces as stateful objects on the server.
- Improves sectioning of development teams
  Due to the separation of logic from presentation each developer in a Web application development team can focus on his or her field of functions of the development process.
- Rich architecture for user input validation, component state managing, component data processing and event handling.

2.3 Model Driven Architecture (MDA)

A MDA is an evolutionary step in the field of developing software [10] and it provides an approach for developing software with a strict separation of functionality and technology. The main idea behind MDA is to separate the business and application logic from the underlying platform. MDA was adopted in 2001 by the Object Management Group (OMG) to use models in the software development process [10]. The main motivations of MDA are interoperability and portability of software systems. Interoperability aims to vendor independence through standardization and portability to platform independence [3].

2.3.1 Terminology

2.3.1.1 Model

A *Model* describes or specifies a system and its environment for a certain purpose [10]. It is an abstract representation of structure, functionality and behavior of a system [3]. Normally models are defined with the Unified Modeling Language (UML).

2.3.1.2 Model Driven

A MDA involves models in the software development process. MDA is *Model Driven* because it operates on underlying models for understanding, design, construction, deployment, operation, maintenance and modification [10].

2.3.1.3 Architecture

An *Architecture* describes parts and connectors of a system and rules for the interaction between the parts using the connectors. A MDA describes the different kind of models and their relationships [10].

2.3.1.4 Platform

A *Platform* is a well defined architecture with an appropriate runtime system [3].

2.3.1.5 Platform Independent Model (PIM)

A *PIM* is a view of a system from a platform independent viewpoint [10]. Functional specifications are defined in the PIM by using a formal modeling language, e.g. UML. A PIM is an abstraction of technological details [3].

2.3.1.6 Platform Specific Model (PSM)

A *PSM* is a view of a system from a platform specific viewpoint. A PSM is achieved by a transformation of the PIM. Next to the specifications in the PIM, the PSM contains details that specify how that system uses a particular type of platform [10]. A PSM uses the concepts of a platform to describe a system [3].

2.3.1.7 Model Transformations

A *Model Transformation* maps a model to another model of the same system. Figure 3 demonstrates the transformation of a PIM into a PSM by using transformation rules [10]. Most MDA tools define the rules of transformations in so called templates.

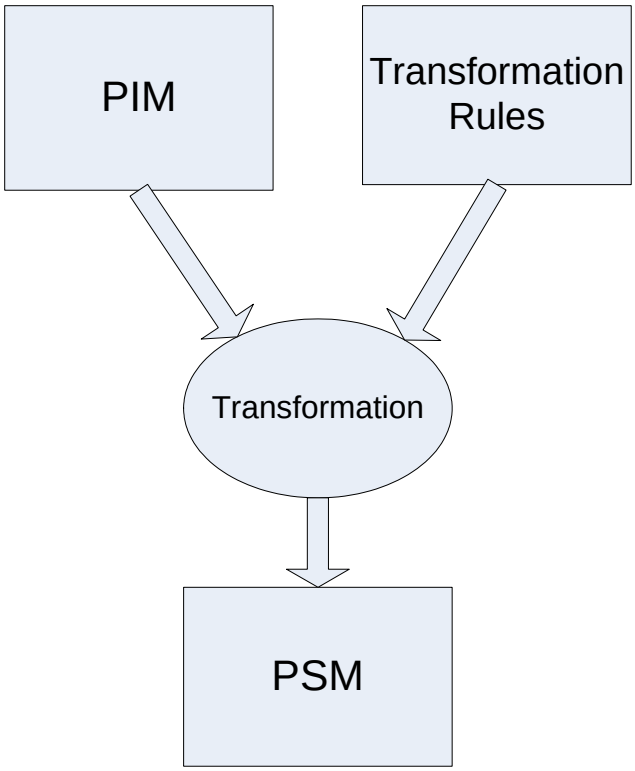

Figure 3: Model Transformation

2.3.1.8 Domain

A *Domain* is a bounded region of interests or knowledge. Domains can be assembled by multiple sub domains [3].

2.3.1.9 Meta-model

A *Meta-model* is responsible for the formalization of the structure of a domain. A meta-model enfolds abstract syntax and static semantic [3].

2.3.1.10 Abstract Syntax

An *Abstract Syntax* defines the structure of a language. The implementation of an abstract syntax is called concrete syntax. Multiple concrete syntaxes can have one abstract syntax in common.

2.3.1.11 Static Semantic

The *Static Semantic* immobilizes the criteria of shapeliness. A typical example is that variables in a programming language have to be declared before they can be assigned or used [3]. Static semantic is very important in the field of model driven software development because it can be used for the detection of malfunctions of the model.

2.3.1.12 Domain Specific Language (DSL)

A *DSL* is used to model the key aspects of a domain. A DSL contains a meta-model including the static semantic and the corresponding concrete syntax [3]. Furthermore it needs a semantic to denotate the constructs of the meta-model. The semantic is relevant to help modelers to understand the meanings of the provided constructs.

2.3.1.13 Meta Object Facility (MOF)

The *MOF* describes the meta-meta-model. MOF is used to define the meta-model. This includes the definition of classes, attributes, relations and constraints of the meta-model. Furthermore the MOF contains repositories to save meta information about meta-models [11]. The main disadvantages of the MOF is that there is no support provided for the definition of concrete syntax, versioning and the composition of sub-meta-models to a meta-model [3].

2.3.1.14 XML Metadata Interchange (XMI)

XMI is a mapping of XML for MOF. XMI builds the basis for the interoperability of models between different MDA tools [3].

2.3.2 Aims of MDA

- **Increasing the speed of development**
  Executable code can be achieved in a fast way by applying model transformations
- **Better maintenance of software**
  Bugs within the generated code can be abolished by simply changing the transformation rules. This results in a better **avoidance of redundancy** and **improved maintenance facilities**.
- **Higher degree of reusing software**
  Once defined architectures, modeling languages and transformations can be reused for the development of other software systems.
- **Better manageability of complexity by abstraction**
  The efforts of programming should be eliminated through modeling languages.
- **Interoperability** and **portability**

2.3.3 Metamodeling

Metamodeling is one of the main issues in the model driven software development process. Metamodeling is used for the following problems [3]:

- **Construction of DSL**:
  The abstract syntax of the language is defined by the meta-model.
- **Validation of models**:
  Validation of models against defined constraints in the meta-model.
- **Model-to-model transformations**:
  Definition of mapping rules between meta-models.
- **Generation of code**:
  Templates for the generation of code refer to the meta-model of the DSL.
- **Integration of tools**:
  A meta-model is used for the adaption of modeling tools for a certain domain.

A meta-model defines the structure of models. It defines in an abstract way the constructs of a modeling language and their relations. Hence, a meta-model defines the abstract syntax and the static semantic of a modeling language. Vice versa every formal language (e.g. Java or UML) owns a meta-model [3].

All models are instances of a meta-model [3].

Models can be described by any modeling language but the domain should be crucial for the assortment of the language. It is important that the selection of the modeling language depends on the applicable tools provided by the modeling language. Hence, UML is the primary choice nowadays [3].

Figure 4 points to the four meta layers for metamodeling defined by the OMG MOF [3].

- M0: Layer M0 is responsible for the construction of instances of classes defined in M1. This is the assignment of values to the attributes of the class.

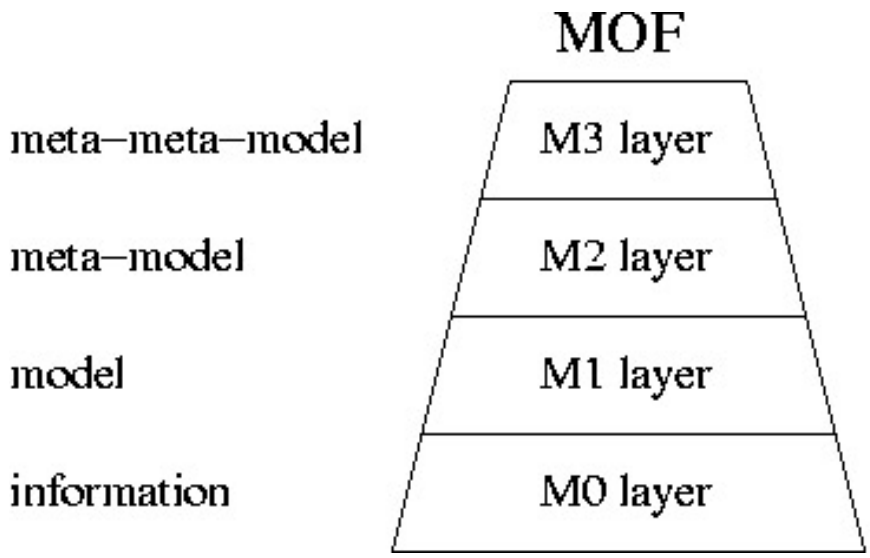

Figure 4: The four meta layers of the OMG

- M1: The declaration of classes takes place in layer M1.
- M2: M2 is the meta-model. It serves for the definition of constructs that can be used in the underlying layer M1. Hence, the elements of layer M1 are instances of layer M2.
- M3: M3 defines so called Meta Object Facilty (MOF) classes. The MOF is the meta-meta-model defined by the OMG. MOF is used to define modeling languages for layer M2, e.g. for UML.

There are no more meta layers above the MOF in the OMG model. Therefore, the MOF describes itself [3].

2.4 Eclipse Modeling Framework (EMF)

The EMF is a Java based framework and code generation facility for building tools and applications based on a structured model. EMF is a facility to generate correct and easily customizable Java code in an efficient way . EMF consists of two fundamental frameworks [15]:

- **core framework**
  The core EMF framework includes a meta-model (Ecore) for defining models. Beside it contains a runtime support for the models including change notification, persistence support with default XMI serialization, and a efficient API for manipulating EMF objects.
- **EMF.Edit**
  EMF.Edit extends and builds on the core framework, adding support for generating adapter classes that enable viewing and command-based editing of a model, and even a basic working model editor.

Because EMF uses XMI for the definition of models there are several ways to import the definition of models into EMF:

- **Creating the XMI file directly with a XML editor**
  This approach is very complex and only recommendable to those who have a good experience with XML.
- **Export the XMI document from a modeling tool**
  This is the most desirable approach.
- **Annotate Java interface with model properties**
  This is a low-cost approach to get the benefits of EMF and its code generator.
- **Use XML schema to describe the form of a serialization of the model**
  This approach is applicable for applications that must read or write a particular XML file format.

Once defining an EMF model the EMF generator can create a corresponding set of Java implementation classes. After generating the Java implementation classes one can add methods and instance variables to these generated classes and still regenerate from the model while the additions will be preserved during the regeneration [15].

The benefits of EMF are:

- Increasing the productivity
- Notifications of model changes
- Persistence support including default XMI and schema-based XML serialization
- Framework for model validation
- Efficient API for manipulating EMF objects
- EMF provides interoperability with other tools and applications based on EMF

2.5 openArchitectureWare (oAW)

oAW is a platform for model driven software development and is free available under the Eclipse Plugin License. It provides facilities to parse, validate and transform models as well as to generate code based on a model. Editors for oAW are based on the Eclipse platform. Hence, oAW has strong support for models based on EMF but it can also work with other models too, e.g. UML [18].

The core of oAW provides the following features [18]:

- A **workflow engine** that controls the workflow of the generator.
- A **suitable instantiator** to read any model.
- A **statically typed language** to check, modify and transform models as well as to generate code.
- **Xpand2** that is a powerful template language.
- A functional model transformation language **Xtend** to extend the meta-model.
- An Object Constraint Language (OCL) to define constraints that is called **Check**.

2.5.1 Workflow Engine

The oAW Workflow Engine is an XML based configuration language to describe the generators workflow. A generator workflow consists of workflow components that are executed sequentially in a single Java Virtual Machine (JavaVM). Workflow components are usually model parsers, model validators, model transformers and code generators [18].

A workflow description consists of a list of configured workflow components [18]:

```
<workflow>
 <property name='baseDir' value='./'/>
 <property file='\${baseDir}/my.properties'/>

 <component class="firstWorkflowComponent">
    <property value="prop1" />
```

```
  </component>

  <component class="secondWorkflowComponent">
     <property value="prop2" />
  </component>

  <component class="thirdWorkflowComponent">
     <property value="prop3" />
  </component>
</workflow>
```

Listing 9: Definition of workflow components

This workflow consists of three components where the order of the components is important. Each component consists of a property. Properties can be declared anywhere in a workflow file. Properties are separated into simple properties and properties files. A simple property can be used in attributes in the workflow file after their declaration. Property files use the Java properties file syntax [18].

