Technical University Hamburg-Harburg

A Module Generator for Web application interfaces to CCMS

Master Thesis
Submitted in partial fulfilment of the requirements for the degree
Master of Science in Information and Media Technologies

Submitted by:

Shafeer Hussain Hajamohideen
Information and Media Technologies
Matriculation No. 16752

Supervised by:

Prof. Dr. Joachim W. Schmidt
Institute for Software, Technology & Systems

Prof. Dr. Friedrich H. Vogt
Institute of Telematics

M.Sc. Sebastian Boßung
Institute for Software, Technology & Systems

October 2006

Declaration

I declare that:

this work has been prepared by myself,
all literal or content based quotations are clearly pointed out,
and no other sources or aids than the declared ones have been used.

Hamburg, 31st October 2006
Shafeer Hussain, Hajamohideen

Acknowledgements

I would like to thank *Prof. Dr. Joachim W. Schmidt* of Institute for Software, Technology and Systems (STS) for giving me this opportunity to do the master thesis at his department and *Prof. Dr. Friedrich H. Vogt* of Telematics (TI5) department for being the co-supervisor.

I extend my special thanks to *Mr. Sebastian Bossung* for his guidance and thoughts during the entire thesis. I am also thankful to *Dr. Hans-Werner Sehring* for his support, advice and the fellow students at the STS department sharing their views on the project.

Finally, I wish to thank all my friends who stood by me for all the good and bad times, their help and suggestions during critical times. Also, I am grateful to my parents for keeping faith on me, their patience and *love*.

Hamburg, Germany Shafeer Hussain, Hajamohideen

Abstract

Information systems development and its user interface design are a complex task and has been undertaken using different methods and techniques. It must fulfill a variety of application and architectural requirements. Every information system has a user interface which for many purposes is perceived as the whole system. For the user, the distinction between interface and system looks, for most practical purposes, meaningless.

Furthermore, applications have domain models that are nimbly changed and also evolve to the new requirements that are added during their lifetime. Many users work collaboratively on data based on a common domain model of the application domain. The application needs to handle changes in a way that ensures less or no rework. A main source of problems of such system originates from the lack of a systematic subdivision of large software systems into manageable modules. A change in the application's domain model affects most part of the system, resulting in a time consuming and error prone adjustments. As a consequence developers are traditionally involved in a complex patchwork of refactoring various parts of the system manually to keep in sync with the system's requirements.

The Conceptual Content Management Systems have developed an approach to stay open to constantly changing domain models and dynamically evolve and integrate the changes without a manual intervention. This approach is based on a conceptual modeling language, a model compiler framework and a system architecture. The model compiler generates a system that conforms to openness and the system architecture ensures that the system adapts to the changes dynamically. Although, CCMSs lacks a user interface that would be able to work on the same approach. This thesis work is aimed at addressing this problem and explore the possibilities to provide a web application interface to CCMS.

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

Information systems development and its user interface design are a complex task and has been undertaken using different methods and techniques. It must fulfill a variety of application and architectural requirements. Every information system has a user interface which for many purposes is perceived as the whole system. For the user, the distinction between interface and system looks, for most practical purposes, meaningless.

Furthermore, applications have domain models that are nimbly changed and also evolve to the new requirements that are added during their lifetime. Many users work collaboratively on data based on a common domain model of the application domain. The application needs to handle changes in a way that ensures less or no rework. A main source of problems of such system originates from the lack of a systematic subdivision of large software systems into manageable modules. A change in the application's domain model affects most part of the system, resulting in a time consuming and error prone adjustments. As a consequence developers are traditionally involved in a complex patchwork of refactoring various parts of the system manually to keep in sync with the system's requirements.

1.1 Problem statement and objectives

In this thesis, an approach based on a model for Conceptual Content Management (CCM) [27] is used for information system development. The CCM approach profits from the dynamic, model-driven generation of smaller modules, which can be combined automatically into the full system. The generation process uses a CCM model of the application domain(s) from which a compiler framework dynamically generates the schema-dependent parts of the system. Due to the dynamic nature of this generation process, it is possible to provide adequate support for both schema evolution and personalization of such a system.

However, the current development stage of the CCM does not have means to provide a user interface to its conceptual content management system (CCMS) [4]. This thesis work is aimed to address this problem:

- To generate a web application interface based on the domain model of the application and thus,
- To explore the possibilities in providing a user interface that fulfill the requirements of open and dynamic content management systems.

Web applications are built according to a layered architecture and is also based on a application domain model. The *data layer* which stores the information including, data, content, attributes of the domain entities as well as the management information needed to enable all the operations of the application and presentation layers. The *application layer* mediates between data storage and presentation. The *presentation layer* represents the information to the user based on the data and takes user input to navigate, create, modify and delete data. All these layers are highly depend on the conceptual model of the application. Since the layers are interrelated, despite the layering, changes made to one layer affects the whole system. This aspect of system evolution based on changes to the schema can be complemented by the CCM approach and thus leads up to the following objectives of this thesis:

- To generate a web application interface to conceptual content management systems
- To identify and generate schema dependent part of the system
- To find ways to integrate the dynamic code generation with manually written application code, thus to support schema evolution and personalization

1.2 Approach

The main objective is to provide a user interface to a CCMS, which in context of a web application is the presentation layer that is responsible for interacting with the user. In CCMS, the separation of concerns are enabled by layers of modules. The main concern of providing a web application interface to the CCMS would be to implement the server module. The server module is responsible for providing an interface to enable external communication, be it a human interface or a standardized distribution mechanism.

The presentation layer not only represent the information but also to interact with the user. Therefore, the requirements to implement the user interface are not only the conceptual model but also the usability issues. Usability serves as a measure of how effective, efficient and satisfying the usage of the system is from the user's perspective [12]. Most of the web application

functionalities such as representation of data, form based input and manipulation of data can be generated. Other issues such as error handling, custom layout and configuration require manual extension of the generated code. This part of the system should be invariant to the schema and has to be integrated on top of the generated code. The functionalities of the data layer and the application layer would be implemented by the respective modules below the server module.

1.3 Related work

There has been several attempts to provide a user interface to CCMS, either by generic means or as a homogeneous system. As pointed out in [4], user interface development deals with numerous special cases and requires some manual work. The following four possibilities have been identified to influence the generation of user interfaces:

- Changing the conceptual domain model
- Configuring the generator
- Passing parameters to the generated system, and
- Hand-coding parts of the user interface

A generator for web applications has been implemented that combines these four cases but is weak in its configuration language.

In [26], the author provides an approach to *conceptual content management application development by means of storyboarding*. It is based on the *SiteLang* [35] modeling paradigm for rapid prototyping of user interfaces and provides a web application specification language that allows to generate a web application by means of Storyboarding.

[24] presents another approach to model user interfaces for CCMSs. In this work, a custom UI Component model has been proposed in addition to the domain object model to implement a prototypical Swing (fat client) interface. It also mention about user interfaces created in generic fashion by means of User Interface Markup Language (UIML) [1]. UIML is an XML based language for describing user interfaces that can be implemented on any platform. The advantages of such languages is that it can be used in a verity of platforms such as desktops, handheld PCs, and Mobile phones. Another example in this area is the Abstract User Interface Markup Language (AUIML) [23] which assists in developing graphical user interfaces running as Swing (fat client) or Web applications (thin clients). XML User Interface Language (XUL) [6] and eXtensible Application Markup Language (XAML) [11] are some more examples of XML-based user interface definition languages. The drawback, as studied in this work, is that

implementing user interfaces using the above mentioned technologies are not flexible enough coping with system evolution.

All these works are done to build user interfaces for a open and dynamic content management system based on the CCM approach. This thesis works aims to contribute a generic web application interface (thin-client) by exploring the possibilities in today's web development world in realizing one for the CCMSs.

1.4 Outline of the thesis

In the next chapter 2 the conceptual content management approach is discussed, giving a more detailed overview of its approach, the asset language, the compiler framework and its architecture. Chapter 3 will discuss the existing application frameworks, analyze its features and explores how they can be used to build an interface for a web application that supports the open and dynamic content management system. Chapter 4 will explain the design decision made to transform the conceptual model into the different facets of the chosen application framework. Chapter 5 discuss further on implementation details of a web application generator that follows the design decisions and will also address any limitations in realizing it. Chapter 6 will explain the main contributions that have been done in this project and also the limitations and future work based on this thesis.

Chapter 2

Conceptual Content Management Systems

Chapter 1 explained the difficulty in maintaining software adaptations to information systems which are based on a common conceptual model. Since users need to have a personal and subjective view of the conceptual model, the applications require openness (defined in next section) to the model so that users can change it according to their needs. Also, the system must dynamically adapt to the changes made to the model without manual intervention. This chapter discusses how these issues are handles using the CCM approach.

2.1 Conceptual Content Management

A system based on the conceptual content management approach should allow users to support subjective view of the application domain, allowing users to express their subjective view on the conceptual model. Furthermore, the system must also be able to interact with the personalized view of the individual users as well as to maintain the content available before personalization in order to exchange data between other users of the system. As proposed in [9], the CCM approach address these requirements by developing a open and dynamic content management system.

The generated systems are based on a component architecture and each component are specific to the application domain. Components are constructed from modules which provide a separation of concerns. A module is a unit with interfaces to all other modules. Each of these specific modules are created by a generator. A compiler framework which constitutes a set of generators will generate the complete system. The CCM architecture enables the reuse of code by using the same module implementation on various levels.

As seen above, to enable the users to change the conceptual model to the underlying system and to react to the changes without a manual intervention, the system supports the following properties:

- *openness* that allows the users to express their personal view of the application domain
- *dynamics* that allows the users to employ their personalized view to create instances of the system

To achieve this the CCM approach contributes:

- A conceptual modeling language called the *Asset Language*
- A model *compiler framework* that generates conceptual content management systems (CCMS)
- A *system architecture* for these systems

The primitives of openness and dynamics allow the system to realize schema personalization, schema evolution and interaction of application domains [29].

2.2 Asset Definition Language

In the CCM approach, the entities of real world are modeled as content-concept pairs called *Assets* based on the observation that neither content nor concept exist in isolation. Figure 2.1 represents an Asset. The *content* represents the multimedia content—texts, images, maps videos etc—of an Asset whereas the *concept* consists of the *characteristic* properties of entities, its *relationship* with other assets and *constraints* to those characteristics and relationships. More details on the Asset language can be found in [31].

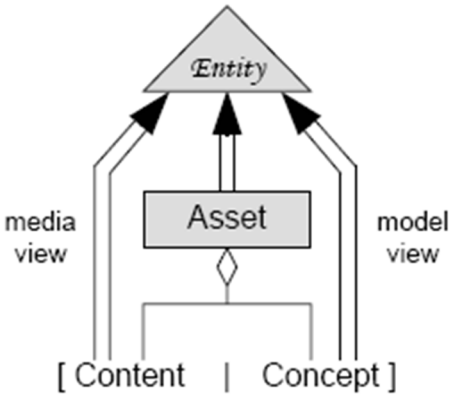

Figure 2.1: Content-Concept representation of an Asset [27]

The Asset Definition Language (ADL) [27] defines the conceptual model of the application domain. The model consists of an aggregation of Asset class definitions which describes the structure of assets. The structure and syntax of Asset definitions corresponds to the class definitions of the object-oriented languages. Listing 2.1 illustrates an example a Asset definition:

```
model Iconography
from Artists import Artist

class Picture {
  content image : Image
  concept
  characteristic title : String
  characteristic placeOfCreation : Place
  relationship artist : Artist
  constraint placeOfCreation : artist.placeOfBirth
}

class Portrait refines Picture {
  concept
  characteristic sizeX : int
  characteristic sizeY : int
}
```

Listing 2.1: Example of a model containing Asset definitions

The Asset definitions are organized under the keyword `model` followed by its name and the Asset classes belonging to the model. An Asset class consists of the two sections, the `content` which references to the content part (multimedia) and `concept` section which consists of the conceptual part representing the attributes of the entity by the following three contributions.

2.2.1 Characteristic

Characteristic values are inherent to an entity and are identified by a name and a type handle determined by the underlying base language. In the above example every `Picture` has a title of type `String`. The currently supported base language for these type handles is Java.

2.2.2 Relationship

Relationship between assets describe entities by their relation to others. It is identified by a name and the type of the Asset to which it is referred. The relationship `artist` in the example references to another entity described by the Asset `Artist` that has been imported from another model called `Artists`. An asterisk (*) on a relation type refers to a many-to-many relationship on a set of associated Asset instances.

2.2.3 Constraint

Constraints impose value restrictions on the attributes of instances of an Asset. In the example here, the `placeOfCreation` in which the `Picture` has been created is required to be the same as that of the `placeOfBirth` of the associated `Artist`.

Besides Asset definition language, there is also the *Asset Query Language* and *Asset Manipulation Language*. They provide means to query Asset instances (`lookfor`), to create new instances (`create`), to manipulate (`modify`) as well as delete (`delete`) existing instances. These commands have variants that can handle single or multiple instances at one time as well as more complex tasks. For more details refer to [31].

2.3 Model Compiler

A complete CCMS is created by an instance of the compiler framework using an appropriate set of generators inserted at extension points. The compiler is created as framework which controls the overall compilation process following the classical compiler architecture. The framework consists of frontend and backend components where generators form the backend. A central generator of the backend is a API generator which takes the conceptual domain model and produces a uniform module API. A server module for a web service would require a generator for WSDL interface descriptions and the uniform module API. Section 2.4.1 explain details about the different modules which implement the module API, to provide different functionality to the system. The execution of generators are scheduled by interdependencies based on a extended notion of symbol tables, which are used for communication between the generators. The symbol tables also provide the schema and API definitions. Figure 2.2 shows a model compiler framework.

2.4 CCMS Architecture

To satisfy the dynamic property of the CCM approach, the conceptual content management systems (CCMSs) must dynamically react to changes made by the users to the application domain model. To achieve this behavior the CCMS architecture follows a standardization of creating a complete system from a set of *modules* which are the building blocks of individual systems at large.