2.5.2 Expression Framework

Check, Xtend and Xpand2 are built up on a common expression language and type system. Therefore they can operate on the same models without using different syntaxes. The expressions framework provides a uniform abstraction layer over different meta-meta-models like EMF Ecore. Additionally, it offers a statically typed expressions language [18].

2.5.3 Xpand2

Xpand2 is a special language inside the oAW framework that is used to control the output of the generator within templates. Templates are stored in files with a *.xpt* prefix. Templates must reside on the Java classpath of the generator process [18].

2.5.4 Xtend

Xtend provides the possibility to define libraries of independent operations and metamodel extensions based on either Java methods or oAW expressions. Those libraries can be referenced from all other textual languages, that are based on the expressions framework [18].

2.5.5 Check

Check is a DSL that is specialized on model validation. Model validation should happen as early as possible in a development process. The best way to achieve this is to integrate it in the model editor. Check files have the prefix *.chk* and must reside on the Java classpath of the generator process. Check can be used to specify constraints in model transformation steps. It can be used with all kinds of model representations as long as a suitable meta-model implementation is available [18].

2.6 Apache Ant

Apache Ant is a software tool for building software applications, primary Java applications. Ant is written in Java and therefor it needs a Java platform. Ant is similar to *make* but Ant uses XML for the description of the build process. The default XML file for Ant is named `build.xml`.

Each buildfile contains one project and at least one default target. Each target consists of one or more tasks which are executed.

- **Project**:
  The `<project>` tag consists of the attributes *name*, *default* and *basedir*. The optional *name* attribute contains the name of the project. The *default* attribute is also optional and can contain the name of a default target. The *basedir* attribute contains the base directory from which all path calculations are done.

- **Targets**:
  Targets can depend on other targets. Therefore the `<target>` tag provides beside the *name* attribute a *depends* attribute. The *name* contains the name of the target and the *depends* attribute the targets on which the current target depends. The *depends* attributes specify the order in which targets should be executed [2].

- **Tasks**:
  A task is a piece of code that can be executed. A task can have multiple attributes where the value of an attribute can contain references to a property. These references will be resolved before the task is executed.

A project can have a set of properties. These might be set within the `<project>` tag or in an external property file. Each property consists of a name and a value where the name is case-sensitive. Properties can be used in the values of task attributes.

2.7 Separation of Generated and Handwritten Code

Due to the fact that the developer is responsible for the Model of the MVC pattern in the course of this work, a solution for separating generated and non-generated code must be found. The simplest method to combine generated and handwritten code are so called *Protected Regions*.

2.7.1 Protected Regions

Protected regions are labeled areas within the generated code in which the developer can insert code. Protected regions are designated that they can be read by the generator and they will not be overwritten in a subsequent generator run [3].

The following snippet shows protected regions within a generated Java class:

```
public class GeneratedClass {

   ...

   public void foo() {
      //protected region begins - 0001
      //place code here
      //protected region ends - 0001
   }

   public void bar() {
      //protected region begins - 0002
      //place code here
      //protected region ends - 0002
   }

   ...
}
```

Listing 10: Protected regions within a Java class

Protected regions are labeled by comments and consecutive numbers in this example. The generator notices the designated region and does not overwrite the code between the comments `//protected region begins` and `//protected region ends`.

However, the main disadvantage of protected regions lies in the fact that the developer must work within the generated code and therefore the developer must understand the generated code which is not always easy. Furthermore the utilization of protected areas results in a more complex generator because

the generator has to recognize the protected areas and has to preserve the manually written code. Practical experiences show that a preservation of handwritten code is not easily accomplished, hence pieces of code get lost. Moreover the separation between generated and non-generated code becomes blurred because both are in the same file or rather in the same class.

2.7.2 Solutions for Protected Regions

As result of this disadvantages a solution must be found. Popular solutions are the utilization of interfaces, abstract classes and design patterns in object-oriented languages like Java. Figure 5, that was taken from [3], demonstrates possible object-oriented solutions for handling generated and non-generated code where the white boxes denote generated and black boxes non-generated code.

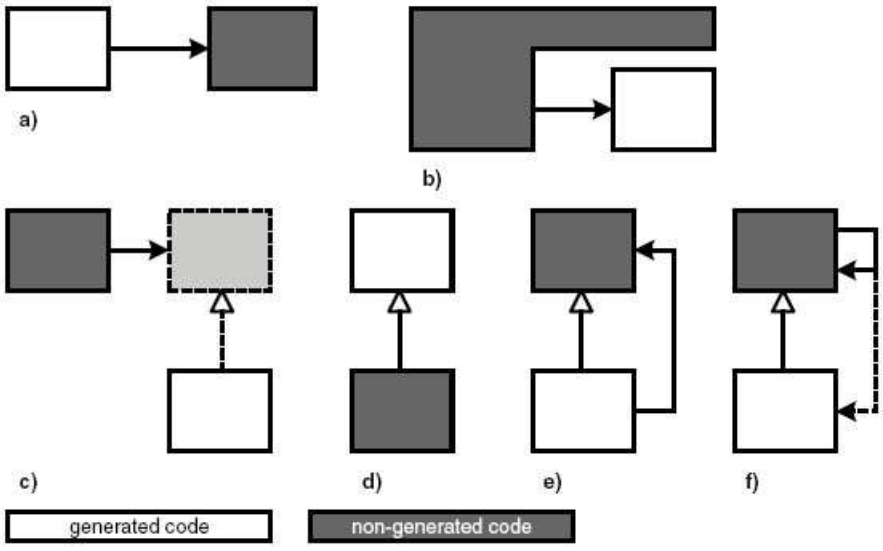

Figure 5: Handling generated and non-generated code [3]

Case a) demonstrates generated code that calls non-generated code. b) shows that manually written code calls generated code. Thereby a problem exists because manually written code must know the generated code and this can lead to unpleasant dependencies during the build process. Case c) denotes

that generated code can inherit from handwritten code or implement an handwritten interface. d) shows an implementation call that inherits from the generated class. e) depicts that a generated class can inherit from a non-generated class or invoke its operations. Case f) shows that manually written classes call operations of generated subclasses.

If files with generated code are never modified, the generated can can simply be overwritten in subsequent generator runs. Protected regions should be used if generated code must be modified manually. [3] gives the following important advices:

Keep generated and non-generated code in separate files!
Never modify generated code!

These statements discourage from the usage of protected regions. Protected regions should only be used if the target platform does not support any other options to place manually written code within generated code.

3 Related Work

A lot of work is done in the field of an automated generation of Web applications based on a MDA. We mention the three papers [19], [21] and [22] because they have something in common. All those works are concentrated on Web applications based on the MVC pattern. They introduced meta-models for the Model, the View as well as the Controller. Paper [24] describes the modeling of Web applications based on Java and XML technologies. [23] introduce a methodology for testing Web applications. Furthermore [25] describes JBoss Seam that offers facilities to model the pageflow of a Web application.

Paper [19] introduces the WebSA as well as the UWE (UML-based Web Engineering) approaches for the model-driven development of Web applications. The WebSA approach aims to avoiding gaps between Web design models and the final implementation. It covers all phases of the development process of Web applications where the focus lies on the software architecture. Besides it defines a set of architectural models for the specification of the architectural viewpoint. Furthermore it establishes an instance of the MDA development process. The WebSA approach is divided into three phases:

1. **Analysis phase**
   The analysis phase divides the Web application into the functional and architectural viewpoints. The functional perspective is given by Web functional models provided by Web methods. The architectural view contains a subsystem model and a configuration model to define the software architecture of the Web application.

2. **PIM-to-PIM transformations**
   The analysis models are transformed to platform independent design models. The output is an integration model that contains information about the functionality and architecture.

3. **PIM-to-PSM transformations**
   The model transformations for each platform are built-up on the integration model. The output of this phase is the specification of the Web application for a given platform.

The UWE approach defines a meta-model that acts as a conservative extension of the UML meta-model that has a mapping to an UML profile. UWE separates the modeling of different points of view into content, navigation structure, business process and presentation. The content is modeled by UML class diagrams. The navigation structure is based on all conceptual classes that are relevant for the navigation structure and represents the navigation paths. The behavior of the business logic describes the processflow in UML activity diagrams. The presentation model is used to sketch the layout of the web pages.

[20] uses the WebSA approach to achieve a logical architectural view for Web applications. Three models are defined by the WebSA approach that are responsible for the definition of the logical components and their relationships within a system.

- **Subsystem Model (SM)**
  The *Subsystem Model* reduces the complexity of the Web application by providing an abstract perspective of the logical architectural view. Subsystems obtained in this phase are later identified with each logical layer in the application.

  The subsystem model is divided into subsystems and dependency relationships. Subsystems define groups of software components to support the functionality of certain logical layer. Dependency relationships describe the relationship between subsystems.

- **Web Component Configuration Model (WCCM)**
  The *WCCM* consists of abstract elements that are produced in the refinement process performed on each subsystem. It contains abstract components as well as abstract connectors. Abstract components are abstractions of one or more software components with shared functionalities. Abstract connectors represent dependency relationships between two abstract components.

  Architectural design patterns are used for powerful configurations, reuse mechanisms and contributions to more efficient development processes.

- **Web Component Integration Model (WCIM)**
  For the connection of functional and architectural views under a com-

mon set of concrete components the *WCIM* is used. It is defined during the WebSA platform-independent phase. The WCIM consists of Concrete Components (CC), Modules (M) and Concrete Connectors (CN). CCs represent software components in a certain application domain. Ms are containers of one or more concrete elements. CNs express the relationship between two CCs.

Muller et al. [21] deal with the question of making a new meta-model or to customize an existing one. Customizing existing meta-models is known under the term Profiling. They are coming to the following conclusion:

Creating a new meta-model makes more sense than profiling when the semantic distance between existing UML modeling elements and newly defined modeling elements is becoming too large.

Furthermore a separation into three meta-models is described:

- **Business Model**
  The *Business Model* is built-up on the entity-based structural model introduced by paper [22]. UML class diagrams are used to represent the business classes. Furthermore the business model is used to describe session management, personalization, search or statistics. An action language Xion is introduced which describes the behavior of classes represented by their methods. Xion is based on the syntax of OCL augmented with Java-like control structures and affections. Xion is used to create, update and delete instances at runtime.

- **Hypertext Model**
  The *Hypertext Model* is an abstract description of composition of Web pages and navigation between Web pages. In other words, it describes how Web pages are linked and built. The composition describes the way the various Web pages are composed from other pages as well as the inclusion of information coming from the business model. The navigation describes the links and specifies the parameters between Web pages.

- **Presentation Model**
  The *Presentation Model* is responsible for the graphical appearance of the Web pages.

The Web Modeling Language (WebML) is introduced in paper [22]. WebML is based on four Models:

- **Structural Model**
  The *Structural Model* is responsible for the data content of a Web page.
- **Hypertext Model**
  The *Hypertext Model* consists of the Composition and Navigation Models. The Composition Model specifies which pages compose the hypertext and which content units make up a page. The Navigation Model describes how pages and contents are linked to form the hypertext.
- **Presentation Model**
  The *Presentation Model* expresses the layout and graphical appearance of the Web pages.
- **Personalization Model**
  The *Personalization Model* administrates user and user groups. Hence, contents can be stored user- or group-specific.

Chung and Lee [24] describe the modeling of Web applications using Java and XML related technologies. They consider Web applications that are built-on a three-tier architecture. Visual models are proposed and analyzed based upon criteria for relative model comparison. Two criteria are proposed: the degree of language independence and the degree of location independence. Furthermore comparison criteria are used to figure out how relatively difficult it is to model which component. A component means a physical and replaceable part of a system. To compare the modeling complexity of components the Rational Unified Process (RUP) in companion with UML is applied. By applying RUP to a software system the architectural views of models can be achieved.

Alava et al. [23] introduce a methodology to test Web applications based on a *Design View (DView)* of the pageflow model. A DView is a representation of the pageflow based on graphs. The methodology extracts information from a pageflow model and creates a Typed Attributed Directed Graph (TGraph) that is used in the analysis of test coverage criteria. TGraphs are used to

model an object-oriented view of the system where vertices and edges represent objects and relationships. TGraphs are used to apply traditional test coverage criteria in terms of *Action*, *Forward*, *Link* and *Page* are the test coverage criteria that are used by TGraphs. *Action* maps incoming HTTP requests to the corresponding methods for execution. *Forward* represents a destination to which the pageflow controller might be directed to. *Link*s represent the actual HTTP request from the JSP pages to the Actions. A *Page* is a JSP page that handles user interfaces. A TGraph is transformed to an Attributed Directed Graph (AGraph) and the resulting graph is analyzed to obtain traditional structural testing criteria.