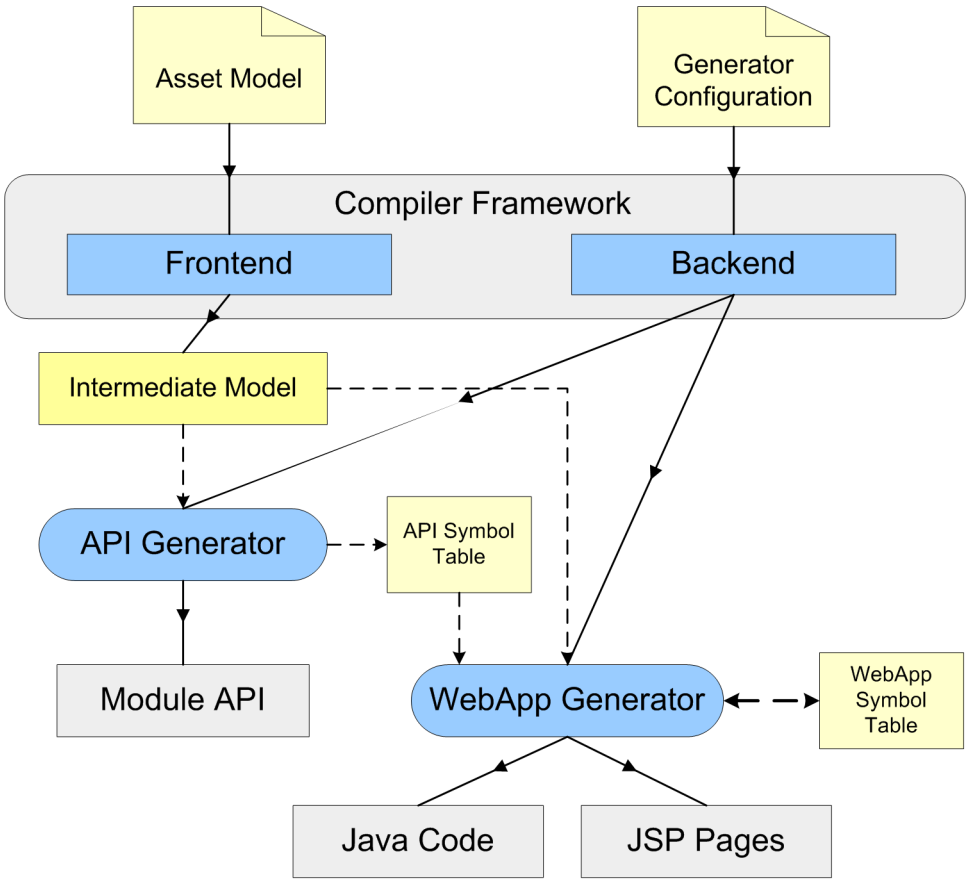

Figure 2.2: Model compiler architecture

2.4.1 Modules

Modules are self-contained units, each having a specific purpose in the system as a whole to provide separation of concerns. They are the generation targets of the compiler framework and are arranged in layers. The modules on the each layer use the modules on the layer below them and provide services to the layer above. A standardized module interface common to all modules provide the means to interact between modules. This interface provide means to create, modify and delete assets as well as query existing ones. All these operations have variant forms that deals with a single or a set of assets. The following operations describe one of these variants:

- `create(class, inits)`: returns a new Asset instance of type `class` and initializes its attributes according to `inits`.
- `modify(a, updates)`: updates the attributes of an Asset instance a with `updates` which can contain information in the form of name-value pairs

- `delete(a)`: deletes an asset instance `a`
- `lookfor(class, constraints)`: executes a query for the instance of `class` with the given `constraints`

Due to the common interface, the operations are available to all modules and deal with Assets of a fixed model. The modules can be combined in various ways, while still being domain specific. The kind of modules available are discussed below.

Client Module

Client modules provide the uniform interface to the modules on the higher layers. They map the calls they receive to the third party system, such as a database system used to store Asset instances. They do not use any further modules and other client modules can forward calls to remote systems, such as a system accessed through a web service.

Server Module

Server modules are complementary to client modules. They use the layers below them through the standard interface but do not provide this interface. They are intended to provide other interfaces, for example a user interface (thin or fat clients) to the CCMS or a web service.

Transformation Module

Transformation modules use as well as provide the standard interface and based on exactly one module on the layer below them. They have three types of transformations, namely *Temporal Transformation* which acts as persistence mechanism, *Spatial Transformation* which is used to achieve physical distribution between systems and *Schema Transformation* use and provide the standard module interface but conform to different schemata and generate a module for instance-level transformation of Assets.

Mediation Module

Mediation modules act as a mediator proposed by [36] to provide access to different sources of information in a homogeneous way. Mediators include a wrapper around the information sources and some mediation logic to tie them together. This mediation logic is implemented by mediation modules and the wrapper by transformation modules.

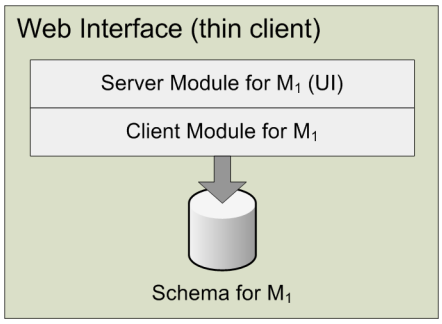

Figure 2.3: Component implemeting a User Interface based on the model M_1

Figure 2.3 shows a component implementing a web based user interface based on the conceptual (domain) model M_1. The client module is responsible for providing the data stored in a database. The server module queries the client module's standard module interface and receive a reply back with the data it needs to present to the user. More details on modules and its uses can be found in [29].

2.4.2 Components

Modules of a same model are combined into *components* to achieve a particular task, for example providing personalized instances. Components provide services to their modules, the important one being identifier resolution. Based on the identifier of an Asset instance, each module can ask its component to retrieve the instance. This request will be delegated to all top modules by the component and returned to the caller. This enables a module to reach instances that are not stored in the base module but elsewhere in the component.

The assembly of a component is described in its configuration containing the details on which modules the component is composed of, their dependencies as well as the parameters (e.g database names,) to pass to the modules. Modules can be reused in different location by means of the component configuration. Components interact with other components by through Transformation modules, explained in the previous section. For interaction with external systems such as a relational database, a web service or a user interface, the boundary of component is described in the client or server module.

2.4.3 Systems

A conceptual content management system is formed by combining the components. On the one hand, components of the same system can use different models depending upon the openness and dynamics of the users personalized view on the application domain. For this reason, distinct components are necessary to accommodate the personal opinion of the user as well as to ensure

the accessibility of existing instances. On the other hand, inter-domain operations require two components which handles the assets of that particular domain. More details on how application models can be used in different ways and how application domains can be modeled by reusing existing models can be referred in [29].

2.5 Summary

The CCM architecture is suited for generating systems based on a conceptual model. Its ability to combine modules to realize specific functionalities and standardized interfaces with domain specification make it possible to generate such systems. It supports system evolution by utilizing openness and dynamics. The architecture and its corresponding facilities for system creation have been used in many application projects. As seen above, the server module has to be implemented to provide a web application interface to CCMS. The possibilities of providing such a thin client using existing technologies and frameworks will be discussed in the next chapter.

Chapter 3

Frameworks for building Web Application Interfaces

In section 1.2 proposed that to provide a web application interface to the CCMS only the presentation layer of an conventional web application build on a layered architecture has to be implemented. Furthermore, the model compiler generates the target code in the base language of the CCMS, which currently use Java. For building enterprise application in Java, the J2EE platform has been emerged as a standard platform. To ensure a seamless interaction to the changes made to the domain model, the frameworks used should be able to support the openness and dynamics of the CCM approach.

3.1 Introduction

Design patterns are proven and reusable and are most important resources to application developers. They provide a common language to express experience and save time and effort. One of the core J2EE design pattern used in today's most web applications is the Model-View-Controller (MVC) pattern. Figure 3.1 shows a variation of the MVC pattern specific to web applications, known as *Model 2*. The *Model* consists of plain old java objects (POJOs), EJBs and in CCM its Assets. The *View* can be JSPs, XSLT or any other view technology. The *Controller* is always implemented as a servlet.

It creates a decoupling between data access, data presentation and user interaction layers. This enables fewer interdependencies between components and higher level of re-usability, such as implementing new views without changing the underlying code. This separation allows different people work with the layers independently and also lets portions of the some layers be integrated before all of the three layers are complete. These are the benefits why most frameworks implements some variation of this MVC pattern for building interactive web applications.

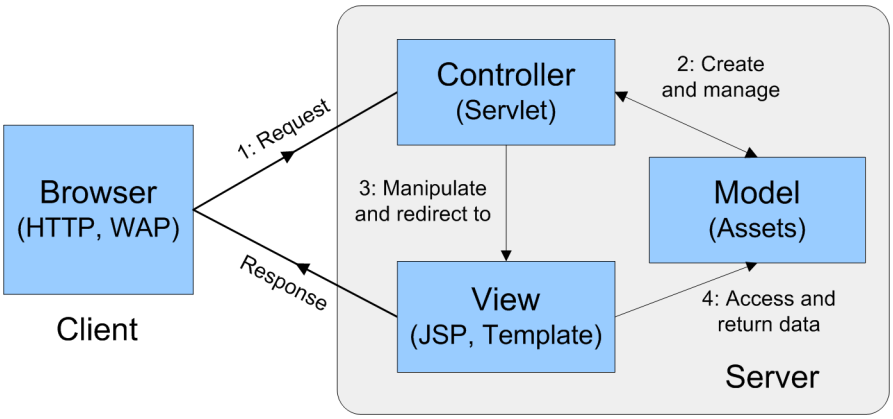

Figure 3.1: MVC design pattern

3.2 Using existing Web Application Frameworks

The "first principle of reusable object-oriented design" advocated by the classic Gang of Four (GoF) design patterns book [14] is: "Program to an interface, not an implementation". This decouples the interfaces from their implementation. Using interface-based architecture is important in J2EE application because of their scale. A few advantaged of an interface-based approach include:

- The ability to change the implementation class of any application without affecting calling code. This enables us to parameterize any part of an application without breaking the component.
- Total freedom in implementing interfaces. There is no need to commit to an inheritance hierarchy. However, its still possible to achieve code reuse by using concrete inheritance in interface implementations.
- The ability to provide simple test implementations and stub implementations of application interfaces as necessary and enable multiple teams to work parallel after they have agreed on interfaces.

More about design techniques and recommendations for J2EE applications can be found in [18]. The best practice would be to choose an existing and proven web application framework for e.g., Struts [2], WebWork [25], Spring [20] and JSF [33] to name a few. The next section analyzes two of these frameworks which could prove to be flexible and provides components to build a dynamic, server-side user interface for CCMSs.

3.3 Spring Application Framework

Spring is a open source framework which allows to build lightweight and robust J2EE applications. It follows the above principle of implementing to interface and based on the famous *Inversion of Control (IoC) / Dependency Injection* pattern. The concept behind this pattern is expressed as the Hollywood Principle: 'Don't call me, I'll call you'. Martin Fowler's article [13] would give a decent insight of this pattern. The basic use of Inversion of Control is that objects are not created directly, but described how should they be created. The components and services are not connected together in code but a configuration file contains the description on which services are needed by which components. A container, in the case of Spring framework, the IoC container takes the responsibility to hook everything together.

3.3.1 Spring Web MVC

The Spring framework consists of seven well defined modules built on top of the core container, which defines how beans are created, configured and managed. Figure 3.2 below shows a layered architecture of a full fledges web application using Spring framework.

Each of the seven modules shown at the background can stand on its own or implemented together with one or more of the others. The functionality of each of these modules are as follows:

- The **Core** provides the essential functionality of the framework. The primary component here is there `BeanFactory`, which implements the Factory pattern and the IoC pattern separating the application's configuration and dependency from the application code.
- **Context** is a configuration file providing a way to access beans and services such as JNDI, email, internationalization, validation and scheduling.
- The **AOP** module integrates the aspect-oriented programming functionality into the framework.
- The **DAO** is a JDBC abstraction layer that simplifies database access and manges exception handling by parsing vendor specific error codes.
- The **ORM** package provides integration layers for several ORM frameworks including JDO, Hibernate and iBatis and comply with the transaction management and DAO exceptions hierarchies mentioned above.
- Spring's **Web module** build on top of application context module, providing features such as multipart functionality and contexts for web applications. It supports integration with other frameworks like Struts, WebWork etc.

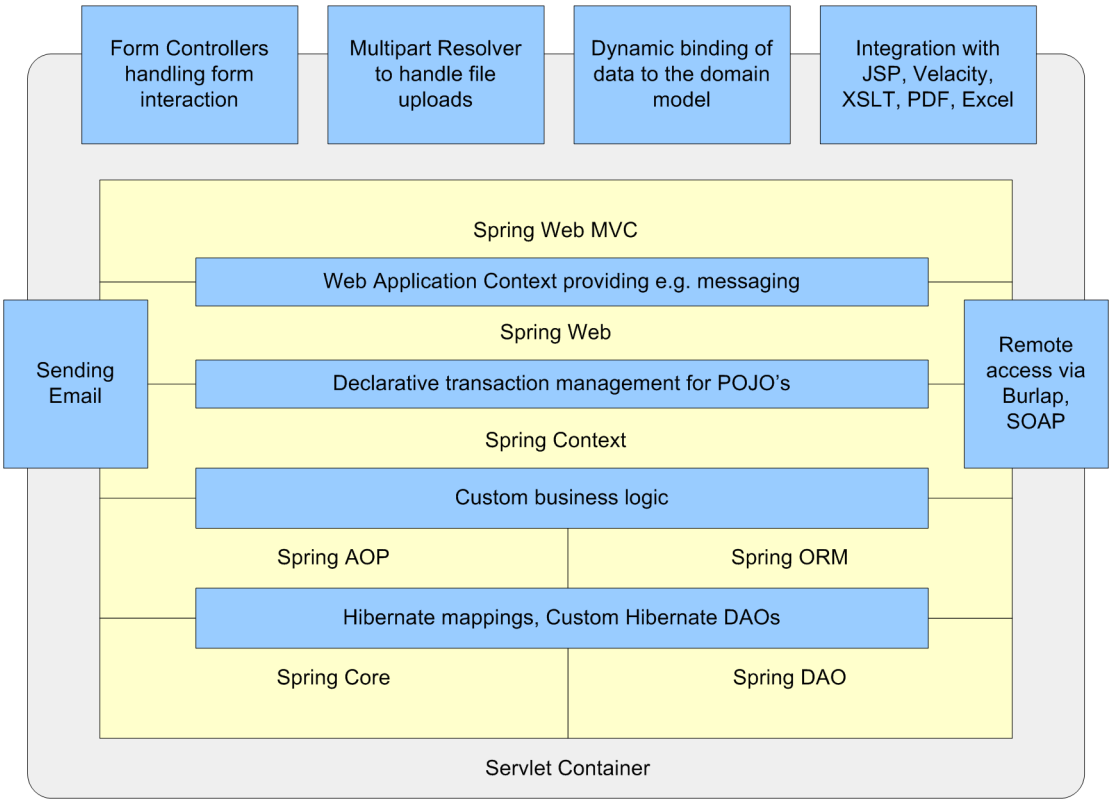

Figure 3.2: Overview of a typical Spring web application [20]

- The **MVC framework** is a full featured MVC implementation for building web applications. It is highly configurable, provides a clean separation between code and web forms, accommodates numerous view technologies including JSP, Velocity, Tiles, iText, POI and even JSF.

The architectural benefits of Spring, as proposed by the author in [19], is that one can choose to use part of it in isolation, either be it to simplify the use of JDBC or to manage business objects. It is essentially dedicated to enable you to build applications using POJOs to implement the business logic tier, which would be applicable to a wide range of environments. Form controllers seamlessly integrate the web layer with the domain model, removing the need of `ActionForms` to transform HTTP parameters to values of the domain model. It allows to reuse business objects and data-access objects that are not tied to specific J2EE services. Such objects can be reused across Web or standalone applications without any hassle.

3.3.2 Application development with Spring Web

The MVC module is designed around a `DispatcherServlet` that dispatches request to handlers with handler mappings, view resolution, locale as well as support for file up-

loads. The default request handler is a simple `Controller` interface with just one method, `ModelAndView handleRequest(request, response)`. Spring contains different kinds of controllers, which are form-based such as `AbstractFormController` or command-based `AbstractCommandController` and also `AbstractWizardFormController` that execute wizard style logic. Application controllers typically subclass them and an appropriate base class has to be selected with respect to the form that is being used on that view. Spring allows any object to be used as a form or command object and need not require any framework specific interface or base class. Understanding these features and how they enable building an web application interface with Spring Web is demonstrated in A.

3.3.3 Using Spring Web for the Presentation layer

Spring's full-featured Web MVC module is powerful and provides a clean division between controllers, JavaBean models and views. Its is flexible and completely based on interfaces. As seen in A, it separates the roles of the controller, model, dispatcher and the handler object which makes them easier to customize. Spring also provides mechanism to use custom templates by implementing the Spring View interface to integrate it.

The benefits of using Spring MVC as the user interface framework would be that it is view agnostic, which means that the developer is not forced to use JSP, but any other view technologies like XML/XSLT. This makes the application maintainable for future update or substituting the current one with another one (JSTL/JSF) or enhance to a newer version (JSP 2.0/EL) or using different technologies in conjunction. In this case, only the page templates has to be re-generated leaving the controllers and the form or command objects intact.

Apart from choosing a view to present to the user, the presentation layer must also enforce constraints on model data and also validating user inputs. The way these responsibilities are handled has a significant impact on the development efforts, which in our case, code generation. Choosing Spring Web for the presentation layer needs the generator to implement controllers suitable to the page being processed. For code generation, this means that for every view associated with a model object, choosing an appropriate controller would be difficult and error prone.

Even though any object can be used as form object or command object, it is feasible to implement one for each Asset to provide any view specific data and the operations performed on those Assets. Then, generating bean configuration files registering the controllers and their command classes and views as well as resource mappings.

Since validation and conversions in Spring are evaluated as application-level errors, invalid submissions should be handled by a validator object. The validator requires implementation of two methods : the `support` and the `validate`, which check the validation method and carries out the validation. This requires for every form objects, the generator should implement a validator to handle those application errors thrown by invalid form submission. All these features are highly flexible, but would rather be a tedious task considering the complexity of code generation.

3.4 JavaServer Faces

JavaServer Faces (JSF or simply *Faces*) is an increasingly popular component-based, event-driven, tools-friendly server-side framework. It enforces a clean separation of presentation and business logic. However, it focuses more on the UI side of things and can be integrated with other frameworks. The functionality provided by JSP 2.0, JSTL and MVC would be enough to generate simple dynamic web pages with limited user interaction. But when it comes to developing complex user interfaces, lot of work has to be done to extract request parameters, validate, process and render them back to HTML controls, which is tedious and error-prone. Faces allows easy-to-use tools that shorten the time needed to implement server-side user interfaces.

3.4.1 Faces and MVC

Faces follows the MVC architecture by providing a controller servlet, allows separation of the view using renderers and the model using component class and an event based mechanism. The way components relate to each other is described using XML markup with custom JSP tag libraries. The following features differentiate Faces from other application frameworks.

- A **UI component model** with event listeners and handlers for object-oriented web application development.
- **Bean management with Backing beans** that are JavaBeans associated with the UI components and separates UI component objects from objects that perform business logic and process data.
- **Extensibility of UI components** that compose the user interface of Faces application by allowing to configure, reuse and extend the components to develop complex ones.
- **Flexible rendering model** that separates the UI components functionality from the view.
- **Extensible conversion and validation model** to provide a enhanced protection by extending the standard converters and validators.

In the following sections, the key conceptual base of these features that Faces provides are briefly explained to understand what they mean in a Faces context and how they are used in writing a Faces application. A detailed insight into these features, list of component tags, and other information on JSF, please refer to the JavaServer Faces specification at [16]. Figure 3.3 shows a high-level view of a Faces application and how it integrates with other subsystems, like EJB or database services.

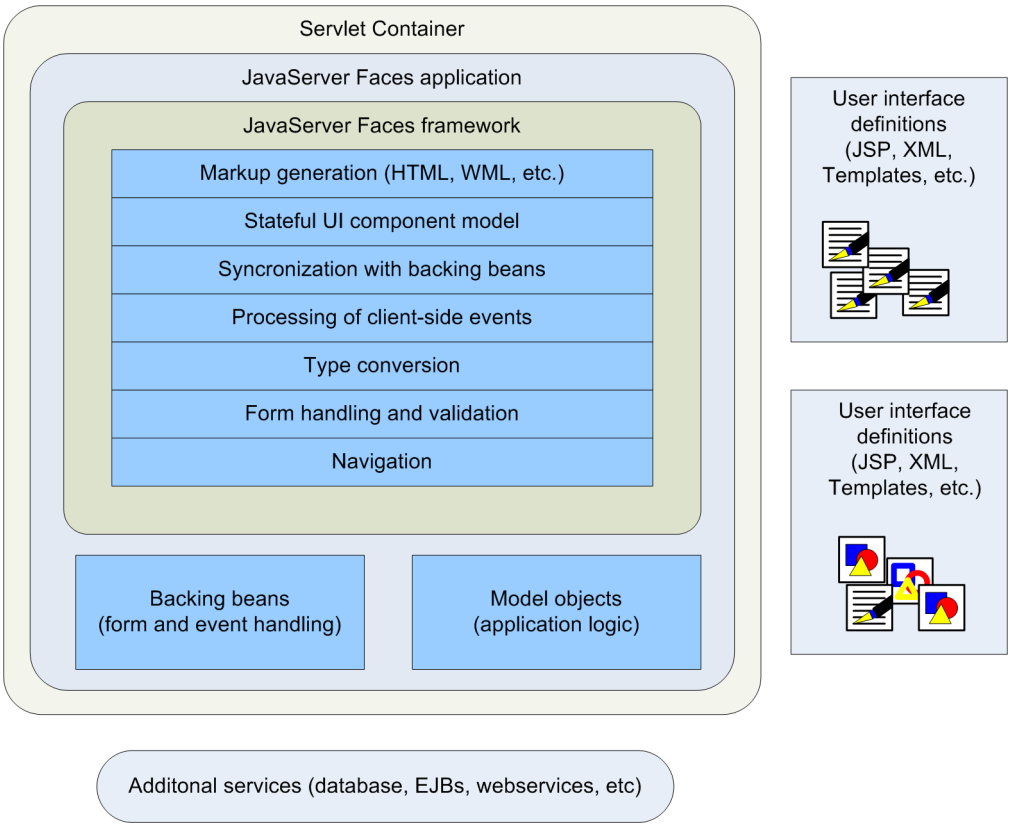

Figure 3.3: A view of JavaServer Faces application [22]

UI Components

Faces user interface components are stateful objects that provides specific functionality for interacting with an user. Unlike in RAD (Rapid Application Development) environments like Visual Basic, UI components in Faces are build on JavaBeans with properties, methods, and events. User interface elements are packed as a component, which makes development faster and easier because the basic functionality of a components is same and can be reused with only changing some properties such as color or style. They are specifically designed for web applications which means that they are on the server side of the application rather than the client side. They are organized into a *view*, which is a tree of components usually displayed as a page. This enables the components remember their values between requests. Components are identified by a *component identifier*, which can be set by the developer and can be associated with one another through named relationships, for example “header” or “footer”—called *facets*.

Building user interfaces with Faces is more about assembling and configuring the components than writing tedious code. Faces includes standard components such as labels, hyperlinks, text boxes, list boxes, radio buttons, panels and data grids. Apart from these standard components, it also provides way to extend or create custom components.

Renderer

Renderers are responsible for displaying a UI component and translating a user's input into the component's value. They are organized into *render kits* which focus on a specific type of output. It works like a translator between the client and server. It creates a representation of a component in a way that the client can understand. It also processes the response from the user to extract the correct request parameters and set the component's value based on those parameters.

For example, the following code defines a `HtmlInputText` component:

```
<h:inputText id="name" size="20" maxlength="30"/>
```

This component when encoded and sent to the user will produce the following HTML snippet:

```
<input id="myForm:name" type="text" name="myForm:name" size="20"
   maxlength="30"/>
```

Renderers can be designed to work with one or more UI components, and a UI component can be associated with many different renderers.

Validator and Converter

Validators ensure that the value entered by a user is acceptable. It is a complex and error prone task to check them with the correct type of data to be entered and the displaying the errors appropriately. Faces handles validation not only through validator methods in backing beans or in validator classes and but also at the UI component level. Although, component level validation can handle only simple validation such as length or range of the input, pluggable external validators can be used that can be attached to any component. One or more validators can be associated with a single UI component. The errors encountered by the validator will be added to the current message list which makes it easy to display the errors back to the user using standard Faces components. The following example shows a `Length` validator associated with an `HtmlInputText` component that check if the given user input is between two and ten characters long.

```
<h:inputText>
  <f:validateLength minimum="5" maximum="10"/>
</h:inputText>
```

Components usually associated with backing bean properties, which can be a `String` representing a name or a `Date` representing a date-of-birth. Converts a component's value to and from a string for display. A UI component can be associated with a single converter. They also handle localization and formatting. For example, the `DateTime` converter can format a `Date` into a short, long or a full style. The date will also consider the user's locale while displaying it in the given style. The following code shows how to register a converter on an `HtmlInputText` component:

```
<h:outputText>
  <f:convert_datetime type="both" dateStyle="long"/>
</h:outputText>
```

Apart from converting common data types, converters also allow application developers to write converters for their own model objects.

Managed and Backing Beans

Faces introduces two new terms : *managed bean* and *backing bean*. The managed beans are specialized JavaBean objects managed by Faces implementation to describe how a bean is created and managed. They collect values from UI components and implement event listener methods. They can also hold references to UI components.

The backing beans plays the role of the controller in MVC pattern. They contain properties that has to be retrieved from users and handling-logic associated with the UI components used on the page. A component is associated with a backing bean through the Faces expression language (EL), which is similar to the JSTL and JSP 2.0 expression languages. For example, the following code snippet looks up an `HtmlInputText` component's value to the `name` property of an `PersonBean` object. Whenever the value of the component or the `personBean.name` property changes, they are kept in sync automatically.

```
<h:outputText id="personName" value="#{personBean.name}"/>
```

Each backing bean property is bound directly with either a component instance or its value. It also defines methods that process some functions to manipulate a component, such as validating the component's data or handling events fired by the component and the changes are updated the next time the page is displayed to the user. A backing bean can be shared with one or more views and vice-versa. This avoids code duplication forcing a one-to-one relationship between a backing bean and a view. Understanding how Faces interacts with backing beans and model objects is an essential part of building a Faces application.

Events and Listeners

Events capture the way user interacts with UI components, such as clicking on a component or executing a command. Faces uses the JavaBeans to handle events and listeners. Events provides an alternative to the complexity of developing web applications in terms of requests and responses. In Faces, application logic is integrated by assigning appropriate listeners to components. UI components (and other objects) generate events, and listeners can be registered to handle those events.

Faces provide four standard events, namely value-change events, action events, data model events and phase events.

- *Value-change* events are fired when a user changes the component's value of a input control.
- *Action* events are fired when a command component, like a button or a link is activated.
- *Data model* events are fired when a data-aware component is selected for processing.
- The *phase* events are executed while processing an HTTP request.

Events and listeners are fundamental part of a Faces application.

Messages

Displaying error messages properly back to the user is one of the issues in developing user interfaces. Basically, errors are classified into two categories: *application* errors resulted from application logic or database connection errors and user input errors such as a invalid input or required entries. Application errors are usually displayed on a different page whereas input errors redisplay the same page. Also, error messages of an input field or control are consistent over different pages.

Faces provide messages that consists of a summary text, detailed text and severity level to display the error messages on a view. Just about any part of the application (backing beans, validators, converters, and so on) can generate information or error messages that can be displayed back to the user. The messages can be associated with a specific component (input errors) or no specific ones (application errors). For example, the following snippet shows the error message associated to a specific component by using the `HtmlMessage` component:

```
<h:message id="errors" for="personName" style="color: red"/>
```

This code displays all errors that were generated for the `personName` input component, which is declared on the same page. Since messages are integral part of the Faces validation and type conversion, error messages are generated when a validator encounters an invalid input or a converter processes a incorrect type. It also provide ways to customize standard application messages and to create them in Java code.

Navigation

Web applications have multiple pages and navigation provides the ability to move from one page to the next. JSF has a powerful navigation system that's integrated with specialized event listeners. The *navigation handler* decides what page to load based on the logical outcome of an action method. A *navigation rule* defines what outcomes are understood and what pages are loaded on those outcomes. Each specific mapping between an outcome and a page is called as *navigation case*.

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

Listing 3.1: An example of a navigation rule in JSF

The example above shows a navigation rule with each navigation case maps the outcome to a specific page without any extra code. The rules are defined in a Faces configuration and are maintained in a single file which acts as a central location to all pages.

3.4.2 Using JavaServer Faces for the Presentation layer

One of framework's primary goal is to ease the burden of integrating user interface with the model. Compared to the `ActionForm` and `Action` approach in Struts, development with backing beans in Faces enables a better object-oriented design practices. A backing bean not only contains view data but also behavior related to that data. Traditionally, validation can be a tedious web development task and so do the generation of validators for every single form object. The standard Faces validators would reduce this complexity and also enable the use of third party or custom validators. The advantages of using Faces is that it provides:

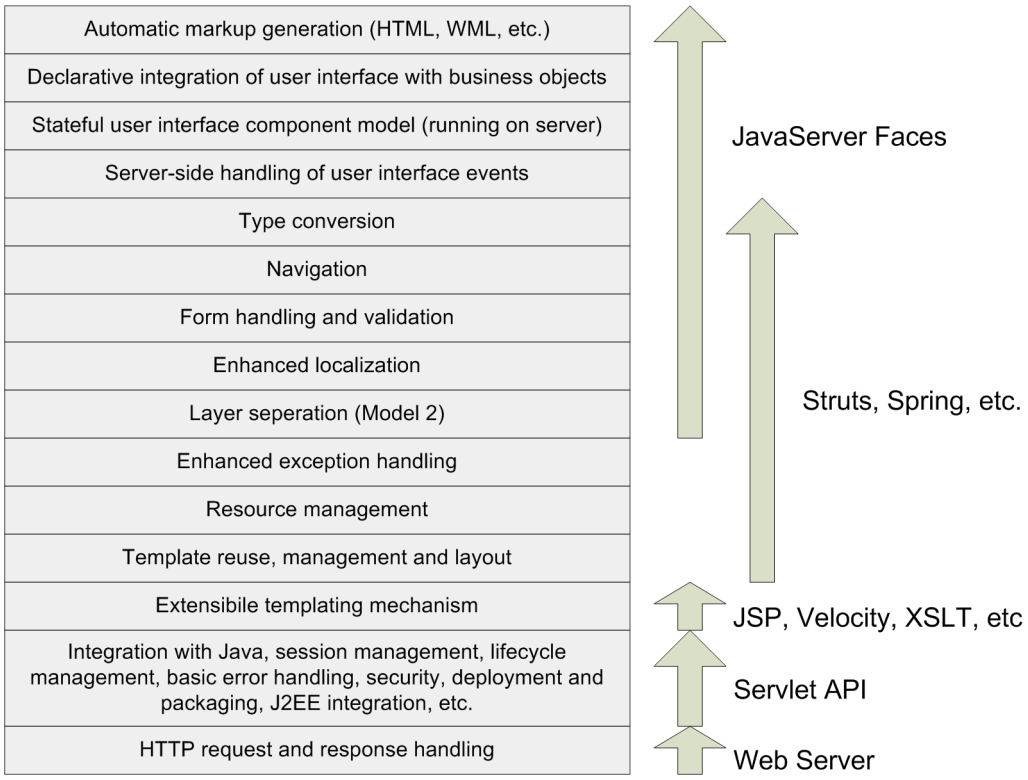

Figure 3.4: Web application infrastructure as a stack of services [22]

- Java APIs to represent UI components, handle events, manage state and validate input.
- Custom tag libraries to express the UI components within the JSP pages or templates and wiring them with the server-side objects.
- Provides clean separation between mode and view.
- Flexible rendering mechanism and support for other view technologies.
- Extendability to all its features and third party tools.