JBoss Seam [25] provides the business process manager jBPM to represent business processes or user interactions as graphs. The graphs are defined in an XML dialect, called jPDL. jPDL is suitable to define the pageflow of a Web application. Seam provides two ways to define the pageflow:

- **JSF/Seam**
  JSF/Seam provides a *stateless* definition of the pageflow. The stateless model defines a set of named, logical outcomes of events (buttons and links on Web pages). Each event is bonded to an action listener method that must make decisions of the pageflow, since only they have access to the current state of the Web application.

- **jPDL**
  jPDL provides a *stateful* model that defines a set of transactions between a set of named, logical Web application states. It is possible to write action listener methods that are completely unaware of the flow of interactions because the flow of any user interaction can be expressed entirely in the jPDL pageflow definition.

JSF/Seam navigation rules are much simpler but the underlying Java code is more complex to understand. On the other hand, jPDL makes user interactions immediately understandable without looking to the JSP or Java code.

Stateful models are more constrained. For logical state there are a constrained set of possible transactions to other states. The stateless model is more suitable to relatively unconstrained navigation rules where the user decides which Web page to display next, not the application.

Like mentioned above, this approaches can help us for further work. Mainly for modeling the Model component of the MVC pattern for an automated generation of fetching data, e.g. from databases. The next section describes our approach of modeling Web application.

4 Description of our Approach

With all this theoretical background about the JSF framework and MDA, it is time to make something useful of it. This section describes our defined meta-model that defines the DSL as well as the process of generating a JSF Web application based on a given model that is an instance of the defined meta-model. Furthermore a prototype of a JSF Web application is introduced that demonstrates the functioning of our approach.

4.1 The Meta-Model

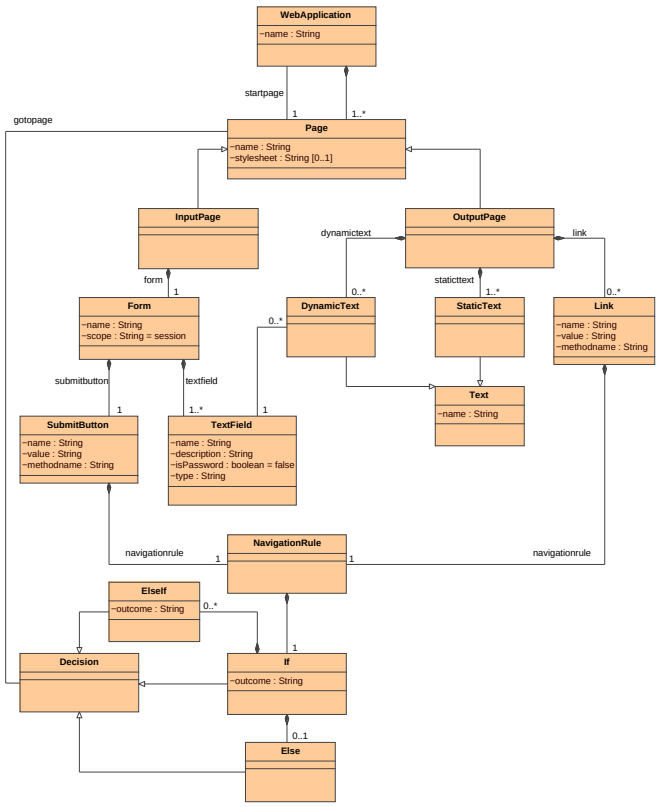

Figure 6: Metamodel

Figure 6 shows our defined meta-model illustrated by an UML class diagram. A certain Web application is an instance of this meta-model. Our meta-model provides the modeling of Web pages as well as the modeling of the pageflow. First we will describe the parts of our meta-model that are responsible for modeling the Web pages.

4.1.1 Modeling of Web pages

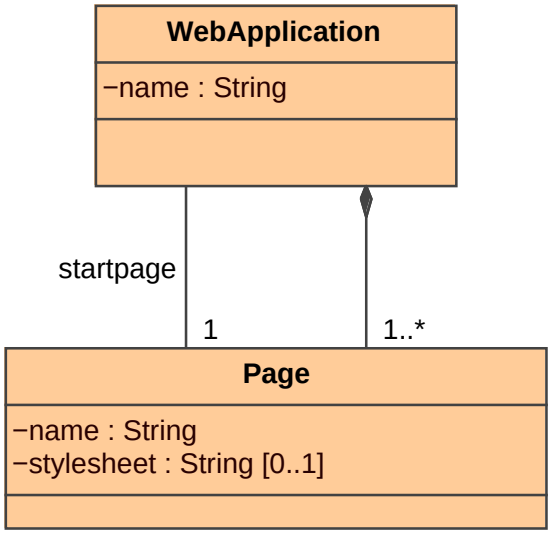

Figure 7: Relation between a `WebApplication` and `Pages`

Each `WebApplication` consists of:

- An attribute *name* of type `String` that specifies the name of the Web application.
- A *startpage* and
- one or more (**1..***) `Pages`.

The relationships between the classes `WebApplication` and `Page` is illustrated in Figure 7.

The class `Page` is responsible for Web pages characterized by the following attributes:

- A *name* of type `String`:
  The attribute *name* contains the name of the Web page as well as the title.
- A *stylesheet* of type `String`:
  The *stylesheet* attribute is an optional attribute that can contain the filename of an user-defined stylesheet for a certain page. If the *stylesheet* attribute is not given, the generator depicts a default stylesheet.

We divide Web pages into `InputPage`s and `OutputPage`s. This is depicted in Figure 8. An input-page contains components of user interfaces where the user can enter data. On the other hand an output-page is only responsible for displaying data. Both can contain one or more (**0..***) `Link`s that are responsible for the pageflow.

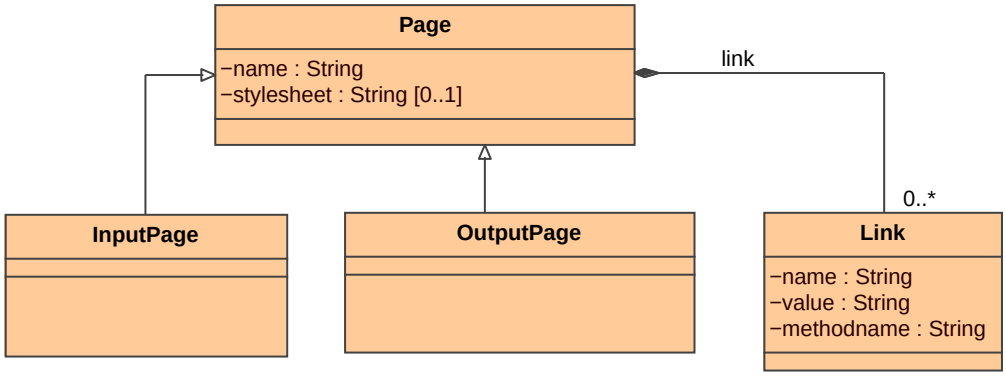

Figure 8: Division of `Page` into `InputPage` and `OutputPage`

The `Link` class contains the following attributes:

- A *name* attribute of type `String` that specifies the name of the link component,
- a *value* of type `String` that captures the link and

- a *methodname* of type `String` that contains the name of the method of a Java Bean that is responsible for delivering the outcomings of the given link.

Now we will take a closer look to input-pages. An `InputPage` consists of one (**1**) `Form`. A `Form` contains one or more (**1..***) `TextFields` and one (**1**) `SubmitButton`. These relationships are demonstrated in Figure 9.

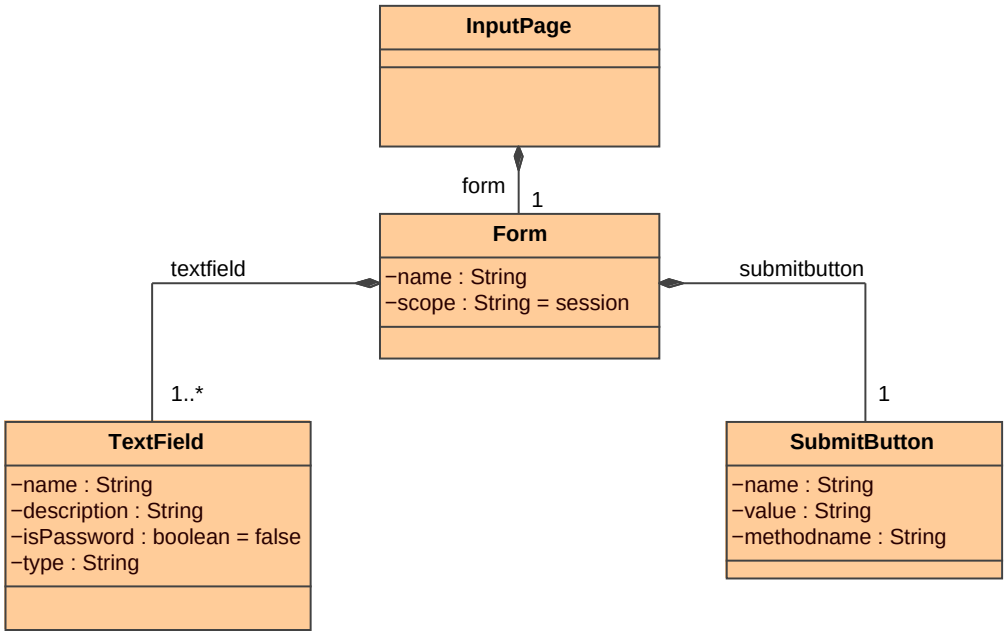

Figure 9: Composition of `InputPage`

A `Form` is described by the following attributes:

- A *name* of type `String` that indicates the name and
- A *scope* of type `String` that specifies the duration of validity of the entered data within the form. Valid values of the scope are: `application`, `session`, `request` and `page`.

A `TextField` consists of the following attributes:

- A *name* of type `String` that contains the name of the textfield,
- a *description* of type `String` that captures the textfield,
- an attribute *isPassword* of type `boolean` that indicates if the textfield is a password field or not and
- a *type* of type `String` that specifies the type of the value of the textfield. Valid values are: `int` and `String`.

A `SubmitButton` is characterized by the following attributes:

- A *name* of type `String` that contains the name of the button,
- a *value* of type `String` that specifies the caption of the button and
- a *methodname* of type `String` that qualifies the method of a Java Bean that is responsible for the outcome of the button.

Next, we will take a closer look to output-pages. Figure 10 demonstrates the constitution of an output-page. An `OutputPage` consists of `DynamicText` and `StaticText`. `DynamicText` and `StaticText` are composed to the class `Text` that consists of the attribute *name* of type `String` that specifies the name of the text component.

The `StaticText` class is specified by the *value* attribute of type `String` that contains the value of the static text. `DynamicText` fetches the value from data that was entered by the user in a textfield. Hence an association between `DynamicText` and `TextField` exists.

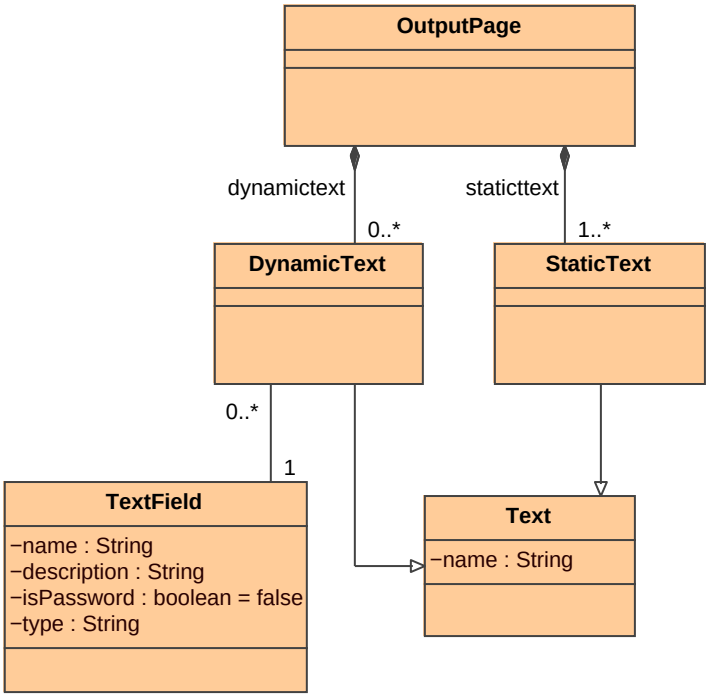

Figure 10: Composition of `OutputPage`

After modeling the Web pages of a Web application we can model the pageflow. The parts of our defined meta-model that are responsible for the modeling of the pageflow are described in the following section.

4.1.2 Modeling of the Pageflow

Due to the fact that only links and buttons are responsible for outcomings that define the pageflow we defined navigation rules that are associated with the `Link` and `SubmitButton` classes. These associations are shown in Figure 11.

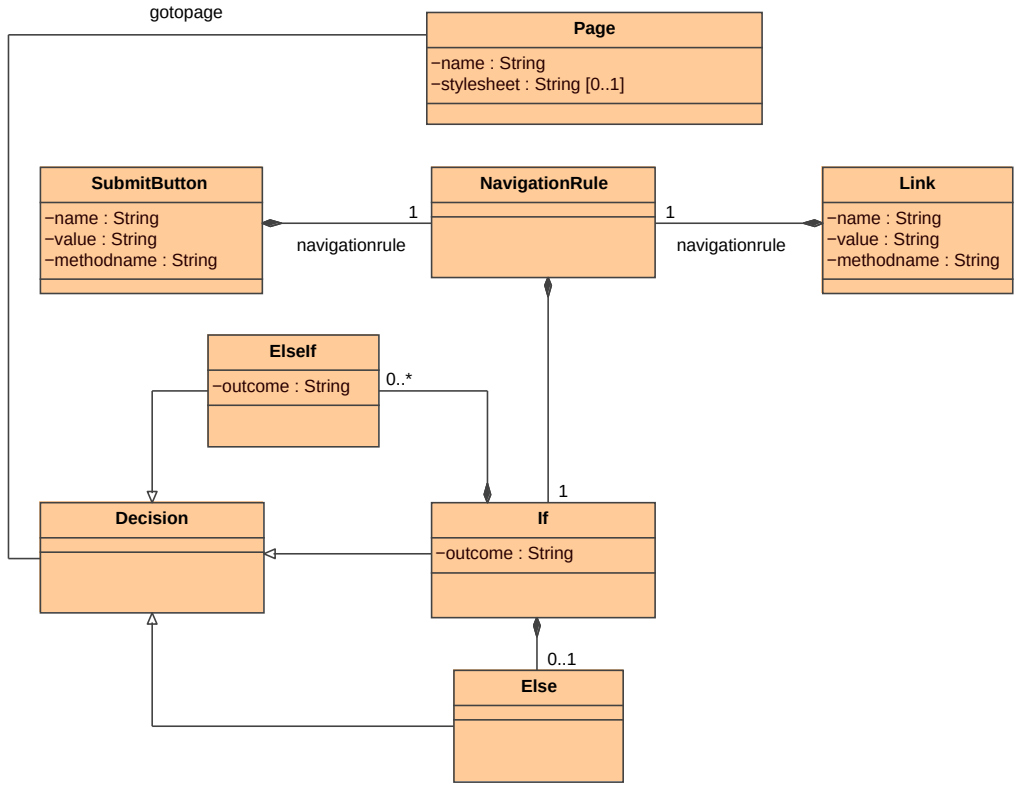

Figure 11: Modeling of the Pageflow

Each `SubmitButton` and each `Link` consists of one or more (**1..***) `NavigationRule`s. Due to the fact that we wanted to achieve a process-oriented modeling of the pageflow, our meta-model contains classes that enable the pageflow modeling through Java-like `IF-ELSE` statements. A `NavigationRule` consists of one (**1**) `If` statement. The `If` class is specified by the *outcome* attribute of type `String` that is responsible for the decision which Web page has to be

displayed next. An `If` can consist of one or more (**0..***) `ElseIf` statements. Each `ElseIf` statement consists of an *outcome* attribute that has the same signification as the *outcome* attribute for the `If` class. Furthermore an `If` statement can consist of an (**0..1**) `Else` statement. `If`, `ElseIf` and `Else` are composed to the `Decision` class that is associated with the `Page` class that defines the Web page that has to be displayed next.

Our defined meta-model provides possibilities to model the Web pages as well as the pageflow of a Web application. How a modeled Web application is generated based on our defined meta-model is described in the following section.

4.2 Code Generation with oAW

oAW provides a helpful plug-in for Eclipse ([13]) to generate a Web application based on a given model. Like mentioned in the Theory section, the oAW core consists of a number of features. The main part of the code generator builds the workflow that is described in the following section.

4.2.1 Workflow

The workflow builds the engine of the code generator. Like described in the Theory section, the workflow can consist of a number of components. Our workflow consists of seven components that are responsible to generate certain parts of the Web application. Furthermore our Workflow uses a property file that contains the name of the XMI file where the model is located as well as the path where the generated code should be placed. The name of the property file is specified by the `file` attribute within the *<property>* tag. The following listing depicts an example for a property file:

```
model = model.xmi
srcGenPath = src-gen
```

Listing 11: Workflow property file

The main components of our workflow to generate a Web application are:

- **XMI Reader** Component

  The **XMI Reader** component reads a given model that is specified by the `<modelFile>` tag and puts it into the `outputSlot`. Furthermore our defined meta-model must be passed to the XMI Reader component. This is done by the `<metaModelFile>` tag. Now the input model can be referenced by the *value* attribute specified within the `<outputSlot>` tag. The following listing demonstrates the definition of the XMI Reader component.

```
<component class="org.openarchitectureware.emf.XmiReader">
  <!-- Specify the Meta-Model -->
  <metaModelFile value="metamodel/MetaModel.ecore" />

  <!-- Specify the Model -->
  <modelFile value="${model}" />
```

```
  <!-- Load model into outputSlot -->
  <outputSlot value="model" />
</component>
```

Listing 12: The XMI Reader Component of the Workflow

- **Check** Component

  The second component of the workflow is the **Check** component that is used for model validation. The Check component needs the meta-model that is specified by the `<metaModel>` tags. The meta-model serves as basis for the model validation. Furthermore the check file has to be specified. This is done by the `<checkFile>` tag within the Check component. The Check component requires the given model as input that has to be validated. This is specified through the `<expression>` tag. The *value* attribute accords to the value of the `<outputSlot>` tag of the XMI Reader component. The following listing demonstrates the declaration of the Check component within our defined Workflow.

```
<!-- COMPONENT: CHECK -->
<component id="checker"
    class="org.openarchitectureware.check.CheckComponent">

    <!-- Specify the Meta-Model -->
    <metaModel class="oaw.type.emf.EmfMetaModel"
               metaModelPackage="org.eclipse.emf.ecore.EcorePackage">
      <metaModelFile value="metamodel/MetaModel.ecore" />
    </metaModel>

    <!-- fully qualified name of the check-file (.chk) -->
    <checkFile value="check::JSFChecks"/>

    <!-- check the model against constraints -->
    <!-- defined in the check-file           -->
    <emfAllChildrenSlot value="model"/>
</component>
```

Listing 13: The Check Component of the Workflow

- **Generator** Components

  The **Generator** components are responsible for the code generation. Again, the meta-model must be specified within the `<metaModel>` tag. The `<expand>` tag refers to a template that contains the transformation

rules that are responsible for the code generation. The template files are described later in this section. The *value* attribute of the `<expand>` tag refers to the root of the template. Furthermore the path where the generated code should be placed is specified within the `<outlet>` tag. Finally two code beautifiers are specified within the `<beautifier>` tags. Beautifiers beautify the generated code for improving the clarity. oAW supports two code beautifiers: a Java- and a XML-code beautifier. Due to the fact that we have to generate Java- and XML-code, both beautifiers are included within the particular generator component. The following listings describe our defined components that are responsible for generating Web pages, Java Beans and the XML file that contains information about the pageflow.

– **Component to generate Web pages**

```
<!-- COMPONENT to generate Web pages  -->
<component
 id="generator"
 class="org.openarchitectureware.xpand2.Generator">

 <!-- Specify the Meta-Model -->
 <metaModel
  class="org.openarchitectureware.type.emf.EmfMetaModel">
  <metaModelFile value="metamodel/MetaModel.ecore" />
 </metaModel>

 <!-- Using Template for Code Generation -->
 <expand value="templates::generatePages::Root FOR model"/>

 <!-- Specify Output Path -->
 <outlet path="${srcGenPath}"/>
</component>
```

Listing 14: The Generator Component for Web pages

– **Component to generate Java Beans**

```
<!-- COMPONENT to generate Java Beans  -->
<component
 id="BeanGenerator"
 class="org.openarchitectureware.xpand2.Generator">

 <!-- Specify the Meta-Model -->
 <metaModel
  class="org.openarchitectureware.type.emf.EmfMetaModel">
  <metaModelFile value="metamodel/MetaModel.ecore"/>
 </metaModel>
```

```
 <!-- Using Template for Code Generation -->
 <expand value="templates::generateBeans::Root FOR model"/>

 <!-- Specify Output Path -->
 <outlet path="${srcGenPath}"/>

 <!-- Code Beautifiers -->
 <beautifier
  class="org.openarchitectureware.xpand2.output.JavaBeautifier"/>
</component>
```

Listing 15: The Generator Component for Java Beans

– **Component to generate the Pageflow**

```
<!-- COMPONENT to generate the Pageflow -->
<component
 id="PageFlowGenerator"
 class="org.openarchitectureware.xpand2.Generator">

 <!-- Specify the Meta-Model -->
 <metaModel
  class="org.openarchitectureware.type.emf.EmfMetaModel">
  <metaModelFile value="metamodel/MetaModel.ecore"/>
 </metaModel>

 <!-- Using Template for Code Generation -->
 <expand
  value="templates::generateFacesConfigXML::Root FOR model"/>

 <!-- Specify Output Path -->
 <outlet path="${srcGenPath}"/>

 <!-- Code Beautifier -->
 <beautifier
  class="org.openarchitectureware.xpand2.output.XmlBeautifier"/>
</component>
```

Listing 16: The Generator Component for the Pageflow information

To validate a given model, the **Check** component of the Workflow is used. The definition of the check constraints within this project is described in the next section.

4.2.2 Check - Model Validation

As mentioned in the Theory section, oAW provides a feature for model validation that is called **Check**. All defined constraints must be located in a file with the prefix *.chk*. The following listing shows a snippet of the check file used in the course of this project:

```
import MetaModel;

context MetaModel::WebApplication
        ERROR 'a WebApplication must have a name!':
    this.name!=null;

context MetaModel::WebApplication
        ERROR 'invalid name for WebApplication!':
    name.length>0;
```

Listing 17: The *chk* file

This snippet checks if a name is given for `WebApplication` as well as that the name is not empty in the given model. Furthermore checks are done if values are set correctly within the given model, e.g. the *scope* attribute of a `Form` can only contain the values `application`, `session`, `request` or `page`.

After the given model is validated the process of code generation is accomplished. The templates that are used as the basis for generating code are described in the next section.

4.2.3 Xpand2 - Templates

oAW provides a special language called Xpand2 that is used in templates to control the code generation. Templates must resist in files with the prefix *.xpt*. The tag brackets («,») are characters introduced by oAW to define control sequences like `FOREACH` or `IF` within the template files to control the generation of code. The following listings describe the defined template files within this project to generate the Web pages, the Java Beans as well as the XML file that contains the pageflow information.

- **Template to generate Web pages**

  The following listing shows how Web pages are generated with an oAW template file:

```
<<FOREACH inputpage AS p>>

<<FILE p.name+".jsp">>
     <%@ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>
     <%@ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>
     <html>
      <head>
      ...
      </head>
      <body>
       <f:view>
         <<EXPAND InputJSP FOR p>>
       </f:view>
      </body>
     </html>
<<ENDFILE>>

<<ENDFOREACH>>
```

Listing 18: Template to generate Web pages

For each instance of the class `InputPage` we create a JSP file that is equivalent to the name of the given input-page. First the tag libraries for JSP are generated through the `<taglib>` tags. Afterwards the header information including the stylesheet is generated. Then the user interface components of an input-page are generated. That is achieved by calling the subrouting `InputJSP`. This subroutine is responsible to generate JSP code for all instances of the `Form`, `TextField`, `SubmitButton` and `Link` classes. The following listing shows the subroutine `InputJSP`.

```
<<DEFINE InputJSP FOR MetaModel::InputPage>>
    <h:form id="<<form.name>>">
    <<FOREACH form.textfield AS tf>>
       <!-- TEXTFIELD -->
       <h:outputText id="<<tf.description>>"
                     value="<<tf.description>>" />
       <<IF tf.isPassword>>
          <h:inputSecret id="<<tf.name>>"
                         value="#{Generated<<name.toFirstUpper()>>Bean.
                                  <<tf.name>>}" />
```

```
        <<ELSE>>
           <h:inputText id="<<tf.name>>"
                        value="#{Generated<<name.toFirstUpper()>>Bean.
                                 <<tf.name>>}" />
        <<ENDIF>>
    <<ENDFOREACH>>
    <!-- SUBMITBUTTON -->
    <h:commandButton id="<<form.submitbutton.name>>"
                     action="#{Generated<<name.toFirstUpper()>>Bean.
                              <<form.submitbutton.methodname>>}"
                     value="<<form.submitbutton.value>>" />
    </h:form>

    <!-- LINKS -->
    <<FOREACH link AS l>>
    <h:form>
    <h:commandLink id="<<l.name>>"
       action="#{Generated<<name.toFirstUpper()>>Bean.
                 <<l.methodname>>}">
       <h:outputText value="<<l.value>>" />
    </h:commandLink>
    </h:form>
    <<ENDFOREACH>>

<<ENDDEFINE>>
```

Listing 19: Generating user interface components for `InputPage`

First we generate the JSF tags for each textfield. If the attribute *isPasswort* of the `TextField` is set to `true` a password field is generated otherwise a textfield. All Textfields get an `id` that is derived from the *name* attribute. Next the JSF tag for the `SubmitButton` is generated. The button gets an `id`, a `value` and an `action`. The `action` is linked to a method of a Java Bean that is executed when the user pressed the button and returns a certain outcome. The name of the method is given by the *methodname* attribute of the `SubmitButton` class. Finally the JSF tags for links are generated. Similar to buttons the `action` of a link is linked to a method of a Java Bean that is responsible for the outcome of the link, hence for the pageflow.