Choosing JavaServer Faces for the presentation layer needs the generator to implement backing beans (JavaBeans) representing the model objects containing the application specific functionality, JSP pages with Faces UI components, managed bean configurations with navigation rules and any custom tag libraries, custom validators or listeners needed.

Faces can also be combined with Spring framework. Since we need to implement only the presentation layer and the access to other layers would be provided by the respective modules of the CCMS, using would be considered overkill. Figure 3.4 helps understanding this overlap between framework functionalities. In this thesis project, the need is more directed to a UI-oriented framework, which is why the reason to choose Faces. Also it supports enough services

to fit to the need to generate a user interface based on the domain model. Apart from that additional services can be added in future as well.

3.5 Summary

Frameworks are extremely common these days and they help make web development easier. Frameworks enforce a clean separation of presentation and business logic. For this thesis, a framework that focuses more on the UI side of things would be beneficial. Spring provides a powerful and flexible MVC framework. It provides ways to manage business objects, uses IoC container to provide solutions that addresses all the architectural layers, supports use of different display technologies, data access abstraction and transaction management.

Faces is a UI framework for building web applications. It provides standard UI components (buttons, hyperlinks, check boxes and so on), options for creating custom tags and components, server-side event processing and a tool support to simplify coding web-based applications. On how the user interface is implemented by transforming an Asset model based on ADL into a JavaServer Faces components, how the framework configuration works and what it takes to get the application running will be discussed in the next chapter.

Chapter 4

Designing a Web Application Interface

In accordance with the CCM approach discussed in chapter 2, the next step is to implement the server module which will provide the web application interface to CCMS. After exploring the possibilities on building a web application interface using existing application frameworks in chapter 3, JavaServer Faces has been chosen as the UI framework for the presentation tier. This chapter will discuss the ways to build a Faces application from an Asset model by creating backing and managed beans, JSP pages with Faces UI components (views) and defining page navigations. To make understand this design, a online product catalog application is considered as an example through out this discussion. Listing 4.1 represents the Asset definition of the applications conceptual model and provide the basis for a meaningful discussion in generating a web application interface.

4.1 Creating User Interface from an Asset Model

In Faces, backing beans defines properties and handling logics associated with the UI components used in a page. Views can be built in a form-centric manner in which each view is associated with a single bean containing objects that represent data for that page. Components of the view can be directly connected to the data sources using the backing-beans with action methods that perform any operations. This approach of one-to-one binding might work well for a small application, however for large applications a more object-oriented approach would be ideal. As shown in Figure 4.1, this approach requires to develop model objects representing the application domain, backing beans that work with view and access the model objects. With this approach, several views can share the same backing bean and provides an abstraction to the source of data. It is also possible to combine these two approaches.

```
model Catalog

class Product {
     content
       productImage: String
     concept
       characteristic name: String
       characteristic price: int
       characteristic description: String
       relationship category: Category*
}

class Category {
     concept
       characteristic name: String
       characteristic description: String
}
```

Listing 4.1: Asset definition of product catalog application

4.1.1 Asset Members

Asset members are the model objects of a CCMS (left part of Figure 4.1). Each `AssetClass` is associated with a backing bean which relates to a view object (middle part of Figure 4.1). In contrast to the model object that contains only data, a view object is a model object that contains additional presentation specific data and behavior. The backing beans are named after the Asset names followed by a *Bean* as suffix. For example, the Asset `Product` in the model 4.1 is implemented as `ProductBean`. The listing below shows how this is represented as a backing bean for the interface implementation. For clarity, the backing bean in general will be denoted as an <*AssetBean*> throughout this discussion.

```
public class ProductBean { ... }
```

The <*AssetBean*> will have properties that associates with the concept part of an Asset which will be referenced by a UI component. Apart from properties, the <*AssetBean*> will also have action methods (discussed in section 4.1.6) and action listener methods that performs the operations on an Asset. As learned from section 3.4.1, the backing beans are *declaratively* associated with a component of the view (right part of Figure 4.1). With JSF expression language (EL),

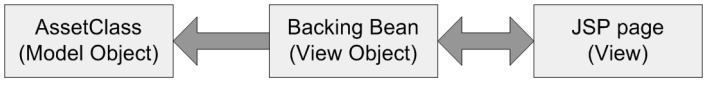

Figure 4.1: Association between Model - Backing Bean - View

any property or action of the <*AssetBean*> can be accessed somewhere in the application. The next three sections will explain how the conceptual part is integrated in the <*AssetBean*>.

4.1.2 Content

The content part of the Asset is presented as multimedia contents. In 4.1, every `Product` has a `productImage`, which depicts a visual representation of the `Product`. The type of these content handles depend on the base language, for which currently Java is used. The `productImage` handle is defined to hold a `String` literal, which can refer to a URL or a absolute or relative path to a file or directory. It can also be defined as an `Image` object or a binary stream that can be stored in the underlying database. The `productImage` can be implemented as a static property in a common backing bean that holds a URI to the product image directory. For this project the `String` variant is used and since every instance has a `productId` property, the name of the image file is assumed to be the same. This would be advantageous to associate the image property with URL property of the `HtmlGraphicImage` component as shown below.

```
<h:graphicImage url="images/products/#{productBean.productId}.jpg"/>
```

4.1.3 Characteristics

Characteristics are immanent properties of an Asset and in ADL they are defined by the keyword **characteristic**. In 4.1, for example, every `Product` has a `name`, a `price` tag and a `description` of the product. The types of these characteristics are determined by the base language as well. This simplifies the possibility to map them directly as native Java primitive as well as built-in types to the corresponding bean properties of the <*AssetBean*>. The bean properties are referenced through getters and setters (also called *accessors* and *mutators*) via value-binding by the UI components that access these properties in the their respective views. Type conversions are discussed in section 4.2.1.

```
public class ProductBean {
  private de.tuhh.sts.cocoma.generic.ID productId;
  private String name;
  private String description;
  private int price;
  ...
  ...
}
```

Listing 4.2: Representing Asset characteristics

Listing 4.2 shows the characteristics as bean properties and their corresponding built-in Java types. The setter and getter methods to these bean properties are generated as well. In addition to characteristics, an `productId` property to identify the instance of an Asset has been included to every *<AssetBean>*. Thus, `ProductBean` would have a bean property called `productId` of type `de.tuhh.sts.cocoma.generic.ID` as shown in the Listing above. Adding this property will be beneficial in many ways, especially when dealing with views, one of which is discussed in the next section.

4.1.4 Relationships

Relationships are established between Assets which describe autonomous entities and are defined by the keyword **relationship**. There are three kinds of relationships an Asset class can be defined with: *one-to-one*, *many-to-many* and *recursive*, which are based on their cardinalities between the entities. To demonstrate these types, consider the example given in Listing 4.3.

```
class Product {
  ...
  concept
    ...
    relationship make: Company
    relationship category: Category*
}

class Category {
  concept
    relationship categories: Category*
}

class Company {
  concept
    characteristic name: String
    ...
}
```

Listing 4.3: Types of Relationship between Assets

One-To-One

Here, the relationship `make` is a one-to-one relationship, which means every `Product` is made by at most one `Company`. A one-to-one relationship is implemented as a *<AssetBean>* property that holds an instance of the relation Asset itself and its type. In the given example 4.3, the relationship will be implemented as shown below.

```
public class ProductBean {
  ...
  private Company make;
  ...
}
```

Listing 4.4: Representing One-to-One relationships

Similar to other bean properties, the one-to-one relation can be referenced by components using them through value-binding in their respective views. The `name` property of the company (`make`) to which the product (`ProductBean`) belongs to is associated with the value property of the `HtmlOutputText` component as shown below.

```
<h:outputText id="product:madeBy" value="#{productBean.make.name}"/>
```

Many-To-Many

The relationship `category` depicts that every `Product` is classified into one or more categories that it belongs to. The asterisk (*) on a relation type `Category` refers to this many-to-many relationship. Similar to one-to-one relationship, many-to-many can also be implemented as a *<AssetBean>* property except that it is a list of Asset instances of the relation Asset. For small applications with less number of items this would be a good option, but for large applications this is not feasible. In that case, it makes sense to hold only a list of IDs of Asset instances of the relation Asset and the relationship will be implemented as shown below.

```
public class ProductBean {
  ...
  private List selectedCategoryIds;
  private List categorySelectItems;
  ...
}
```

Listing 4.5: Representing Many-to-Many relationships

Binding a bean property of type `List` in a view can be realized with *SelectMany* UI components, which are responsible for selecting one or more items from a list. Depending on the size of the list, a drop-down list (`HtmlSelectManyMenu`), combo-box (`HtmlSelectManyListbox`) or a group of check-boxes (`HtmlSelectManyCheckbox`) associate the contents of the list with a set of dynamic values. The list can be populated from a lookup or reference data table. The following listing shows a combo-box that allows the user to select one or more categories from the dynamically populated `categorySelectItems` instances, which is then associated with the `selectedCategoryIds` property of the `ProductBean`.

```
<h:selectManyListbox value="#{productBean.selectedCategoryIds}" id="
   selectedCategoryIds">
  <f:selectItems value="#{productBean.categorySelectItems}" id="
     categories"/>
</h:selectManyListbox>
```

Listing 4.6: Example view for Many-to-Many relationship as Combobox

The `categorySelectItems` is a convenience property that returns a collection of `SelectItems` instances containing all available categories which is populated by `init` method, as shown in Listing 4.7. Instead of having `SelectItem` instances, the property can also return a `Map` whose key/value pairs will be converted into `SelectItem` instances.

```
protected void init(){
   ...
   AssetClass categoryAssetClass = this.module.getClass("Category");
   AssetIterator iter = module.lookfor(categoryAssetClass, new
      QueryConstraint[]{});
   while (iter.hasNext()) {
     Category category = (Category) iter.next();
     this.categorySelectItems.add(new SelectItem(category.getID(),
        category.getName()));
   }
   ...
}
```

Listing 4.7: Example for populating a Many-to-Many relationship property

The bean property `categorySelectItems` could be implemented on the (`ProductBean`) itself along with `selectedCategoryIds` or as a property in a different bean, for example in `CategoryListBean` or a `ApplicationBean` which will be common for the whole application. The major difference in implementing this property in a separate bean depends on the scope of the bean being accessed in the application. If `categorySelectItems` has to be available only for a *request* or through out the *application*. More on bean management and its scope will be discussed later in this chapter.

Recursive

The previous sections discussed the two types of relationships in which two different Asset classes were related to each other. In example 4.3, the Asset class `Category` defines one more kind of relationship. The relationship `categories` relates the Asset class `Category` to itself and is called *recursive* relationship. However, this follows the same relationship type as the many-to-many relationship by definition with an asterisk (*), it is considered to be a many-to-many relationship.

When the Asset class is related to itself it does not mean that its instance is related to itself. However, an instance of this Asset class is related to another instance of the Asset class. In this example, an instance of `Category` can be a part of other `Category` instance. The implementation of this relationship is same to the many-to-many relationship, except that it has to be populated prior in the `CategoryBean` for the property `categories` before it can be populated in the `ProductBean`. Thus, it required two separate views for the user to make this selection.

4.1.5 Inherited Asset Members

New asset classes can be defined by inheriting existing asset classes using the keyword *refines*. It inherit the definitions of all characteristics, relationships and constraints from the parent class but the subclass may redefine or extend them as well as define their own. Listing 4.8 shows a new Asset class `PremiumProduct` introduced to the model that inherits the Asset class `Product` and defines two more characteristics, `width` and `height`.

```
model Catalog
...
class PremiumProduct refines Product {
  concept
    characteristic width: int
    characteristic height: int
}
...
```

Listing 4.8: Example of inherited Asset members

Inherited members are implemented similar to the Asset member as a <*AssetBean*> but inherits all the bean properties of the parent <*AssetBean*>. Thus, `PremiumProductBean` will inherit all the properties defined in `ProductBean` plus the two new characteristics as its bean properties, as shown below 4.9.

```
public class PremiumProductBean extends ProductBean {
  private de.tuhh.sts.cocoma.generic.ID premiumProductId;
  private int height;
  private int width;
  ...
}
```

Listing 4.9: Representing inherited Asset members

However, the components that are associated with these bean properties should be accessed in separate views. That is, `ProductBean` will be displayed on its own view with components associated with all of its properties, where as the `PremiumProductBean` will have a separate view representing the components associated with all inherited properties of `ProductBean` along with its own properties, as shown in Listing 4.10.

```
<td align="left" width="400">
   <h:inputText value="#{premiumProductBean.name}" id="name"/>
</td>
<td align="left" width="400">
   <h:inputText value="#{premiumProductBean.description}" id="
      description"/>
</td>
...
<td align="left" width="400">
   <h:inputText value="#{premiumProductBean.width}" id="width"/>
</td>
<td align="left" width="400">
   <h:inputText value="#{premiumProductBean.height}" id="height"/>
</td>
```

Listing 4.10: Example view for inherited Asset members

4.1.6 Operations on Asset

Besides ADL that defines Assets, there also exists Asset query and manipulation language that allows operations, i.e., to query and manipulate Asset instances. All these operations are available in different forms to handle single or a set of Asset instances. In this thesis, operations are limited to the basic operations of the module interface, i.e., `create`, `edit`, `delete` and `lookfor`. For every operation, a action method is implemented in the corresponding <*AssetBean*>. Thus every <*AssetBean*> has a `createAction`, `updateAction` and `deleteAction` respectively. The following section explain the straightforward forms mapping implemented for this thesis.

Create

The `createAction` method is responsible for creating a new Asset instance. The form of operation used here is `create(class, inits)` to create the Asset instance of type `class` and initializing its attributes with `inits`. The <*AssetBean*> properties of the Asset hold the values entered by the user are mapped to the newly created Asset instance before its saved in the underlying database. The data mapping is necessary because of the separation of the two different object models between <*AssetBean*> and Asset instances as shown in Figure 4.1. To map the data between the two object models the reflection-based `BeanUtils` is used.

Update

The `updateAction` method is responsible for modifying and and updating an existing Asset instance. The Asset instance `asset` is first loaded by calling `lookfor(ID)` with its ID and then `modify(asset, inits)` is called to initialize the attributes of the Asset instance with the values contained in `inits`. Similar to create, data mapping is carried out by `BeanUtils`.

Delete

The `deleteAction` removes an Asset instance by calling `delete(asset)`. The Asset instance `asset` is loaded prior to `delete` by calling the `lookfor(ID)` with its ID.

Lookfor

Similar to other operations `lookfor` also has many form, one of which has been shown in the previous sections to load an Asset instance by passing its ID. Apart from *<AssetBean>*, a separate backing bean is implemented to contain the search actions on an Asset and other presentation-specific data and behavior, for example, pagination logic. For every Asset there exists such a backing bean which will be denoted hereafter as *<AssetListBean>*. This also enables to apply different scopes on availability of data to the application.

Backing Bean Name	Scope	Properties	Action Methods
AssetBeans	Session or Request	Asset characteristics and relationships	Create, update and delete Actions
AssetListBeans	Request	List of AssetBeans	Search Actions
ApplicationBean	Application	Application level properties	

Table 4.1: Overview of the Application Beans, its Scope, Properties and Actions

Table 4.1 summarizes the beans representing Assets into view objects with its properties, action methods and the scope used to access it. The *<AssetBean>* provides access to visual representation, contain its characteristics and relationships as properties, provides operations to query and manipulate them and valid within a request. The *<AssetListBean>* contains properties that hold a list of *<AssetBean>*s that can be presented to the user based on the search operation and

also valid within a request. Additional to these beans, <*ApplicationBean*> is included that contains properties that are common to the whole application domain, such as a URI to a directory to store Asset contents and valid throughout the application. Armed with this information, the Assets can be intelligently hooked up to the views by JSF EL expressions. The beans provides functionality for one or more specific views; the `create` and `edit` operations on an Asset share the same backing bean.

4.2 Technology Dependent Artifacts in the User Interface

4.2.1 Importance of Type Conversion

In order for user interfaces to display objects (bean properties) in terms that the user understands, they must be converted into strings for display. These strings can be based on different factors, for example dates displayed on the format of the user's locale or simply conversion of different built-in types. This feature is important for code generation based on conceptual mode because the user need not worry about implementing handlers for type conversion every time the model is changed.

One of the features of Faces is its support for type conversion. Faces converts the value of a bean property to a `String` for display to the user. If no converter is registered on the UI component associated with the property, then the converter registered for that type will be used. In case for some reason a value cannot be converted then an error message will be generated and can be displayed back to the user.

Faces provide standard converters for basic Java types: `BigDecimal`, `BigInteger`, `Boolean`, `Byte`, `Character`, `Integer`, `Short`, `Double`, `Float` and `Long`. For example, if a component is associated with a property of type `Integer`, it will be converted using the `Integer` converter. Faces also allows using custom built converters and third-party vendor provided ones. A converter can be associated with almost any components that accepts user input.

4.2.2 Validating User Input

As discussed in section 3.4.1, the standard components level validators available in Faces are simple enough to handle validation such as length or range of the input, but supports pluggable external validators. For example, a `SelectItemsRange` validator can be developed with a custom tag (as shown below) to validate the number of items selected by the *SelectMany* UI component in Listing 4.6.

```
<h:selectManyListbox value="#{productBean.selectedCategoryIds}" id="
    selectedCategoryIds">
  <catalog:validateSelectedItemsRange minNum="1"/>
  <f:selectItems value="#{productBean.categorySelectItems}" id="
      categories"/>
</h:selectManyListbox>
```

Listing 4.11: Example of a custom validator for *SelectMany* UI component

Although, custom validators can be implemented easily, currently there is no standard way to model validation constraints for characteristics and relationships in ADL. For this thesis, no custom validators have been developed as it is not ambiguous to decide which UI components should use which set of validators based on the model definition.

4.2.3 Bean Management

By now, the use of backing beans in Faces application should be clear and the last step is to make sure the application can access them. In order the JSF EL expression be able to find them, the backing beans have to be exposed as a scoped variable (see Table 4.1). This is handled by the managed bean configuration, where backing beans are configured as managed beans and expose the model objects, i.e., the Assets through those backing beans. It allows to specify which objects will be available throughout the lifecycle of the application. The example below 4.12 shows registering `ProductBean` which is responsible to handle the `Product` Asset instances from the catalog model 4.1 discussed earlier.

```
<managed-bean>
  <description>
    Backing bean that contains product information.
  </description>
  <managed-bean-name>productBean</managed-bean-name>
  <managed-bean-class>de.tuhh.sts.cocoma.catalog.view.bean.ProductBean</
      managed-bean-class>
  <managed-bean-scope>request</managed-bean-scope>
  <managed-property>
    <property-name>productId</property-name>
    <value>#{param.productId}</value>
  </managed-property>
</managed-bean>
```

Listing 4.12: Example of a Managed Bean Configuration

As seen in this example, the managed bean facility provides possibilities to initialize object properties. The `productId` property of the `ProductBean` is populated by the request parameter

`productId`. Faces implementation gets the parameter from the request and sets the managed-property. The `ProductBean` is set to have a scope of request which means for each request a new `ProductBean` instance is references in the JSP page.

4.2.4 Page Navigation Rules

Section 3.4.1 explained Faces navigation handler that operates on a set of navigation rules which defines the applications possible navigation paths, represented by means of navigation cases. A navigation rule specifies which pages can be selected from a specific page or a set of pages. The navigation case is selected based on a logical outcome. Figure 4.2 show a general navigation rule in which for every operation on an Asset a navigation case is defined with a page that has to be loaded on the outcome of that operation. In the front page, for every Asset there exists a action which displays a list of Asset instances on "asset list" page. On the Asset list page, there exists actions to create, edit, delete and view an Asset instance. Except for view action, the outcome of these action depends on two cases: namely "success" and "failure" which navigates to a result or error page, respectively. The navigation rules are configured in *faces-config.xml*.

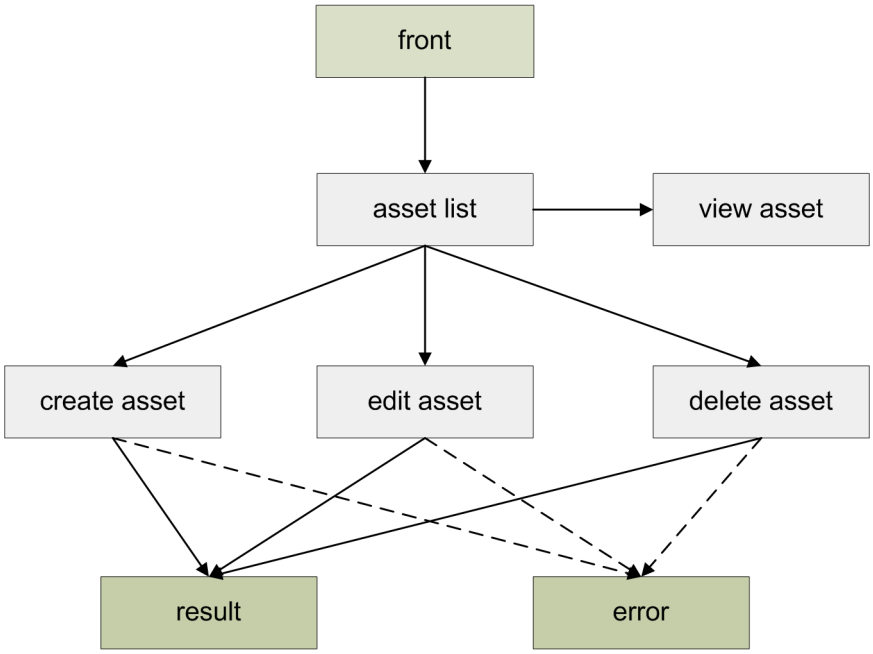

Figure 4.2: Overview of navigation rules

To make this navigation rule and its navigation cases clear, consider the example of the Asset `Product`. The front page consists an action that provides a list of products (productList.jsp) from which the user can choose to view (viewProduct.jsp), create (createProduct.jsp), edit (editProduct.jsp) or delete (deleteProduct.jsp) a product. Depending on the outcome of these action methods, the success (result.jsp) or failure (error.jsp) page is loaded.

4.3 Design overview of Web application generator

The *WebAppGenerator* enables a unidirectional mapping of conceptual model into the entities of a Faces application. Figure 2.2 shows a typical model compiler architecture with generators at its extension points. The front-end of the compiler does the parsing and checking the Asset definitions (4.1) and produces a internal representation of the Asset model in the form of a *Intermediate Asset Model*. As depicted in Figure 2.2, the Intermediate model provides access to all Asset classes, its characteristics, relationships and constraints.

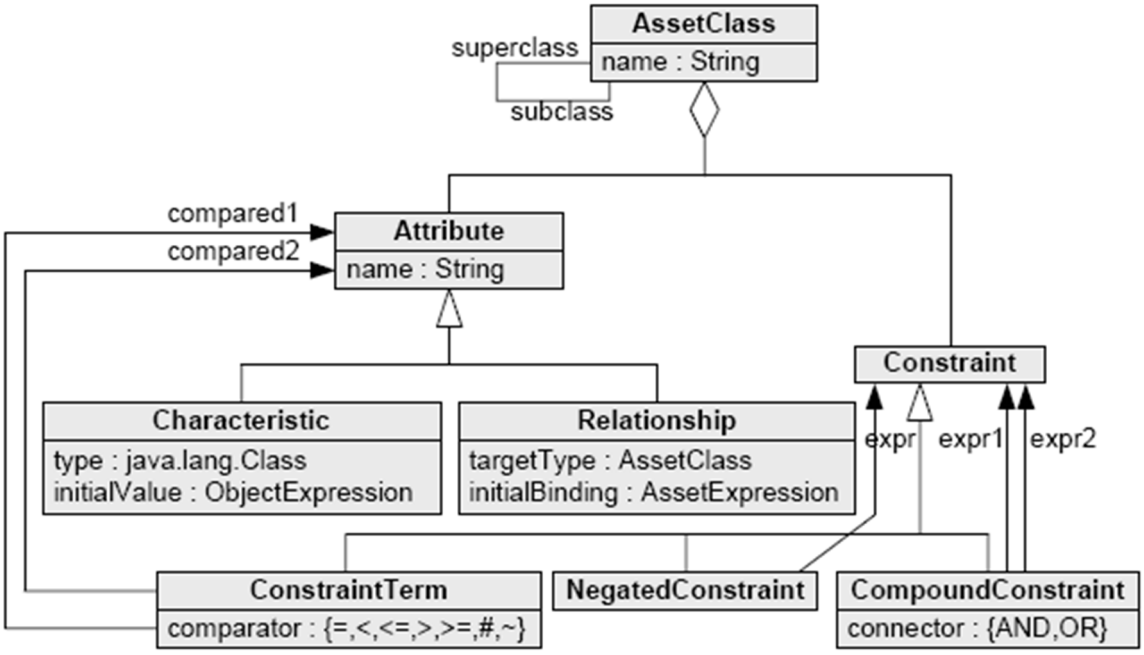

Figure 4.3: Meta model of the Intermediate Asset Model [29]

The Intermediate model is passed to the *APIGenerator*, which is usually the first generator to be executed and produces the *APISymbolTable* from which unique interfaces are available to all generators. Generators communicate with each other through symbol tables and can depend on the symbol tables of many other generators. The dependency between the symbol tables are considered as the dependencies between the generators themselves and determines their scheduled execution. Therefore, the *WebAppGenerator* depends on the *Module API* generated by the *APIGenerator* which has to be executed prior to it.

The backend of model compiler takes a generator configuration file with all the parameters required for the execution of the *WebAppGenerator* and produces the Asset Object API. By extending the `Generator` class of the Module API, the *WebAppGenerator* gets access to all these components. Listing 4.13 shows the code generation process is triggered by invoking the `generate` method.

```
public SymbolTable generate(IntermediateModel im, SymbolTable[] tables,
    Map<String, ? extends Object> params) throws GeneratorException {
   init(im, tables, params);
   ...
   try {
   ..
    generateAssetBeans();
    generateAssetListBeans();
   } catch (ModelException e) {
    e.printStackTrace();
   }
   generateJSP();
   ...
   return getWebST();
 }
```

Listing 4.13: A typical code generation process for generators

The `generate` method takes an Intermediate model, API symbol table and the parameters defined in the generator configuration file as input parameters. This method returns the symbol table it creates and produces Java (backing beans), JSP and HTML (views) and XML (faces configuration) files as a side effect of its execution. Invoking `getIm().getClasses()` from the Intermediate model give access to all the Asset classes. Its characteristics and relationships will be available by invoking `ac.getCharacteristics()` and `ac.getRelationships()` respectively.

Java Code Generation Toolkit (JCGTk) is responsible for generating Java code. Unfortunately, the Code Generation toolkit does not provide any libraries to generate JSP or HTML code and the pages are generated as `String` arrays and then written into a file. This is a tedious task because these sting arrays are wrapped around with Java code that will provide access to the Asset definitions needed to create these pages and the generated pages are not possibly well formed. The implementation of the *WebAppGenerator* is discussed in detail in the next chapter.

4.4 Summary

This chapter explained the design issues in building a Faces application from a Asset model. Asset characteristics and relationships were represented as Backing beans properties and operations on Asset instances as Actions. Backing beans were also implemented with handling logics associated with the UI components used in a JSP page. Backing beans can be shared with one or more views thus preventing code duplication and provides an abstraction to the source of data. Backing beans are registered and managed through the Managed Bean facility in order to expose the underlying Asset to the user interface. It also explained the need for type

conversions, validating user input as well as navigation rules and how Faces handles them. The next chapter applies these design decisions to implement a generator that will generate a user interface based on Faces application framework.

Chapter 5

Implementation of a Web Application Generator

Now that having studied the different facets of converting an Asset model, its time for the next step to implement a Web application generator that will generate these facets from the conceptual model by using the Model compiler. In this chapter will discuss the implementation details of the generator and also any implementation issues and limitation arise in realizing a open and dynamic web application interface to compliment the CCM approach.

To demonstrate the the design decision made in the last chapter and apply them in the generator implementation, consider the conceptual model 4.1 of an real world Web application. The model `Catalog` identifies that the system is composed of two model objects `Product` and `Category`. Each product belongs to at least one category which is why the relationship `category`, represented by an asterisk (*). To make the implementation simple, it is assumed that the catalog is not frequently updated, no internationalization is required and no more than 300 products exist in the catalog.