After all input-pages are generated the generation of the output-pages can start. The generation of output-pages is similar to the approach of generating input-pages. The only difference is that the subroutine for generating the components of an output-page. The subroutine for output-pages is shown in the following listing:

```
<<DEFINE OutputJSP FOR MetaModel::InputPage>>

    <!-- STATIC OUTPUT TEXT -->
    <<FOREACH staticoutputtext AS t>>
    <h:outputText id="<<t.name>>"
                  value="<<t.value>>" /><br>
    <<ENDFOREACH>>

    <!-- DYNAMIC OUTPUT TEXT -->
    <<FOREACH dynamicoutputtext AS t>>
    <h:outputText id="<<t.name>>"
                  value="<<t.textfield>>" /><br>
    <<ENDFOREACH>>

    <!-- LINKS -->
    <<FOREACH link AS l>>
    <h:form>
    <h:commandLink id="<<l.name>>"
       action="#{Generated<<name.toFirstUpper()>>Bean.
                 <<l.methodname>>}">
       <h:outputText value="<<l.value>>" />
    </h:commandLink>
    </h:form>
    <<ENDFOREACH>>
<<ENDDEFINE>>
```

Listing 20: Generate user interface components for `OutputPage`

The subroutine `OutputJSP` generates tags for each `staticoutputtext` and for each `dynamicoutputtext`. The `id`s are the name of the labels. The difference between static and dynamic text is that the value of a static text contains a String and a dynamic text is linked to the value of a textfield of an input-page. Afterwards the JSF tags for links are generated similar to the subroutine for input-pages.

- **Template to generate Java Beans**

  Java Beans are generated for each input-page as well as for each output-page that contain at least one link. Furthermore Java classes are generated for each output-page containing links. The following listing shows a snippet of the template file:

```
<<FOREACH inputpage AS p>>
  <<FILE "Generated"+p.name.toFirstUpper()+"Bean.java">>
  public class Generated<<p.name.toFirstUpper()>>Bean {
```

```
  public Generated<<p.name.toFirstUpper()>>Bean() {}

  <<EXPAND FormBeans FOR p.form>>

  ...

<<ENDFOREACH>>
```

Listing 21: Template to generate Java Beans

For each instance of the class `InputPage` we create a Java class that gets the suffix `Generated` and the prefix `Bean`. This should help the programmer to distinguish between generated classes and manually written classes. First we create the constructor that is equivalent to the class name. Afterwards the subroutine `FormBeans` declares a variable for each textfield within the form and generates get- and set-methods for each variable.

The following listing describes the generation of methods that are linked to buttons that are responsible for returning the outcomings:

```
  <<IF p.form.submitbutton.navigationrule!=null>>

  public String <<p.form.submitbutton.methodname>>() {

    <<p.name.toFirstUpper()>>Util util<<p.name.toFirstUpper()>> =
                    new <<p.name.toFirstUpper()>>Util();
    String strOutcome = util<<p.name.toFirstUpper()>>.
                          <<p.form.submitbutton.methodname>>(
    <<FOREACH p.form.textfield.name AS tf SEPARATOR ','>>
    this.<<tf>>
    <<ENDFOREACH>>);

  ...
```

Listing 22: Generating methods for each `SubmitButton`

We achieve that the programmer has to create for each input-page a Java class that accords to the name of the input-page with the prefix `Util`. Furthermore the programmer must implement a method that matches the value of the *methodname* attribute of the `SubmitButton` class. This method gets the values from the textfields as parameters and must return the defined outcomings in the given model. Therefor

we generate code that compares the returned String from the manually written method. This is shown in the following listing:

```
<<IF p.form.submitbutton.navigationrule.if!=null>>
  if(strOutcome.equals("
     <<p.form.submitbutton.navigationrule.if.outcome>>")) {
     return "<<p.form.submitbutton.navigationrule.if.outcome>>";
  }
  <<IF p.form.submitbutton.navigationrule.if.elseif.size>0>>
  <<FOREACH p.form.submitbutton.navigationrule.if.elseif AS ei>>
  else if(strOutcome.equals("<<ei.outcome>>")) {
          return "<<ei.outcome>>";
  }
  <<ENDFOREACH>>
  <<ENDIF>>

  <<IF p.form.submitbutton.navigationrule.if.else!=null>>
  else {
    return "*";
  }
  <<ELSE>>
    return null;
  <<ENDIF>>
<<ELSE>>
   return null;
<<ENDIF>>
```

Listing 23: Generate outcomings for each method

To achieve the correct outcomings we have to define `IF-ELSE` statements that are derived from the process-oriented definition of the page-flow within the model. First, code is generated for the `IF` branch. Afterwards is code generated for each `ELSEIF` branch. Finally the `ELSE` branch is generated if it is given. Otherwise the method returns `null` within the else branch. With this approach we achieve that the programmer is responsible for returning the correct outcomings through manually written methods. Otherwise the wildcard `*` or `null` is returned that signifies that the current page is displayed again.

Thereby we solve the problem of dealing with generated and manually written code because the programmer is responsible to validate the entered data in the textfields, e.g. by accessing a database. Moreover the manually written code does not become overwritten in a subsequent generator run because it exists outside the generated code.

To separate generated from manually written code we chose the proven approach to call handwritten Java classes by generated code to avoid severe effects in case of interventions in the model. Changes in the templates lead to changes in the generated code. This can further leads to the problem that the developer must change the manually written classes that extend generated classes. To avoid this problem and decrease the coupling of the generated and the manually written code we propose the following solution: generated code calls methods of handwritten classes.

- **Template to generate Pageflow information**

  The template to generate information about the pageflow generates an XML file that serves as input for a Java Servlet that is provided by the JSF framework. This Java Servlet is responsible for controlling the pageflow based on the outcomings of buttons and links. The following listing shows the template that generates the XML file:

```
<<FILE "faces-config.xml">>

...

<<FOREACH inputpage AS p>>
  <<IF p.form.submitbutton.navigationrule!=null>>
    <navigation-rule>
      <from-view-id>/pages/<<p.name>>.jsp</from-view-id>
       <<EXPAND NavigationCase FOR
           p.form.submitbutton.navigationrule>>
       <<IF p.link.size>0>>
         <<FOREACH p.link AS l>>
           <<EXPAND NavigationCase FOR l.navigationrule>>
         <<ENDFOREACH>>
       <<ENDIF>>
    </navigation-rule>
  <<ENDIF>>
<<ENDFOREACH>>

<<FOREACH outputpage AS p>>
  <<IF p.link.size>0>>
    <navigation-rule>
      <from-view-id>/pages/<<p.name>>.jsp</from-view-id>
      <<FOREACH p.link AS l>>
        <<EXPAND NavigationCase FOR l.navigationrule>>
      <<ENDFOREACH>>
    </navigation-rule>
  <<ENDIF>>
```

```
  <<ENDFOREACH>>
```

Listing 24: Template to generate Pageflow information

First we generate information about the pageflow for each `InputPage`. Information about the outcomings delivered from the method of the `SubmitButton` is generated. Therefor the subroutine `NavigationCase` is called. Aftwards information about the outcomings delivered from the `Link`s is generated. The `NavigationCase` is called for each link. After generating the pageflow information for each input-page the pageflow information for each `OutputPage` is generated. This is similar to `InputPage`. The following listing shows the subroutine `NavigationCase`:

```
<<DEFINE NavigationCase FOR MetaModel::NavigationRule>>
    <navigation-case>
       <from-outcome><<if.outcome>></from-outcome>
       <to-view-id>/pages/<<if.gotopage.name>>.jsp</to-view-id>
    </navigation-case>

    <<IF if.elseif.size > 0>>
    <<FOREACH if.elseif AS e>>
    <navigation-case>
       <from-outcome><<e.outcome>></from-outcome>
       <to-view-id>/pages/<<e.gotopage.name>>.jsp</to-view-id>
    </navigation-case>
    <<ENDFOREACH>>
    <<ENDIF>>

    <<IF if.else!=null>>
    <navigation-case>
       <from-outcome>*</from-outcome>
       <to-view-id>/pages/<<if.else.gotopage.name>>.jsp</to-view-id>
    </navigation-case>
    <<ENDIF>>

<<ENDDEFINE>>
```

Listing 25: Generating navigation-cases

First we generate the navigation-cases for the `IF` branch of the process-oriented defined pageflow. A navigation-case consists of an outcome and a Web page that has to be displayed by the certain outcome. Afterwards the navigation-cases for each `ELSEIF` branch are generated. Finally the `ELSE` branch outcome is generated.

After generating Web pages, Java Beans and information about the pageflow an Apache Ant file is generated that facilitates the deployment of the generated JSF Web application on the Apache Tomcat Webserver. The generation of the Ant file is described next.

- **Template to generate the Apache Ant File**

  This template is used to generate the *build.xml* file that is executed by the developer using Apache Ant. Apache Ant is used to ease the deployment of the generated Web application on the Apache Tomcat Web server. Listing 26 shows a snippet with the main components of the template that generates the *build.xml* file.

```
<<DEFINE Root FOR MetaModel::WebApplication>>
<<FILE name+"\\ant\\build.xml">>
<project name="jsf" basedir="../" default="deploy">
    <property file="ant/build.properties" />

    ...

    <!-- Deploy Web application -->
    <target name="deploy" depends="war">
       <copy file="${build.dir}/${project.distname}.war"
             todir="${tomcat.dir}" />
    </target>

    <!-- Create WAR file -->
    <target name="war" depends="build">
      <mkdir dir="${build.dir}" />
      <war basedir="${webroot.dir}"
        warfile="${build.dir}/${project.distname}.war"
        webxml="${webinf.dir}/web.xml">
        <exclude name="WEB-INF/${build.dir}/**"/>
        <exclude name="WEB-INF/src/**"/>
        <exclude name="WEB-INF/web.xml"/>
       </war>
    </target>

    <!-- Build entire project -->
    <target name="build" depends="prepare,compile"/>

    <target name="prepare">
        <tstamp/>
    </target>

    <!-- Compile Java files -->
    <mkdir dir="${webinf.dir}/classes" />
    <target name="compile" depends="prepare,resources">
```

```
        <javac srcdir="JavaSource" destdir="${webinf.dir}/classes">
            <classpath refid="compile.classpath"/>
        </javac>
    </target>

    ...

</project>
<<ENDFILE>>
<<ENDDEFINE>>
```

Listing 26: Generating the Apache Ant file *build.xml*

The *build.xml* file is located in the `ant` directory of the generated Web application. Like mentioned in the Theory section, an Apache Ant file consists of multiple targets. Our generated Ant file has a `deploy` target that is set to default in the attributes of the `<project>` tag. The `deploy` target depends on the `war` target. The `war` target is used to create a *war* file that is copied to the `webapps` directory of the Apache Tomcat Web server within the `deploy` target. The `war` target depends on the `build` target that is used to compile the Java source files. The compilation of the Java files results in errors if the developer does not implement the Java classes in which the called methods rely that are called by the generated Java Beans. After compiling the Java sources the *war* file is created and copied to the `webapps` directory of the Apache Tomcat Web server. Our generated *build.xml* file uses a properties file that is specified within the `<property>` tag at the top of the *build.xml* file. The property file contains all needed directories of the *build.xml* file, e.g. `${build.dir}`.