5.1 Structure of Code Generation

Writing generators to generate application software is a complex task because setting up an infrastructure for them is difficult [32]. In chapter 2 it has been clearly explained the way CCMSs are created from a set of components comprised of one or more modules with generators at extension points. Figure 5.1 depicts detailed structure of the compiler framework with a Web application generator.

As discussed in 4.3, the code generation process starts when the `generate` method executes with the Intermediate model, API symbol table and the parameters defined in the generator

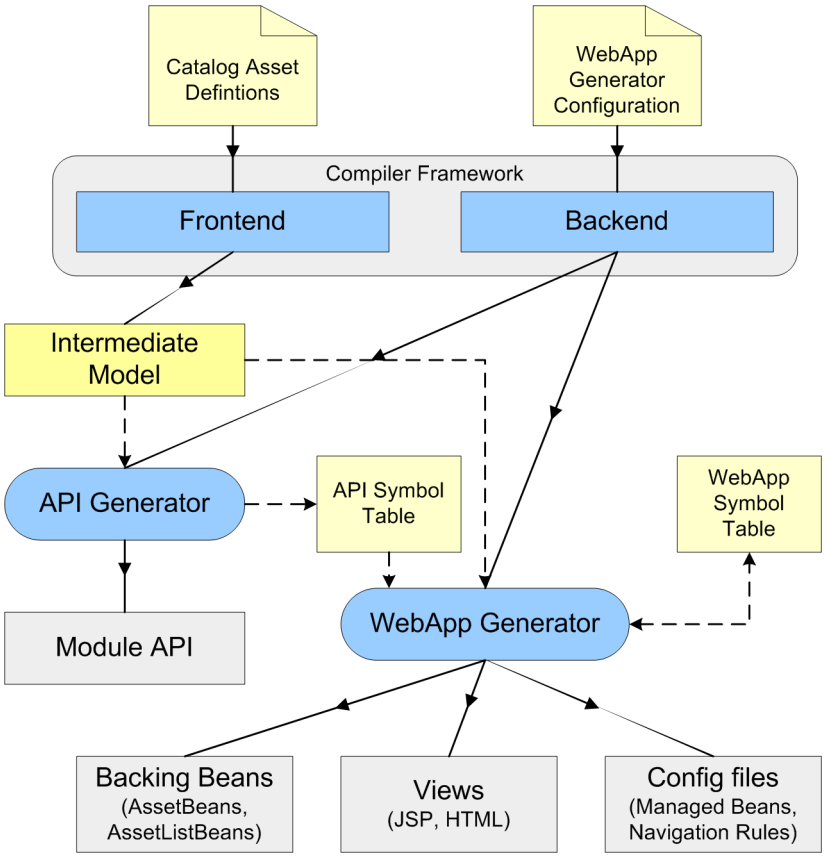

Figure 5.1: Web application generator

configuration file as input parameters. This method returns the symbol table it creates and produces Java (backing beans), JSP and HTML (views) and XML (faces configuration) files as a side effect of its execution. The structure of code generation is also categorized on these steps and will be discussed in the following sections.

5.2 Generating Backing Beans

The backing beans are generated with help of *Java Code Generation Toolkit* (JCGTk), which is used by the CCM generators to generate Java code. Details on the functionalities of the toolkit can be found in [30]. As seen in Figure 5.1 the compiler framework translates the Asset model to an *Intermediate* model which contains the Asset class definitions. First, by invoking the method `getClasses` from the Intermediate model, an array of `AssetClasss` are retrieved. By iterating over this array, every `AssetClass` of the model `Catalog` is retrieved and invokes a set of methods in order to convert the Asset model into backing beans (<*AssetBean*>s and <*AssetListBean*>s) with properties and action methods as discussed in 4.1.

```
private void generateAssetBean(AssetClass ac) throws ModelException,
    GeneratorException {
    JavaClass baseBeanClass = (JavaClass) getModel().getType(BaseBean.
        class);
    JavaClass beanClass = new JavaClass(getModel(), getBeanPackage(),
        JavaVisibility.PUBLIC, createBeanName(ac), baseBeanClass);
    ...
    ...
}
```

Listing 5.1: Generating <*AssetBean*>s and <*AssetListBean*>s from Assets

Listing 5.1 shows that every <*AssetBean*> extends a `BaseBean`. The `BaseBean` is defined with a common `moduleLocator` property that will hold a reference to the base module which provides access to the query and manipulate Asset instances. This makes it easier to initialize the property's value when the bean is created in the Managed Bean configuration file, instead of in Java code and keeps from defining the same property in multiple backing beans. Apart from that it also has `init` method that will be implemented by all the <*AssetBean*> and <*AssetListBean*>s. The package in which the beans are created is defined as a parameter in the generation configuration file B.1.

Characteristics of the Asset class are obtained from invoking `ac.getCharacteristics()` and they are added to `beanClass` as properties with *setter* and *getter* methods as shown in Listing 5.2. Each characteristic have name and a type that can be invoked by `assetChar.getName()` and `assetChar.getTypeName()` and set as bean property name and type respectively. The `JavaCodeGenerationHelper.javaGetterMethodName` is a convenience method to create a getter method name.

```
for (Characteristic assetChar : ac.getCharacteristics()) {
    JavaField beanProperty = new JavaField(JavaVisibility.PRIVATE, (
        JavaType) getModel().getType(assetChar.getTypeName()), assetChar.
        getName());
    beanClass.addField(beanProperty);
    generateGetter(JavaVisibility.PUBLIC, beanClass, beanProperty,
        JavaCodeGenerationHelper.javaGetterMethodName(beanProperty.
        getName()));
    generateSetter(JavaVisibility.PUBLIC, beanClass, beanProperty,
        JavaCodeGenerationHelper.javaSetterMethodName(beanProperty.
        getName()));
}
```

Listing 5.2: Generating <*AssetBean*> properties from Asset characteristics

Next, relationships has to be added to `beanClass` and they are available by invoking `ac.getRelationships()`. Similar to characteristics, relationships are also defined as bean

properties depending on the cardinality of the relationship. Every relationship has a name and its referred relation Asset type, which can be invoked by `assetRel.getName()` and `assetRel.getReferredType()`.

```
for (Relationship assetRel : ac.getRelationships()) {
   if (assetRel.isToMany()) {
      JavaField relProperty = new JavaField(JavaVisibility.PRIVATE,
         getModel().getType(Set.class), assetRel.getName() + "Ids");
      beanClass.addField(relProperty);
      JavaField relSelectedProperty = new JavaField(JavaVisibility.
         PRIVATE, getModel().getType(List.class), "selected"+assetRel.
         getName() + "Ids");
      beanClass.addField(relSelectedProperty);
      ...
   } else {
      JavaField relProperty = new JavaField(JavaVisibility.PRIVATE,
         getModel().getType(assetRel.getReferredClass().getClass()),
         assetRel.getName());
      beanClass.addField(relProperty);
      ...
   }
}
```

Listing 5.3: Generating <*AssetBean*> properties from Asset relationships

As discussed in 4.1.4, one-to-one relationships are defined as single relation Asset type property holding a single instance of that Asset whereas a many-to-many relationship will be defined as a `List` holding a list relation Assets instances. Part of the implemented code is shown in Listing 5.3.

For the case of an inherited Asset, it will be implemented as discussed in section 4.1.5. The inherited Asset classes has a property that contains the super class as the parent Asset class. If the method `getSuperClass()` returns an Asset then the inheritance is implemented in a similar way as an Asset with the additional characteristics and relationships. Listing B.2 shows the complete generated code of the Asset class `Product` with characteristics, relationships as well as action methods for create, update and delete operations based on the discussion in section 4.1.6.

5.3 Generating Views with JSP and Faces UI components

Having the controller part being implemented, the next step is to implement the view that will be associated with it. The views are creates as JSP pages with Faces UI components and the access to beans are realized through JSF Expression Language. All the pages are generated in a similar fashion as to the generation of backing beans, that is by iterating over an array of Asset

classes. For every Asset class a set of methods are invoked to generate pages for the actions create, edit and view Asset. In general all pages follows a structure as listed below:

- All pages import the core JSF tag library which includes favious tag like `<f:view>` and various other tags.
- Standard HTML componnets are used, which are referenced by this library.
- All Faces tags are enclosed in a `<f:view>` tag enclosed with in a static header and footer followed by a `<h:form>` tag with a respective `id="formName"`.

Listing 5.4 shows the code to generate a create Asset page which is based on the structure discussed above. The edit Asset page is created in the same way except that it invokes the `generateFormJSP(ac, "update")` method and the view Asset page is created by invoking the `generateViewAssetJSP(ac)` method. The delete Asset action is embedded in the list all Assets page and will display a result page on successful execution or a error page on failure.

```
private void generateAssetJSPs() throws GeneratorException {
   for (AssetClass ac : getIm().getClasses()) {
     String createJSP = createTaglibDeclarations();
     createJSP += createHeaderJSP();
     createJSP += generateFormJSP(ac, "create");
     createJSP += createFooterJSP();
     writeJSP(createJSP, "create"+ac.getName());
     ...
   }
}
```

Listing 5.4: Generating a create page for an Asset

The characteristics of an Asset class are represented with the name of the characteristic and an `<h:inputText` tag, if its an create or edit Asset page or an `<h:outputText>` tag if its a view Asset page. In order to convert or validate the values entered by the user, an appropriate converter or validator has to be added here to the `<h:inputText>` or `<h:inputText>` tag with a `for` attribute representing the ID (see 3.4.1) of the property being handled. The `<h:message>` tag will display any conversion or validation errors to the user. On successful create or edit action a result page will be shown which could be redirected to the front page or the Asset list page. Listing 5.5 shows the way characteristics are added to a page.

```
for(Characteristic assetChar : ac.getCharacteristics()){
     JavaField assetCharField = currentAssetBeanClass.getField(assetChar
        .getName());
     String assetFieldName = assetCharField.getName();
     jsp +=" <tr>\n" +
          "    <td align=\"right\" width=\"100\">\n" +
```

```
          "    <h:outputText value=\""+assetFieldName+"\"/>\n" +
          "  </td>\n" +
          "  <td align=\"left\" width=\"400\">\n" +
          "     <h:inputText value=\"#{"+assetBeanName+"."+
             assetFieldName+"}\" id=\""+assetFieldName+"\" required=\"
             true\"/>\n" +
          "     <h:message for=\""+assetFieldName+"\" styleClass=\"
             errorMessage\"/>\n" +
          "    </td>\n" +
          "  </tr>\n";
}
```

Listing 5.5: Code snippet for adding a characteristic to a form page

The next step is to add relationship propterties to a page. As discussed in section 4.1.4 depending on the cardinality, a one-to-one relationship is directly associated with the page similar to the way a characteristic is represented. A many-to-many relationship has many possibilities to represent them. Considering the simplicity of the application in this implementation, a many-to-many relationship is represented as Combobox. Listing 5.6 shows the way relationships are generated on a page.

```
for(Relationship assetRel : ac.getRelationships()){
     ...
     jsp +="  <tr>\n" +
          "  <td align=\"right\" width=\"100\" valign=\"bottom\">\n" +
          "    <h:outputText value=\""+assetRelName+"\"/>\n" +
          "  </td>\n" +
          "  <td align=\"left\" width=\"400\">\n";
     String assetRelFieldName = assetRelField.getName();
     if(assetRelField.getType().getName().equals("List")) {
       jsp +="   <h:selectManyListbox value=\"#{"+assetBeanName+"."+
          assetRelFieldName+"}\" id=\""+assetRelFieldName+"\">\n" +
          "     <catalog:validateSelectedItemsRange minNum=\"1\"/>\n" +
          "     <f:selectItems value=\"#{"+assetBeanName+"."+
             selectedRelItems.getName()+"}\" id=\""+assetRelName+"s\"/>\
             n" +
          "    </h:selectManyListbox>\n";
     }
     jsp +="   <h:message for=\""+assetRelFieldName+"\" styleClass=\"
        errorMessage\"/>\n" +
          "     </td>\n" +
          " </tr>\n";
}
```

Listing 5.6: Code snippet for adding a relationship to a form page

The limitation in representing a many-to-many relationship in this way is that the Asset instances represented by `SelectItems` component should have a value that can be read by the

user. This means value of the key/value pair of this component should be a property of type `String`, like name property, rather than a number or price which would make less sense. There is no standard way to model this in ADL and this mapping is configured at the moment as a parameter in the generator configuration file B.1.

The views for an inherited Asset class are implemented the same way as of an Asset class, but along with its characteristics and relationships the ones from the inherited Asset class will also be implemented. The complete code of the product list, view and create product pages generated by this generator are listed in B.

5.4 Generating Configuration Files

Apart from generating backing beans and views, the configurations for the managed beans, and navigation rules are partially generated by the *WebAppGenerator* and integrated with the main *faces-config.xml* configuration file. The managed bean are configured in the *faces-managed-beans.xml* and the navigation rules are configured in the *faces-navigation.xml*. To generate the managed beans, for every Asset class, the symbol table is looked up to get the generated <*AssetBean*> and <*AssetLisBean*>s and registered as managed bean. Listing 5.7 shows the way <*AssetBean*>s are registered. The `moduleLocator` property is defined in the `BaseBean` and is common for all <*AssetBean*> and <*AssetLisBean*>s.

```
for (AssetClass ac : getIm().getClasses()) {
   JavaClass currentAssetBeanClass = (JavaClass) getWebST().
      getAssetBeanClassifier(ac);
   ...
   managedBean += "<managed-bean>" +
               " <managed-bean-name>"+assetBeanName+"</managed-bean-name
                  >" +
               " <managed-bean-class>"+getBeanPackage()+"."+
                  currentAssetBeanClass.getName()+"</managed-bean-class>
                  " +
               " <managed-bean-scope>request</managed-bean-scope> " +
               " <managed-property>" +
               "   <property-name>moduleLocator</property-name>" +
               "   <value>#{moduleLocatorBean}</value>" +
               " </managed-property>" +
               "</managed-bean>";
}
```

Listing 5.7: Generating managed beans to register <*AssetBean*> and <*AssetLisBean*>s

Similarly, configuring navigation cases are based on the symbol table and for every generated page a navigation case is added to the navigation rule. Listing 5.8 shows the generation of a navigation case for the create Asset page.

```
for (AssetClass ac : getIm().getClasses()) {
   String createAssetPageName = (JavaClass) getWebST().
      getCreateAssetPage(ac);
   ...
   navigationCase += "<navigation-case>" +
                "   <description>" +
                "   </description>" +
                "   <from-outcome>"+createAssetPageName+"</from-outcome
                   >" +
                "   <to-view-id>/"+createAssetPageName+".jsp</to-view-
                   id>" +
                "</navigation-case>";
   ...
}
```

Listing 5.8: Generating navigation case for create page

The navigation cases generated are for the general navigation cases and any specific navigation cases should be manually configured.