After the JSF Web application is generated the developer can deploy the generated Web application using the Ant file. Ant checks if the developer implemented all needed classes and methods called by the generated Java Beans. By starting the Apache Tomcat Web server the *war* file is extracted and the whole JSF Web application relies in the `webapps` directory. To demonstrate the functioning of our approach a prototype JSF Web application was generated that is described in the following section.

4.3 A Prototype for Visualizing the Mode of Operation

In this section we want to introduce a prototype to demonstrate the functioning of our approach. The prototype features a JSF Web application that can be deployed on the Apache Tomcat Web server. Our prototype pictures a Web application that consists of a registration feature. To login to a certain Web application, a user has to register first. A model-driven development of such a registration Web application is demonstrated by our prototype.

First, the Web application that should be generated must be described in a model. Figure 12 shows the UML class diagram of a Web page that contains a form for user registration.

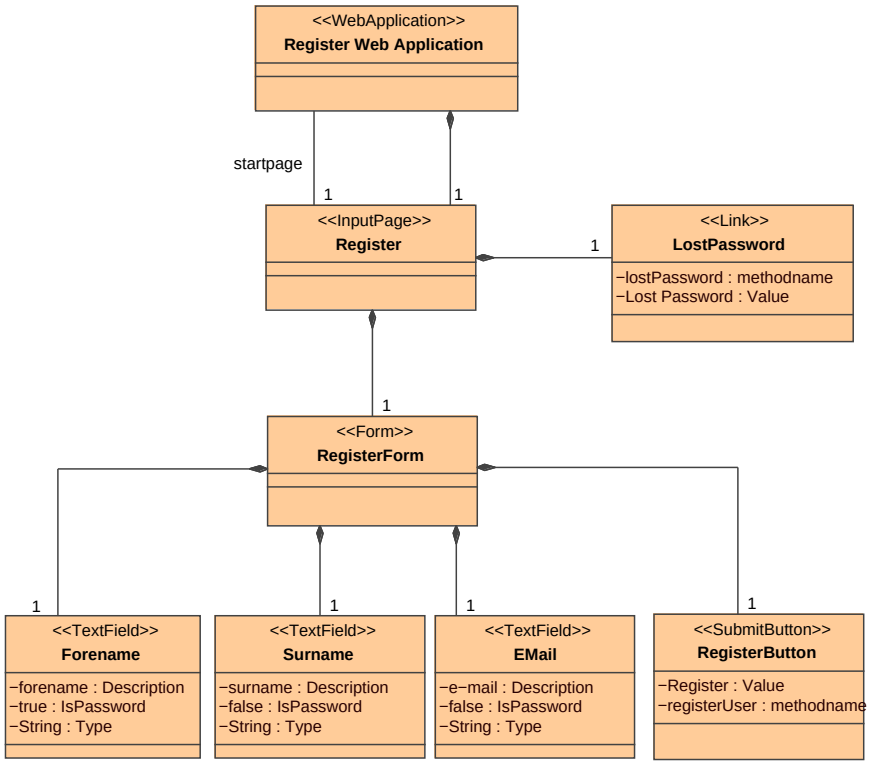

Figure 12: Model of a Web page

The class `Register Web Application` has the stereotype `«WebApplication»` that was introduced by the meta-model. The class `Register` with the stereotype `«InputPage»` is denoted as an input-page as well as the start-page of the Web application. The page `Register` consists of one (**1**) `Form` and one (**1**) `Link`. The `RegisterForm` consists of one (**1**) `Forename` textfield, one (**1**) `Surname` textfield, one (**1**) `EMail` textfield and one (**1**) `RegisterButton`. The attribute *methodname* of `RegisterButton` is set to `registerUser` that denotes the methodname of the Java Bean that has to be executed if the button is pressed. The `LostPassword` link utilizes the method `lostPassword`. These methods are described later in this section.

Analog to the Register Web page all Web pages of the Web application must be modeled. Our prototype consists of the following Web pages beside the Register Web page:

- **thanks.jsp**

  This page is displayed after the registration was successful. The Web page **thanks.jsp** is an output-page that contains a static text as well as a link.

- **failed.jsp**

  This page is displayed when the registration failed, e.g. the entered e-mail address already exists in the database. **failed.jsp** is an output-page with a static text and a link like the Web page **thanks.jsp**.

- **lost.jsp**

  The Web page **lost.jsp** is an input-page and contains a form where the user can enter an e-mail address to which the password should be sent.

- **login.jsp**

  The Web page **login.jsp** is an input-page that contains textfields where the user can enter the e-mail address and the corresponding password to login a Web application.

A graphical representation of the pageflow of our prototype is shown in Figure 13.

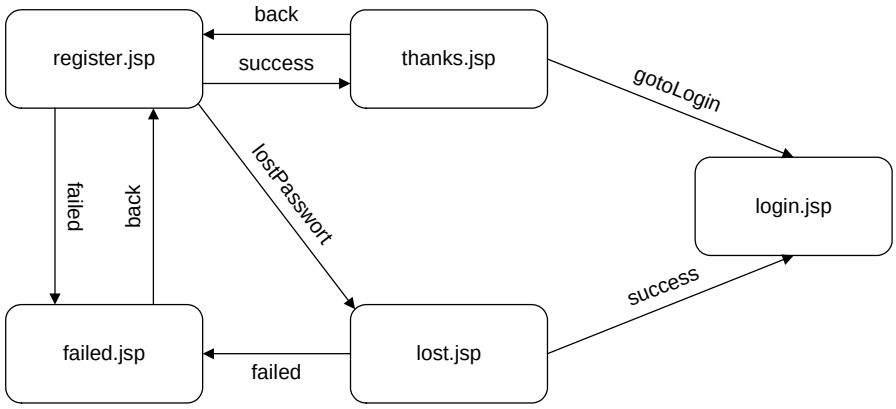

Figure 13: Pageflow of our Prototype

The subsequent Web pages of **register.jsp** dependent on the outcome of the `RegisterButton` and the `LostPassword` link. If the outcome of the `RegisterButton` is *success*, **thanks.jsp** is displayed next. Otherwise, if the outcome is *failed* the **failed.jsp** Web page is displayed. If the user clicks the `LostPassword` link that delivers the outcome *lostPassword*, **lost.jsp** is displayed.

The output-page **thanks.jsp** consists of two links with the outcomings *gotoLogin* and *back*. If the user clicks the `Goto Login` link, the outcome is *gotoLogin* and the user is forwarded to **login.jsp** Web page. Else if the user clicks the `Back` link that delivers the outcome *back*, the **register.jsp** Web page is displayed next.

The output-page **failed.jsp** consists only of the `Back` link that delivers the outcome *back*. The **register.jsp** Web page is displayed next.

The input-page **login.jsp** does not deliver outcomings in our prototype because it acts as an interface to an Web application where the user has to login first before using the functionality of this Web application.

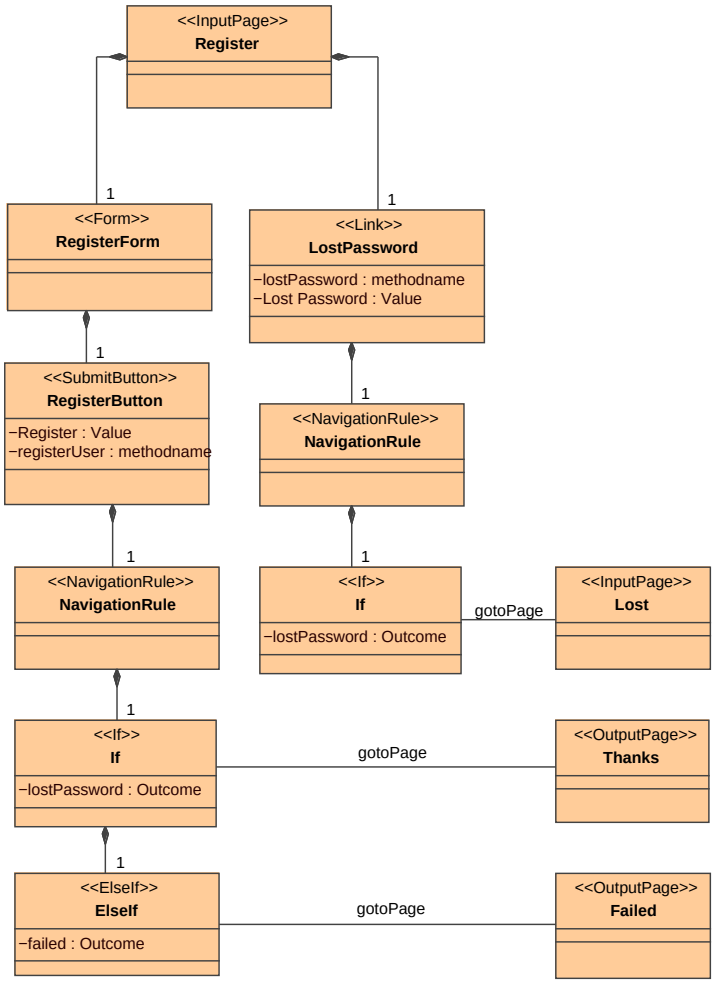

Figure 14: Modeling of Pageflow for the Register Web page

Figure 14 demonstrates the modeling of the pageflow for the Registration Web page. Like mentioned before, the Web page `Register` consists of a `RegisterForm` with a `RegisterButton` and of a `LostPassword` link. Navigation rules are assigned to buttons and links. The `RegisterButton` contains one (**1**) `NavigationRule` that consists of one (**1**) `If` branch. The modeled `If` branch defines if the `Outcome` is *success* than `gotoPage Thanks`. The `ElseIf` branch defines if the `Outcome` is *failed* than `gotoPage Failed`.

The pageflow modeling for a `Link` is analog to the modeling of the pageflow for a `SubmitButton`. The `LostPassword` link consists of one (**1**) `NavigationRule` that consists of one (**1**) `If` branch. The modeled `If` branch defines if the `Outcome` is *lostPassword* than `gotoPage LostPassword`.

The modeling of the pageflow for the Web pages `thanks.jsp`, `failed.jsp`, `lostPassword.jsp` and `login.jsp` is analog to the pageflow model of the `register.jsp` Web page demonstrated in Figure 14.

After modeling the Web application we can start to generate the Web application. The following section shows the generated Web pages, Java Beans and pageflow information generated by our created templates.

4.3.1 The Generated Prototype Web Application

Each generated Web application contains the following directory structure.

```
/<WebApplicationName>
 /ant
    build.xml
 /JavaSource
 /WebContent
    /pages
        jsf-impl.jar
        jsf-api.jar
    /lib
    /WEB-INF
        faces-config.xml
        web.xml
```

Listing 27: Generated directory structure

Each Web application consists of three directories:

- `ant`
  The `ant` directory contains an Ant build file named *build.xml*. This file is responsible for compiling the Java classes as well as to generate a *war* file that can be deployed on an Apache Tomcat Web server.

- `JavaSource`
  The `JavaSource` directory contains all generated Java Beans. Java Beans are generated for each input-page as well as for each output-page that contains at least one link.

- `WebContent`
  The `WebContent` directory contains of the `pages` directory that contains all generated Web pages. Moreover it contains a `lib` directory that holds Java libraries (*jar* files required by the JSF Web pplication. The `lib` directory contains the Java library files `jsf-impl.jar` and `jsf-api.jar`. These files are needed by every JSF Web application because they contain Java classes that are needed by the JSF framework, e.g. the `FacesServlet` Servlet that controls the pageflow. Furthermore the `WebContent` directory consists the `WEB-INF` directory that contains configuration files for the JSF framework and the Apache Tomcat Web server. The *faces-config.xml* file contains information about the pageflow of the generated Web application as well as information about all Java Beans that are used within the Web application. *faces-config.xml* is utilized as input for the `FacesServlet` of the JSF framework that controls the pageflow. The *web.xml* is a configuration file for the Web application that runs on the Apache Tomcat Web server. This file is used to include the `FacesServlet` in the JSF Web application.

4.3.1.1 Generated Web Pages

The following listing shows the generated **register.jsp** Web page.

```
<%@ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>
<%@ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>
<html>
<head> ... </head>
<body>
  <f:view>
    <h:form id="registerForm">
       <h:outputText id="Forename" value="Forename" />
       <h:inputText id="forename"
          value="#{GeneratedRegisterBean.forename}" />
       <h:outputText id="Surname" value="Surname" />
       <h:inputText id="surname"
          value="#{GeneratedRegisterBean.surname}" />
       <h:outputText id="E-Mail" value="E-Mail" />
       <h:inputText id="email"
          value="#{GeneratedRegisterBean.email}" />
       <h:commandButton id="btnRegister"
                        action="#{GeneratedRegisterBean.registerUser}"
                        value="Register" />
    </h:form>
    <h:form>
       <h:commandLink id="lostPassword"
             action="#{GeneratedRegisterBean.lostPassword}">
          <h:outputText value="Lost Password?" />
```

```
        </h:commandLink>
    </h:form>
  </f:view>
</body>
</html>
```

Listing 28: Generated register.jsp

The **register.jsp** Web page contains a form with textfields and a button as well as a link. All user interface components are within the `<f:view>` tag. Buttons and links are bound to methods of a Java Bean that is generated for the certain Web page. Those methods are executed if the user presses the button or clicks the link. Furthermore, those methods are responsible for the outcomings of the Web page. The outcome builds the basis for the decision which Web page has to be displayed next. The graphical appearance of **register.jsp** is shown in Figure 15.