5.5 Symbol Table

The *WebAppGenerator* returns a symbol table which contain all information about the generated *<AssetBean>*s and *<AssetListBean>*s, its create, update, delete action methods as well as any search actions, generated pages and references to all other generated information. As seen in the discussions earlier, these information are used within the generator as well in order to create other parts of the code generation. The symbol table produced by this *WebAppGenerator* could be used by other CCM generators which provide other services, for example, a *ViewGenerator* that generates views based on another view technology or a *WebServiceGenerator*.

Chapter 6

Conclusion and Future work

In this chapter concludes with brief analysis of the work done, the limitations faced in realizing it practically and ideas for future work that can be done in order to improve it. Therefore, the generated web application interface is analyzed with respect to the transformation rules and decisions made in chapter 4.

6.1 Main Contribution

The aim of this thesis was to come up with ideas to model a web application interface for conceptual content management systems. The thesis was motivated by the fact that CCMSs provide a way to develop information systems that are open and dynamic, but have no means to provide a user interface to it. Web applications are built according to a layered architecture and is also based on a application domain model. Despite the separation of concerns through layers, the changes made to the domain model will affect the whole system. To ensure a seamless interaction to the changes made to the domain model, the frameworks used should be able to support the openness and dynamics of the CCM approach.

Today's most web applications are designed around the Model-View-Controller (MVC) pattern. It creates a decoupling between data access, data presentation and user interaction layers. However, to provide a web application interface to the CCMSs only the presentation layer has to be implemented. With respect to CCM, this means that the Model exists, only the View and the Controller has to be implemented by the generator. After analyzing the two Web application frameworks in chapter 3, JavaServes Faces has been chosen to be used for this thesis because of its UI framework including server side UI components, options for creating custom tags and components, server-side event processing and a tool support to simplify coding web-based applications.

A JSF application consists of JSP Pages with JSF components representing the user interface (View) and Backing Beans that hold the data (controller), a bean management facility and navigation rules. A web application interface with respect to these aspect were defined 4 and implemented 5 in order to map the Asset definitions to JSF components. The basic operations such as create, edit, delete and view instances of an Asset were realized. Views were built in which one or more views were associated with a one or more backing beans containing objects that represent Assets. There were elements that are implemented manually, for example, the `BaseBean` and other utility classes as well as and static JSP pages representing the front, header, footer and error pages. However, there were limitation mostly related to the layout of the user interface such as implementing the views in a user friendly manner and the way the Assets and its relationships were represented. The next section will discuss some of the limitations occurred implementing the web application generator.

6.2 Limitations

One of the limitations in during implementation of the generator is that implementing a selection list for a Asset relationship and associating it with a UI component, so that the user can select one or more Asset instances that belongs to that relationship type. As discussed 4.6, `categorySelectItems` was implemented as a a convenience property and associated with a `HtmlSelectManyListbox` UI component. In order to populate this list, a collection of `SelectItems` containing the instances of the relationship Asset has to be displayed. The limitation here was to select a characteristics, such as name, of the related Asset instance to display as key/value pair for the UI component. Currently, the characteristic that has to be use for building such collection is specified as a parameter (`<param name="categorySelectItemName">Category:name</param>`) in the generator configuration file.

Another general limitation is that to specify a Asset as a starting point of the application and also to specify a kind of navigation rule in order to control the application flow of the application. This is important if different users of the system want to have a different application flow or navigation structure of application. Currently only a general navigation rule is defined and applied to the whole application. Apart from this, there could be some mechanism to represent any type conversions or validations that are more specific to the entity in hand than to the data type used to represent it. Consider for example, the price of a product which could be a currency type or a value of a input field that could be validated for its length. Since these are considered more as commercial limitation and do not address the functionality of the implementation they are not considered in the scope of this project. However, availability of such a feature could be used to enhance the standard ones or develop custom validators and type converters.

Faces application doesn't have a built in security feature ad this would be a limitation if the application requires some authentication. Although, there are several ways to handle user authentication in JSF such as implementing a servlet filter to handle authentication checking, which

can be decoupled from the generated Web application and the security rules can be configured in a configuration file.

6.3 Future Work

As discussed earlier, there are certain limitations in representing conceptual part of an Asset to the user, which are mostly some kind of constraints in order to display the UI components associated with it. This can be addressed in future projects to make a proposal to realize a view or presentation model that will define additional parameters to represent an Asset in a UI based on the underlying technology.

In [24], an modeling approach has been introduced to provide interface definitions in order to model UI components that can be mapped to a target technology. It has options to define display constraints that offers support for dynamic binding of instance values of Assets to UI components. A extension of this could implement JSF UI components with the ability to configure its display attributes and use it while generating view for the web application interface. The study made in [24] recommends improvement for using a MVC based presentation layer for its UI modeling approach as well as event-based system which are available in JSF, both can benefit from each others advantages.

Alternatively, a lighter version of the UI modeling discussed above could be realized, some kind of declarative presentation model from which generators implementing a user interface can reason about and automate various aspects of interface design. The presentation model could be able to define the needed display parameter in terms of the Asset definitions rather than a target view technology model so that these information can be used by any generator implementing a user interface to CCMS. There exists model-based user interface paradigms [34] and [10], that use different models, such as Application model, Task model and Presentation model and provide tools for generating portions of the interface automatically and prototyping environments that can generate executable interfaces from a model.

Navigation, application flow and providing support for widget control are also some UI issues where future work can be done. Currently, only basic functionalities on Assets and its characteristics and relationships have been implemented. Constraints on Assets and Recursive Inheritance are not yet realized. This could be also a point to be considered in the future.

Appendix A

Developing Web Applications with Spring

This chapter gives an insight into web application development with Spring Web MVC discussed in section 3.3.2. The different components discussed in the following section shows how the model, view and controller interact with each other. An example application would demonstrate the features of Spring Web and provide a base in understanding which parts of it should be generated and which parts should be hand-coded. Consider a product catalog application to configure and implement the view layer with JSP and JSTL tags for rendering the output.

A.1 Configuring Spring MVC

The first things to do is to configure Spring MVC's `DispatcherServlet` that controls where all requests are routed based on the information that will be provided later. The configurations are registered in *web.xml* file as shown in the listing A.1. The Spring application context is loaded from a XML file named after the servlet-name with *-servlet* appended to it. This is a standard naming convention used in Spring. In our case, the `DispatcherServlet` will load its application context from the*catalogapp-servlet.xml* file.

```
<servlet>
  <servlet-name>catalogapp</servlet-name>
  <servlet-class>
    org.springframework.web.servlet.DispatcherServlet
  <servlet-class>
  <load-on-startup>1<load-on-startup>
<servlet>
```

Listing A.1: Configuring the Spring MVC DispatcherServlet

The next step is to configure the URL patterns that are used in this application, which is also a standard servlet-mapping entry A.2 in the *web.xml* file. In out case, any URL with an '.jsp' extension will be routed to the dispatcher.

```
<servlet-mapping>
   <servlet-name>catalogapp<servlet-name>
   <url-pattern>*.jsp</url-pattern>
</servlet-mapping>
```

Listing A.2: Configuring the URL mapping patterns

Next, register the configuration files that will be loaded by `ContextLoaderServlet` when the application is started. The listing A.3 below registers the `ContextLoaderServlet`. A `contextConfigLocation` parameter defines the location of the configuration files to be loaded.

```
<servlet>
  <servlet-name>context<servlet-name>
  <servlet-class>
    org.springframework.web.context.ContextLoaderServlet
  </servlet-class>
<load-on-startup>1</load-on-startup>
</servlet>

<context-param>
  <param-value>contextConfigLocation</param-value>
  <param-value>/WEB-INF/service-context.xml</param-value>
</context-param>
```

Listing A.3: Register ContextLoaderServlet and bean configuration files

The *service-context.xml* file contains the bean configuration of the services of the catalog application. Multiple configuration files, for example a separate *security-context.xml*, can be specified in the `<param-value>` tag by comma as delimiter.

A.2 Configuring application

The catalog application allows users to view and manipulate products in the catalog based on their authentication. The applications consists of a product page where the user can view products and also create, update or delete products if he is authenticated to do so. The authentication based on simple user login with a name and password.

```
<bean id="productListController"
    class="catalog.controller.ProductListController">
  <property name="commandName" value="productListCommand"/>
  <property name="commandClass" value="catalog.commands.
     ProductListCommand"/>
  <property name="validator" value="catalog.validators.
     ProductListValidator"
  <property name="formView" value="catalog" />
  <property name="successView" value="productList"/>
  <property name="catalogService">
      <ref bean="catalogService" />
  </property>
</bean>
```

Listing A.4: Register the ProductListController

Here for instance, the `ProductListController` extends the `SimpleFormController` to display all the products that belongs to a category. The controller allows to specify a `formView` property for the request page and a `successView` property for the page that has to be shown on successful execution. The names of the page maps to the actual view pages respectively. The `validator` gets control after the user submits the form and the values entered will be set on the command object by the framework. The `commandClass` and `commandName` determine the command object that will be active in those pages. The corresponding code of the `ProductListController` is shown in listing A.5.

```
public class ProductListController extends SimpleFormController {

 public ProductListController() {

 }

 protected ModelAndView onSubmit(Object command) throws Exception {
  ProductListCommand productListCommand = (ProductListCommand) command
     ;
  List<Product> productList = catalogService.searchByCategory(
     productListCommand.getCategoryId());
  return new ModelAndView(getSuccessView(), "productList", productList
     )
 }

 private CatalogService catalogService;

 public CatalogService getCatalogService() {
  return catalogService;
 }

 public void setCatalogService(CatalogService catalogService) {
  this.catalogService = catalogService;
```

```
  }
}
```

Listing A.5: ProductListController

The command object associated with this controller is the `ProductListCommand`. The `onSubmit` method gets control and calls the `searchByCatergory` method to get the list of products of the selected category. It then returns a `ModelAndView` passing a new instance using the URL to the `successView`.

Views in Spring are addressed by a *view resolver* and `InternalResourceViewResolver` is used in this example A.6 to resolve the logical name to an actual resource, in this case a JSP file that is */pages/catalog.jsp*.

```
<bean id="viewResolver"
    class="org.springframework.web.servlet.view.
       InternalResourceViewResolver">
  <property name="viewClass">
    <value>org.springframework.web.servlet.view.JstlView</value>
  </property>
  <property name="prefix" value="/pages/"/>
  <property name="suffix" value=".jsp"/>
</bean>
```

Listing A.6: Register the InternalResourceViewResolver

Finally, the `catalogService` that is wired into the `ViewProductController` has to be registered in the *service-context.xml* file mentioned earlier, which is then loaded in the *web.xml*.

```
<bean id="catalogService"
    class="catalog.services.CatalogService">
  <property name="catalogDao">
    <ref local="catalogDao"/>
  </property>
</bean>
```

Listing A.7: Register the catalog service

The `catalogService` takes a parameter to a data access object that enables access to the data to be displayed on the page. With controller and services configured the application is ready to be deployed.

Appendix B

Configuration and Generated Code Samples

The generator configuration file used by the compiler to run the `WebAppGenerator` is shown below.

```
<?xml version="1.0" encoding="UTF-8"?>

<cat
  xmlns:util="http://www.sts.tu-harburg.de/2004/java/util/xmlconfigfile"
     >
  <scanner class="de.tuhh.sts.cocoma.compiler.ADLScanner" />
  <parser class="de.tuhh.sts.cocoma.compiler.ADLParser" />
  <configuration name="webappgen">
    <param name="outputDirBase">target</param>
    <generator
      class="de.tuhh.sts.cocoma.compiler.generators.webapp.
         WebappGenerator"
      name="WebappGenerator">
      <param name="targetDirectory">
        <util:xpath path="../../../param[@name='outputDirBase']/text()" /
           >
      </param>
      <param name="applicationAsset">Category</param>
      <param name="categorySelectItemName">Category : name</param>
      <param name="webDirectory">web</param>
      <param name="sourceDirectory">java</param>
      <param name="beanPackage">de.tuhh.sts.cocoma.catalog.view.bean</
         param>
      <param name="moduleLocatorPackage">de.tuhh.sts.cocoma.catalog.view.
         modulelocator</param>
    </generator>

    <generator
```

```
    class="de.tuhh.sts.cocoma.compiler.generators.api.APIGenerator"
    name="apigen">
    <param name="outputDir">target/java</param>
    <param name="targetPackage">de.tuhh.sts.cocoma</param>
  </generator>
 </configuration>
</cat>
```