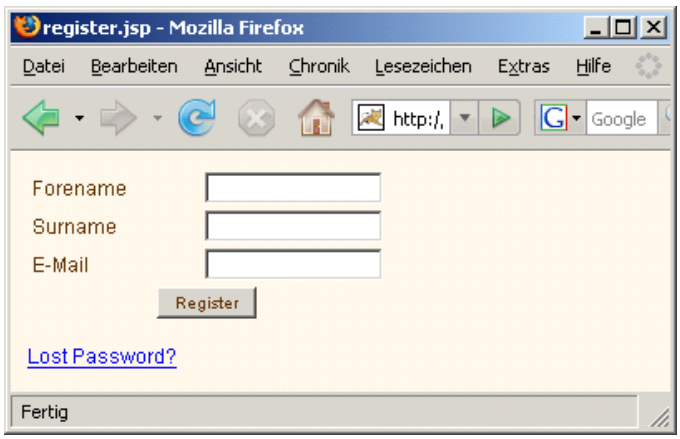

Figure 15: register.jsp

All Web pages are generated analog to the **register.jsp** Web page. The following figures demonstrate the graphical appearance of the other generated Web pages of our prototype.

Figure 16 shows the **thanks.jsp** Web page that is displayed if the **register.jsp** Web page delivers the outcome `success`. **thanks.jsp** consists of a static output text that displays the string `Thank you!`. **thanks.jsp** consists

also of a `back` link that is used to display **register.jsp** again. Furthermore **thanks.jsp** has a `login` link that is responsible to display the **login.jsp** Web page.

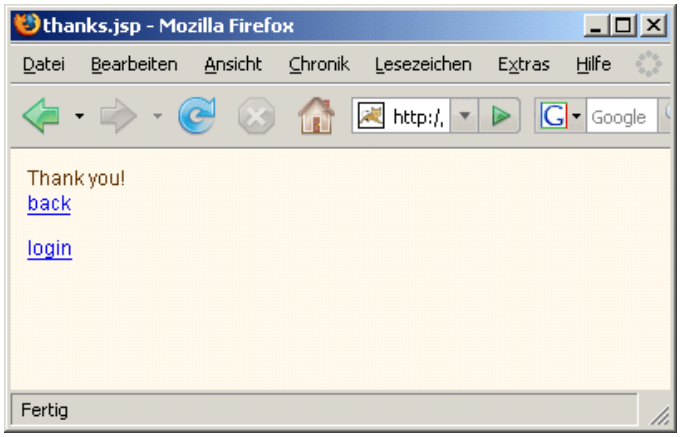

Figure 16: thanks.jsp

Figure 17 shows the **failed.jsp** Web page that is displayed if the **register.jsp** Web page delivers the outcome `failed`. **failed.jsp** consists of a static output text that displays the string `Registration failed!`. **failed.jsp** consists also of a `back` link that is used to display **register.jsp** again.

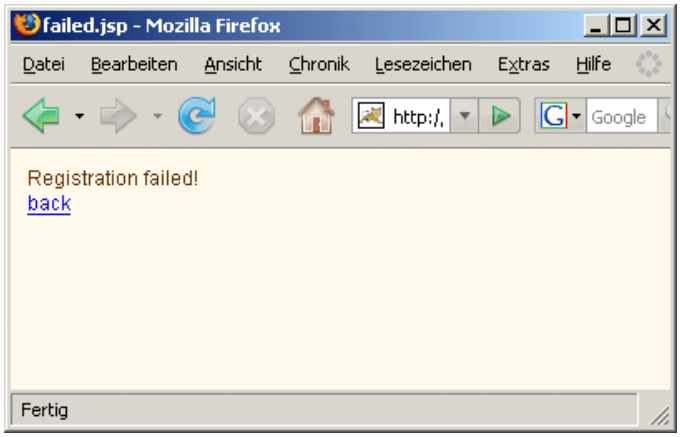

Figure 17: failed.jsp

Figure 18 shows the **lost.jsp** Web page that is displayed if the user clicks the `Lost Password` link of the **register.jsp** Web page. **lost.jsp** is an input-page that contains a form with a textfield and a button. The textfield is used for the user to enter an e-mail address if the user has forgotten the password. The method of the Java Bean bounded to the button is responsible to check the entered e-mail address and deliver an outcome. Depending on the outcome the **login.jsp** or **failed.jsp** Web page is displayed next.

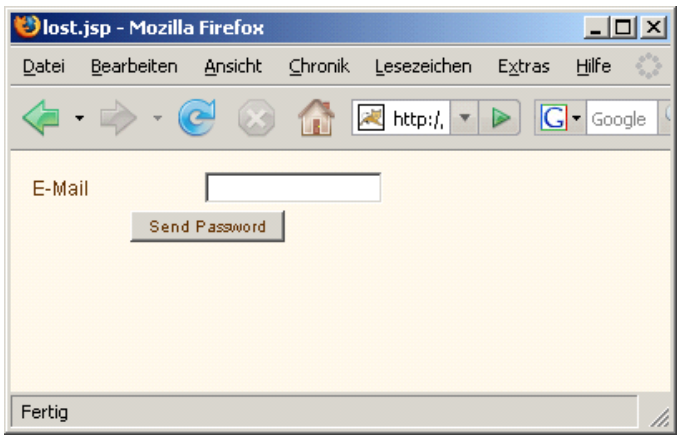

Figure 18: lost.jsp

Figure 19 shows the **login.jsp** Web page that contains a form where the user can enter the e-mail address and the password.

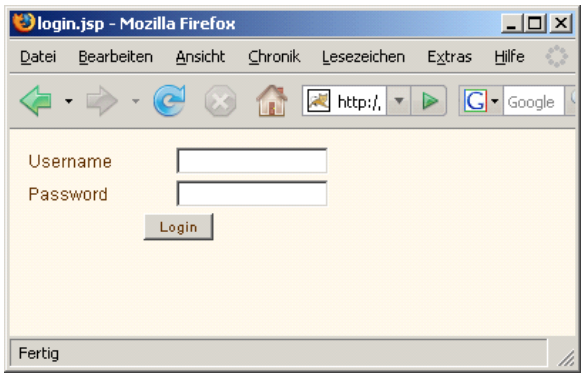

Figure 19: login.jsp

The following section depicts the generated Java Beans of our prototype that are responsible to store the entered values in textfields as well as for the delivery of the outcome of a certain Web page.

4.3.1.2 Generated Java Beans

Java Beans are generated for each input-page as well as for each output-page that contains links. Java Beans are responsible for the Model of the MVC pattern. Thus, Java Beans are used to fetch the by the Web application needed data, e.g. accessing a database. Due to the fact that we only generate Web pages as well as information about the pageflow, the developer is responsible for fetching data. Hence we have to generate code within the Java Beans that calls a method of a Java class that has to be implemented by the developer.

The following source code listing shows the generated Java Bean for the **register.jsp** Web page.

```
public class GeneratedRegisterBean {
    private String forename;
    private String surname;
    private String email;

    public GeneratedRegisterBean() {}

    public void setForename(String forename) {
        this.forename = forename;
    }

    public String getForename() {
        return this.forename;
    }

    public void setSurname(String surname) {
        this.surname = surname;
    }

    public String getSurname() {
        return this.surname;
    }

    public void setEmail(String email) {
        this.email = email;
    }

    public String getEmail() {
        return this.email;
```

```
    }

    public String registerUser() {
        RegisterUtil utilRegister = new RegisterUtil();
        String strOutcome = utilRegister.registerUser(this.forename,
                this.surname, this.email);
        if (strOutcome.equals("success")) {
            return "success";
        } else if (strOutcome.equals("failed")) {
            return "failed";
        }
        return null;
    }

    public String lostPassword() {
        RegisterUtil utilRegister = new RegisterUtil();
        String strOutcome = utilRegister.lostPassword();
        if (strOutcome.equals("lostPassword")) {
            return "lostPassword";
        }
        return null;
    }
}
```

Listing 29: Generated Java Bean for register.jsp

The generated Java Bean is a Java class named `GeneratedRegisterBean`. The Bean contains the attributes *forename*, *surname* and *email* of type `String`. The number of attributes as well as their names are derived from the number and names of textfields of a Web page. For each attribute `get`- and `set`-methods are generated, e.g. `getEmail()` and `setEmail(String email)`. Furthermore the Java Bean contains a method for the button, i.e. the `registerUser()` method, as well as methods for each links, i.e. the `lostPassword` method. Java code is generated within those methods that instructs the developer to implement a Java class that provides the delivery of the outcomings, i.e. the `registerUser(String forename, String surname, String email)` method for the button as well as the `lostPassword()` method for the link of the `RegisterUtil` class. Furthermore the outcomings of the manually written methods are verified so that the handwritten method can not deliver an invalid outcome. If a method returns an invalid outcome the generated method returns `null` that signifies that the current Web page is displayed again.

Analog to the **register.jsp**, Java Beans are generated for each input-page as well as for each output-page that contains links. Due to the fact that all output-pages of our prototype contain links, Java Beans are generated for

each Web page.

The following section describes the generated file that contains information about the pageflow as well as information about all Java Beans used within the Web application.

4.3.1.3 Generated Pageflow Information

The JSF framework consists of a Java Servlet `FacesServlet` that controls the pageflow. This Servlet needs as input an XML file that describes the pageflow the Java Beans that are used within the Web application.

The following listing depicts the generated information about the pageflow for the **register.jsp** Web page.

```
<navigation-rule>
  <from-view-id>/pages/register.jsp</from-view-id>
  <navigation-case>
    <from-outcome>success</from-outcome>
    <to-view-id>/pages/thanks.jsp</to-view-id>
  </navigation-case>
  <navigation-case>
    <from-outcome>failed</from-outcome>
    <to-view-id>/pages/failed.jsp</to-view-id>
  </navigation-case>
  <navigation-case>
    <from-outcome>lostPassword</from-outcome>
    <to-view-id>/pages/lost.jsp</to-view-id>
  </navigation-case>
</navigation-rule>
```

Listing 30: Generated information about Pageflow

For each Web page that delivers outcomings a `<navigation-rule>` tag must be generated. The `<from-view-id>` tag specifies the name of the Web page. For each outcome a `<navigation-case>` tag must be generated that consists of a `<from-outcome>` tag and a `<to-view-id>` tag. The `<from-outcome>` tag specifies the outcome and the `<to-view-id>` tag specifies which Web page has to be displayed for this outcome.

The **register.jsp** Web page has the outcomings `success`, `failed` and `lostPassword`. Dependent on the outcome the **thanks.jsp**, **failed.jsp** or **lost.jsp** Web page is displayed next.

The following listing depicts the generated information within the *faces-config.xml* file that describes all used Java Beans within the generated Web application.

```
<managed-bean>
  <managed-bean-name>GeneratedRegisterBean</managed-bean-name>
  <managed-bean-class>GeneratedRegisterBean</managed-bean-class>
  <managed-bean-scope>session</managed-bean-scope>
</managed-bean>
<managed-bean>
  <managed-bean-name>GeneratedLoginBean</managed-bean-name>
  <managed-bean-class>GeneratedLoginBean</managed-bean-class>
  <managed-bean-scope>session</managed-bean-scope>
</managed-bean>
<managed-bean>
  <managed-bean-name>GeneratedLostBean</managed-bean-name>
  <managed-bean-class>GeneratedLostBean</managed-bean-class>
  <managed-bean-scope>session</managed-bean-scope>
</managed-bean>
<managed-bean>
  <managed-bean-name>GeneratedThanksBean</managed-bean-name>
  <managed-bean-class>GeneratedThanksBean</managed-bean-class>
  <managed-bean-scope>session</managed-bean-scope>
</managed-bean>
<managed-bean>
  <managed-bean-name>GeneratedFailedBean</managed-bean-name>
  <managed-bean-class>GeneratedFailedBean</managed-bean-class>
  <managed-bean-scope>session</managed-bean-scope>
</managed-bean>
```

Listing 31: Generated information about Java Beans

Like mentioned above, every Web page of our prototype contains a form or links. Hence, for each Web page a Java Bean is generated. Java Beans are specified within the `<managed-bean>` tags. Each `<managed-bean>` tag consists of the `<managed-bean-name>`, `<managed-bean-class>` and `<managed-bean-scope>` tags. The `<managed-bean-name>` tag denotes the name of the managed Bean, the `<managed-bean-class>` specifies the class file where the Java Bean resides and the `<managed-bean-scope>` tag denotes the duration of validity of the Java Bean.

This section described a Web application that was generated based on our created meta-model and templates. To following section evaluates our approach of a MDA that generates a modeled Web application automatically.

5 Evaluation

In order to implement a MDA that generates modeled Web applications automatically we defined a meta-model that defines the syntax and structure of models. Furthermore we introduced model transformations, so-called templates, that generate the Web application automatically. Our defined meta-model is a PIM that defines a DSL for modeling the Web pages and the pageflow within Web applications. To demonstrate the functioning of our approach we introduced a prototype. The prototype is a JSF Web application that can be deployed on the Apache Tomcat Web server. Further work must be done to define templates to generate modeled Web applications not only for the Apache Tomcat Web server. Like mentioned in the Introduction section we provide only facilities to model the Web pages as well as the outcomings of user interface components that are responsible for the definition of the pageflow. The developer must implement the Model of the MVC pattern for validating the entered data on Web pages by the user through fetching data, e.g. by accessing a database. Hence, we introduced possiblities to separate generated and manually written code.

5.1 Evaluation of the Meta-Model

The meta-model defines the DSL that is responsible to define the syntax and structure of the models that contain a Web application that should be generated. We wanted to create a meta-model to model the Web pages and the pageflow of a Web application.

This aims were approached by defining an EMF Ecore Model that contains classes to model the Web pages, their user interface components as well as an process-oriented definition of the pageflow. The pageflow can be modeled by the outcomings of action listener methods bonded to buttons and links of the Web pages. We divided Web pages into input- and output-pages. Input-Pages are Web pages that contain forms where the user can enter data. Output-pages are responsible to display dynamic or static text. Furthermore we introduced a possiblity to define the pageflow in a process-oriented manner by defining navigation rules for buttons and links. Navigation rules are modeled like `if-else` statements in Java.

This aspects lead to the conclusion that a possiblity was created for mod-

eling Web pages and the pageflow of a Web application. A shortcoming of our created meta-model is that there is no possibility to model the layout of the Web pages. That is to model regions within Web pages that contain different contents, e.g. TOP, LEFT, CENTER. This is an aspect that we can incorporate for further works.

5.2 Evaluation of the Templates

Templates define transformation rules to transform a given model to another model. Our defined templates generate from a given modeled Web application a JSF Web application that can be deployed on the Apache Tomcat Web server. We defined templates for generating JSP Web pages, Java Beans, the configuration file *faces-config.xml* for the JSF framework, the configuration file *web.xml* for the Web application for the Tomcat Web server as well as an Apache Ant file *build.xml* to facilitate the deployment of the generated Web application on the Apache Tomcat Web server. Templates that are responsible for generating Java Beans also generate Java code that calls a method of an object of a class that must be implemented by the developer. Our chosen methodology to separate generated and handwritten code is evaluated later in this section.

Resultant we defined templates that generate a complete JSF Web application where the developer is responsible for implementing classes for the Model of the MVC pattern that are responsible for fetching the needed data, e.g. accessing a database.

5.3 Evaluation of the Prototype

We wanted to demonstrate a prototype to show the functioning of our approach. Furthermore we wanted that the protoype should be a JSF Web application that can be deployed on the Apache Tomcat Web server. Our introduced prototype satisfies this requirements because our defined templates generated a JSF Web application that can be deployed on the Apache Tomcat Web server. Furthermore we generated an Apache Ant file that compiles all generated Java classes and produces an Apache Tomcat *war* file for deploying the generated Web application. The compilation process checks if the developer created Java classes that provide the called methods from the generated Java Beans. The deployment process given through the Ant file

results in errors if the developer did not satisfy this requirements. Therefore we force the developer to implement methods that process the entered data by the user through the user interfaces on the Web pages.

5.4 Evaluation of Code Separation

We base the decision of how to handle generated and handwritten code on the following statement:

Keep generated and non-generated code in separate files! [3]

The template that generates the Java Beans generates Java code that calls a method of a class that has to be implemented by the developer. Due to the fact that all Java classes reside in different files the developer must create a new file that contains the Java class that must be implement by the developer. Therefore we satisfy the recommendation by [3].

To separate generated from manually written code we chose the proven approach to call handwritten Java classes by generated code to avoid severe effects in case of interventions in the model. Changes in the templates lead to changes in the generated code. This can further leads to the problem that the developer must change the manually written classes that extend generated classes. To avoid this problem and decrease the coupling of the generated and the manually written code we proposed the following solution: generated code calls methods of handwritten classes.

This section described the aims of this work and how we tried to reach this aims. But there is still a lot of work to do in future. Further work is characterized in the following section.

6 Further Work

Our approach covers only a small part of the science of modeling and generating Web applications. Therefore is still a lot of work to do in the future. This section describes how our approach can be extended and improved by further works.

6.1 The Meta-Model

Presently our defined meta-model provides facilities to model Web pages and the pageflow of a Web application. We can extend the meta-model by providing facilities to model the layout of Web pages so that the developer can define and position regions or fragments within Web pages that contain user interface components, e.g. TOP, LEFT, CENTER etc. For the time being the developer has no possibilities to model the Model of the MVC pattern that is responsible for fetching the needed data. This is also a big point of further work. If we provide facilities to model the Model of the MVC pattern we have to evaluate if a separation of the meta-model in multiple meta-models makes sense. Related works, mentioned in the Related Work section (Section 3), provide mutliple meta-models that are responsible for modeling the View, the Controller as well as the Model of the MVC pattern.

6.2 Model Transformations

For the time being we transform a given modeled Web application only to a JSF Web application. The generated JSF Web application is deployed on the Apache Tomcat Web server. In the future we want to create more model transformations so that the modeled Web application should not only be generated for the Apache Tomcat Web server. To generate Web applications beside JSF we have to deal with the question how the Controller of the MVC pattern should be provided. Presently the Controller is given by the Java Servlet `FacesServlet` of the JSF framework that controls the pageflow of the generated Web application. We provide a template that generates the input for this Java Servlet of the JSF framework in an XML file. Furthermore we want to provide the generation of Web applications that are not based on Java technologies, e.g. PHP Web applications.

7 Summary and Conclusion

This work introduced modeling and generating Web applications based on a MDA. The generated Web applications are built-on the MVC pattern. The focus of this work has been the definition of a meta-model that specifies the DSL for models of Web applications. Furthermore transformation steps or so-called templates were created to generate a modeled Web application based on the JSF framework. The generated Web application can be deployed on the Apache Tomcat Web server. Generating Web applications based on a MDA improves the quality and speed of developing Web applications as well as their maintenance.

As shown in section 1, Web applications are nowadays the *first choice for most business applications* [23]. Therefore a MDA improves the development and maintenance of Web applications. Hence the utilization of a MDA in the field of an automated generation of modeled Web application is very helpful.

Section 2 has introduced the used technologies within this work. First it describes the principles of the MVC pattern. Afterwards the JSF framework is described. Then the theory of MDA as well as Metamodeling is introduced. For the definition of the meta-model EMF was used. To generate a modeled Web application we utilized the oAW plug-in for Eclipse. The Theory section ends with a description of possiblities how to deal with generated and handwritten code.

Section 3 shows related works in the field of generating Web applications based on a MDA. Most of the related works provide multiple models for modeling the Model, the View as well as the Controller of the MVC pattern. This related works can help us in further works for evaluating a separation of our defined meta-model in multiple meta-models.

The main focus of section 4 lies in the description of the definition of our meta-model. Furthermore the created templates are described that are utilized for the generation of the Web application. Finally we have introduced a prototype that shows the functioning of our approach. The prototype is a JSF Web application that is deployed on the Apache Tomcat Web server.

Section 5 evaluates our approach by evaluating the meta-model, the tem-

plates, the prototype as well as our chosen possiblity how to deal with generated and manually written code.

Work that should be done in the future is described in section 6. This section gives an overview how our approach can be extended and improved by extending and improving our defined meta-model as well as our created templates for the generation of the modeled Web application.

To conclude this work we can say that this report has shown that the usage of a MDA reduces the time-consuming process of developing Web applications. The field of an automatic generation of Web applications is very huge and therefore is still a lot of work to do in the future. We hope that our approach in the course of this work can support the evolution of MDAs for an automated generation of Web applications.

A Figures

List of Figures

B Tables

List of Tables

C Listings

Listings

D Bibliography

References

[1] Apache Software Foundation, *The Jakarta Site - Apache Tomcat*, HTML, 2007, `http://tomcat.apache.org`.

[2] Alan Williamson, Kirk Pepperdine, Joey Gibson, Andy Wu, *Ant - Developer's Handbook*, Sams, ISBN 0-672-32426-1.

[3] Thomas Stahl and Markus Voelter, *Modelgetriebene Softwareentwicklung: Techniken, Engineering, Management*, ISBN 3-89864-310-7, dpunkt.verlag GmbH (2005).

[4] H.-W. Gellersen, M. Gaedke, *Object-Oriented Web Application Development*, IEEE Internet Computing, 3(1), Jan.-Feb.1999.

[5] Jennifer Ball, Debbie Carson, Ian Evans, Scott Fordin, Kim Haase and Eric Jendrock, *The Java^{TM}EE 5 Tutorial, For Sun Java System Application Server Platform Edition 9*, Sun Microsystems Inc, Santa Clara, California 2006.

[6] Sun Microsystems Inc., *JavaServer Faces Technology*, `http://java.sun.com/javaee/javaserverfaces/`.

[7] Java BluePrints, *Model-View-Controller*, `http://java.sun.com/blueprints/patterns/MVC-detailed.html`, Sun Microsystems Inc., 2007.

[8] Robert Eckstein, *Java SE Application Design With MVC*, `http://java.sun.com/developer/technicalArticles/javase/mvc`, Sun Microsystems Inc., 2007.

[9] Object Management Group (OMG), *Model Driven Architecture*, `http://www.omg.org/mda`.

[10] Joaquin Miller, Jishnu Mukerji, *MDA Guide Version 1.0.1*, Object Management Group (OMG), 2003.

[11] Object Management Group (OMG), *Meta Object Facility (MOF) Specification*, April 2002.

[12] Object Management Group (OMG), *Unified Modeling Language*, `http://www.uml.org/`.

[13] Eclipse.org, *Eclipse*, `http://www.eclipse.org`.

[14] Eclipse.org, *Eclipse - Web Tools Platform (WTP)*, `http://www.eclipse.org/webtools`.

[15] Eclipse.org, *Eclipse Modeling Framework*, `http://www.eclipse.org/emf`.

[16] Sun Microsystems, *Netbeans*, `http://www.netbeans.org`.

[17] Sun Microsystems, *NetBeans Visual Web Pack*, `http://www.netbeans.org/products/visualweb`.

[18] openarchitectureware.org, *openArchitectureWare*, `http://www.openArchitectureWare.org`.

[19] Santiago Melia and Andreas Krau and Nora Koch, *MDA Transformations Applied to Web Application Development*, Proc. 5th Int. Conf. Web Engineering (ICWE'05), volume 3579 of Lect. Notes Comp. Sci., pages 465-471, 2005.

[20] S. Meliá, C. Cachero, *An MDA Approach for the Development of Web Applications*, In Proc. of 4th ICWE 04, LNCS 3140, July 2004, 300-305.

[21] Pierre-Alain Muller and Philippe Studer and Jean Bézivin, *Platform Independent Web Application Modeling*, Lecture Notes in Computer Science, 2003, ISSU 2863, pp. 220-233.

[22] Stefano Ceri and Piero Fraternali and Aldo Bongio, *Web Modeling Language (WebML): a modeling language for designing Web sites*, Dipartimento di Elettronica e Informazione, Politecnico di Milano, Italy, 2000.

[23] Jonatan Alava, Tariq M. King, and Peter J. Clarke, *Automatic Validation of Java Page Flows Using Model-Based Coverage Criteria*, Proceedings of the 30th Annual International Computer Software and Applications Conference (COMPSAC 06), 2006.

[24] Sam Chung, Yun-Sik Lee, *Modeling Web Applications Using Java And XML Related Technologies*, Proceedings of the 36th Hawaii International Conference on System Sciences (HICSS 03), 2002.

[25] JBoss, *JBoss Seam*, `http://www.jboss.com/products/seam`, 2007.

Index