Listing B.1: WebAppGenerator Configuration

Listing B.2 below shows the code generated by the `WebAppGenerator` to create a backing bean for the `Product` class from the model `Catalog`. This code demonstrates all the aspects of code generation in transforming an Asset into a baking bean.

```
/*
 * Generated by de.tuhh.sts.cocoma.compiler.generators.webapp.
    WebappGenerator
 * Thu Oct 12 17:34:15 CEST 2006*/

package de.tuhh.sts.cocoma.catalog.view.bean;
public class ProductBean extends de.tuhh.sts.cocoma.catalog.view.bean.
   BaseBean {
  private de.tuhh.sts.cocoma.generic.ID productId;
  private java.lang.String name;
  private java.lang.String description;
  private int price;
  private java.util.Set categoryIds;
  private java.util.List selectedcategoryIds;
  private java.util.List categorySelectItems;
  private de.tuhh.sts.cocoma.generic.AssetClass assetClass;
  public ProductBean() {
    this.categorySelectItems = new java.util.ArrayList();
    this.assetClass = this.module.getClass("Product");
  }
  public de.tuhh.sts.cocoma.generic.ID getProductId() {
    return this.productId;
  }
  public void setProductId(de.tuhh.sts.cocoma.generic.ID productId) {
    this.productId = productId;
  }
  public java.lang.String getName() {
    return this.name;
  }
  public void setName(java.lang.String name) {
    this.name = name;
  }
  public java.lang.String getDescription() {
    return this.description;
  }
  public void setDescription(java.lang.String description) {
```

```
35    this.description = description;
36  }
37  public int getPrice() {
38    return this.price;
39  }
40  public void setPrice(int price) {
41    this.price = price;
42  }
43  public java.util.Set getCategoryIds() {
44    return this.categoryIds;
45  }
46  public java.util.List getSelectedcategoryIds() {
47    return this.selectedcategoryIds;
48  }
49  public java.util.List getCategorySelectItems() {
50    return this.categorySelectItems;
51  }
52  public void setCategoryIds(java.util.Set newcategoryIds) {
53    this.categoryIds = newcategoryIds;
54    if (this.categoryIds != null)
55      this.selectedcategoryIds = de.tuhh.sts.cocoma.catalog.view.util.
          ViewUtils.convertToList(this.categoryIds);
56  }
57  public void setSelectedcategoryIds(java.util.List newcategoryIds) {
58    this.selectedcategoryIds = newcategoryIds;
59    this.categoryIds = de.tuhh.sts.cocoma.catalog.view.util.ViewUtils.
        convertToSet(this.selectedcategoryIds);
60  }
61  public de.tuhh.sts.cocoma.generic.AssetClass getAssetClass() {
62    return this.assetClass;
63  }
64  protected void init() {
65    try {
66      if (this.productId != null) {
67        de.tuhh.sts.cocoma.catalog.Product product = (de.tuhh.sts.cocoma.
            catalog.Product) module.lookfor(this.productId);
68        org.apache.commons.beanutils.BeanUtils.copyProperties(this,
            product);
69      }
70      de.tuhh.sts.cocoma.generic.AssetClass categoryAssetClass = this.
          module.getClass("Category");
71      de.tuhh.sts.cocoma.generic.AssetIterator iter = module.lookfor(
          categoryAssetClass, new de.tuhh.sts.cocoma.generic.Module.
          QueryConstraint[]{});
72      while (iter.hasNext()) {
73        de.tuhh.sts.cocoma.catalog.Category category = (de.tuhh.sts.
            cocoma.catalog.Category) iter.next();
74        this.categorySelectItems.add(new javax.faces.model.SelectItem(
            category.getID(), category.getName()));
75      }
76    } catch (java.lang.Exception ex) {
```

```
77      java.lang.String errMsg = "Could not retrieve Product";
78      throw new javax.faces.FacesException(errMsg, ex);
79    }
80  }
81  public java.lang.String createAction() throws de.tuhh.sts.cocoma.
        catalog.model.exception.CatalogException {
82    de.tuhh.sts.cocoma.catalog.Product product = (de.tuhh.sts.cocoma.
        catalog.Product) module.create(this.assetClass,
83      new de.tuhh.sts.cocoma.generic.Module.MemberInitialization[]{});
84    if (product == null) {
85      de.tuhh.sts.cocoma.catalog.view.util.FacesUtils.addErrorMessage("
          Error creating Product");
86      return de.tuhh.sts.cocoma.catalog.view.bean.NavigationResults.RETRY
          ;
87    }
88    de.tuhh.sts.cocoma.catalog.MutableProduct mutableProduct = null;
89    try {
90      mutableProduct = product.lockAsProduct();
91    } catch (de.tuhh.sts.cocoma.generic.StateException se) {
92      de.tuhh.sts.cocoma.catalog.view.util.FacesUtils.addErrorMessage("
          Error locking Product");
93      return de.tuhh.sts.cocoma.catalog.view.bean.NavigationResults.
          FAILURE;
94    }
95    try {
96      mutableProduct.commitAsProduct();
97    } catch (de.tuhh.sts.cocoma.generic.StateException se) {
98      de.tuhh.sts.cocoma.catalog.view.util.FacesUtils.addErrorMessage("
          Error committing Product");
99      try {
100       mutableProduct.abortAsProduct();
101     } catch (de.tuhh.sts.cocoma.generic.StateException sex) {
102       sex.printStackTrace();
103     }
104     return de.tuhh.sts.cocoma.catalog.view.bean.NavigationResults.
          FAILURE;
105   }
106   try {
107     org.apache.commons.beanutils.BeanUtils.copyProperties(product, this
          );
108   } catch (java.lang.Exception ex) {
109     java.lang.String errMsg = "Could not save Product";
110     de.tuhh.sts.cocoma.catalog.view.util.FacesUtils.addErrorMessage(
          errMsg);
111     try {
112       mutableProduct.abortAsProduct();
113     } catch (de.tuhh.sts.cocoma.generic.StateException se) {
114       se.printStackTrace();
115     }
116     return de.tuhh.sts.cocoma.catalog.view.bean.NavigationResults.
          FAILURE;
```

```
117     }
118     de.tuhh.sts.cocoma.catalog.view.util.FacesUtils.getSessionBean().
            setProductId(this.productId);
119     de.tuhh.sts.cocoma.catalog.view.util.FacesUtils.resetManagedBean("
            productListBean");
120     java.lang.String errMsg = "Product with ID " + (this.productId + "
            was created successfully");
121     de.tuhh.sts.cocoma.catalog.view.util.FacesUtils.addInfoMessage(
            errMsg);
122     return de.tuhh.sts.cocoma.catalog.view.bean.NavigationResults.
            SUCCESS;
123   }
124   public java.lang.String deleteAction() {
125     try {
126       de.tuhh.sts.cocoma.catalog.Product product = (de.tuhh.sts.cocoma.
              catalog.Product) module.lookfor(this.productId);
127       module.delete(product);
128       de.tuhh.sts.cocoma.catalog.view.util.FacesUtils.resetManagedBean("
              productListBean");
129     } catch (java.lang.Exception ex) {
130       java.lang.String errMsg = "Could not delete Product";
131       de.tuhh.sts.cocoma.catalog.view.util.FacesUtils.addErrorMessage(
              errMsg);
132       return de.tuhh.sts.cocoma.catalog.view.bean.NavigationResults.
              FAILURE;
133     }
134     java.lang.String errMsg = "Product with ID " + (this.productId + "
            was deleted successfully");
135     de.tuhh.sts.cocoma.catalog.view.util.FacesUtils.addInfoMessage(
            errMsg);
136     return de.tuhh.sts.cocoma.catalog.view.bean.NavigationResults.
            SUCCESS;
137   }
138   public java.lang.String updateAction() {
139     try {
140       de.tuhh.sts.cocoma.catalog.Product product = (de.tuhh.sts.cocoma.
              catalog.Product) module.lookfor(this.productId);
141       module.modify(product, new de.tuhh.sts.cocoma.generic.Module.
              MemberInitialization[]{});
142       org.apache.commons.beanutils.BeanUtils.copyProperties(product, this
              );
143       de.tuhh.sts.cocoma.catalog.view.util.FacesUtils.resetManagedBean("
              productListBean");
144     } catch (java.lang.Exception ex) {
145       java.lang.String errMsg = "Could not update Product";
146       de.tuhh.sts.cocoma.catalog.view.util.FacesUtils.addErrorMessage(
              errMsg);
147       return de.tuhh.sts.cocoma.catalog.view.bean.NavigationResults.
              FAILURE;
148     }
```

```
149     java.lang.String errMsg = "Product with ID " + (this.productId + "
            was updated successfully");
150     de.tuhh.sts.cocoma.catalog.view.util.FacesUtils.addInfoMessage(
            errMsg);
151     return de.tuhh.sts.cocoma.catalog.view.bean.NavigationResults.
            SUCCESS;
152   }
153 }
```

Listing B.2: Generated backing bean of the `Product` Asset

The following three listings shows the generated code for JSP pages with UI components representing the view, create, edit, delete as well as listing a Asset. The edit page is same as the create page except the action method will be `editAction` instead of `createAction`.

```
1  <%-- Auto generated code. DO NOT EDIT --%>
2  <%-- Generated by de.tuhh.sts.cocoma.compiler.generators.webapp.
       WebappGenerator--%>
3  <%-- Thu Oct 12 17:34:16 CEST 2006--%>
4
5  <%@ taglib prefix="f" uri="http://java.sun.com/jsf/core"%>
6  <%@ taglib prefix="h" uri="http://java.sun.com/jsf/html"%>
7  <html>
8  <head>
9  <title>Front Page</title>
10 <link rel="stylesheet" type="text/css" href="stylesheet.css" />
11 </head>
12 <body>
13 <f:view>
14   <%@ include file="header.jsp"%>
15   <h:form id="productListBeanForm">
16     <table align="center">
17       <tr>
18         <td><h:dataTable id="table"
19           value="#{productListBean.currentProductBeans}" var="productBean"
                >
20           <h:column>
21             <f:facet name="header">
22               <h:outputText value="Product name" />
23             </f:facet>
24             <h:outputText value="#{productBean.name}" />
25           </h:column>
26           <h:column>
27             <f:facet name="header">
28               <h:outputText value="Product description" />
29             </f:facet>
30             <h:outputText value="#{productBean.description}" />
31           </h:column>
32           <h:column>
33             <f:facet name="header">
```

```
34            <h:outputText value="Product price" />
35          </f:facet>
36          <h:outputText value="#{productBean.price}" />
37        </h:column>
38        <h:column>
39          <f:facet name="header">
40            <h:outputText value="" />
41          </f:facet>
42          <h:commandLink action="viewProduct" styleClass="highLightLink"
              >
43            <h:outputText value="view" />
44            <f:param name="productId" value="#{productBean.productId}" />
45          </h:commandLink>
46        </h:column>
47        <h:column>
48          <h:outputText value=" | " styleClass="highLightText" />
49        </h:column>
50        <h:column>
51          <f:facet name="header">
52            <h:outputText value="" />
53          </f:facet>
54          <h:commandLink action="editProduct" styleClass="highLightLink"
              >
55            <h:outputText value="edit" />
56            <f:param name="productId" value="#{productBean.productId}" />
57          </h:commandLink>
58        </h:column>
59        <h:column>
60          <h:outputText value=" | " styleClass="highLightText" />
61        </h:column>
62        <h:column>
63          <f:facet name="header">
64            <h:outputText value="" />
65          </f:facet>
66          <h:commandLink action="#{productBean.deleteAction}"
67            styleClass="highLightLink">
68            <h:outputText value="delete" />
69            <f:param name="productId" value="#{productBean.productId}" />
70          </h:commandLink>
71        </h:column>
72      </h:dataTable></td>
73    </tr>
74   </table>
75  </h:form>
76  <%@ include file="footer.jsp"%>
77 </f:view>
78 </body>
79 </html>
```

Listing B.3: Generated list page of the `Product` Asset

```
<%-- Auto generated code. DO NOT EDIT --%>
<%-- Generated by de.tuhh.sts.cocoma.compiler.generators.webapp.
   WebappGenerator--%>
<%-- Thu Oct 12 17:34:15 CEST 2006--%>

<%@ taglib prefix="f" uri="http://java.sun.com/jsf/core"%>
<%@ taglib prefix="h" uri="http://java.sun.com/jsf/html"%>
<html>
<head>
<title>Front Page</title>
<link rel="stylesheet" type="text/css" href="stylesheet.css" />
</head>
<body>
<f:view>
 <%@ include file="header.jsp"%>
 <h:form id="createProductForm">
  <table align="center" width="500">
   <tr>
    <td></td>
    <td align="left"><h:outputText value="Create New Product"
     styleClass="headerText" /></td>
   </tr>
   <tr>
    <td align="right" width="100"><h:outputText value="name" /></td>
    <td align="left" width="400"><h:inputText
     value="#{productBean.name}" id="name" required="true" /> <h:
        message
     for="name" styleClass="errorMessage" /></td>
   </tr>
   <tr>
    <td align="right" width="100"><h:outputText value="description" /
       >
    </td>
    <td align="left" width="400"><h:inputText
     value="#{productBean.description}" id="description" required="
        true" />
    <h:message for="description" styleClass="errorMessage" /></td>
   </tr>
   <tr>
    <td align="right" width="100"><h:outputText value="price" /></td>
    <td align="left" width="400"><h:inputText
     value="#{productBean.price}" id="price" required="true" /> <h:
        message
     for="price" styleClass="errorMessage" /></td>
   </tr>
   <tr>
    <td align="right" width="100" valign="bottom"><h:outputText
     value="category" /></td>
    <td align="left" width="400"><h:selectManyListbox
     value="#{productBean.selectedcategoryIds}" id="
        selectedcategoryIds">
```

```
        <catalog:validateSelectedItemsRange minNum="1" />
        <f:selectItems value="#{productBean.categorySelectItems}"
          id="categorys" />
      </h:selectManyListbox> <h:message for="selectedcategoryIds"
         styleClass="errorMessage" /></td>
    </tr>
    <tr>
      <td align="center" colspan="2"><h:commandButton value="Create"
        action="#{productBean.createAction}" /> <h:commandButton
        value="Cancel" action="cancel" immediate="true" /></td>
    </tr>
    <tr>
      <td align="left" colspan="2"><h:messages
        styleClass="errorMessage" globalOnly="true" /></td>
    </tr>
   </table>
  </h:form>
  <%@ include file="footer.jsp"%>
</f:view>
</body>
</html>
```

Listing B.4: Generated create page of the `Product` Asset

```
<%-- Auto generated code. DO NOT EDIT --%>
<%-- Generated by de.tuhh.sts.cocoma.compiler.generators.webapp.
   WebappGenerator--%>
<%-- Thu Oct 12 17:34:15 CEST 2006--%>

<%@ taglib prefix="f" uri="http://java.sun.com/jsf/core"%>
<%@ taglib prefix="h" uri="http://java.sun.com/jsf/html"%>
<html>
<head>
<title>Front Page</title>
<link rel="stylesheet" type="text/css" href="stylesheet.css" />
</head>
<body>
<f:view>
  <%@ include file="header.jsp"%>
  <h:form id="viewProductForm">
   <table width="800" border="0" align="center" bgcolor="#FFFFFF"
    cellpadding="0" cellspacing="0">
    <tr>
      <td colspan="2"> </td>
    </tr>
    <tr>
      <td width="460">
      <table width="460">
        <tr>
          <td align="center"><h:graphicImage
```

```
26            url="images/products/#{productBean.productId}.jpg" /></td>
27          </tr>
28        </table>
29        </td>
30        <td width="340" valign="top">
31        <table border="0" bgcolor="#FFFFFF">
32          <tr>
33            <td><h:outputText value="Product name" /></td>
34            <td><h:outputText value="#{productBean.name}" /></td>
35          </tr>
36          <tr>
37            <td><h:outputText value="Product description" /></td>
38            <td><h:outputText value="#{productBean.description}" /></td>
39          </tr>
40          <tr>
41            <td><h:outputText value="Product price" /></td>
42            <td><h:outputText value="#{productBean.price}" /></td>
43          </tr>
44        </table>
45        </td>
46      </tr>
47      <tr>
48        <td colspan="2" bordercolor="#FFFFFF" bgcolor="#FFFFFF">
49        <hr width="760" color="#CCCCCC" />
50        </td>
51      </tr>
52      <tr>
53        <td colspan="2"><%@ include file="navigation.jsp"%>
54        </td>
55      </tr>
56    </table>
57  </h:form>
58  <%@ include file="footer.jsp"%>
59 </f:view>
60 </body>
61 </html>
```

Listing B.5: Generated view page of the `Product` Asset

Bibliography

[1] ABRAMS, M., PHANOURIOU, C., BATONGBACAL, A. L., WILLIAMS, S. M., AND SHUSTER, J. E. Uiml: An appliance-independent xml user interface language. *Computer Networks 31*, 11-16 (1999), 1695–1708.

[2] APACHE STRUTS FOUNDATION. Apache Struts Framework. *http://struts.apache.org/*, 2006. Framework Specification.

[3] BASS, L. J., AND UNGER, C., Eds. *Engineering for Human-Computer Interaction, Proceedings of the IFIP TC2/WG2.7 Working Conference on Engineering for Human-Computer Interaction, Yellowstone Park, USA, August 1995* (1996), vol. 45 of *IFIP Conference Proceedings*, Chapman & Hall.

[4] BOSSUNG, S., SEHRING, H.-W., HUPE, P., AND SCHMIDT, J. W. Open and Dynamic Schema Evolution in Content-Intensive Web Applications. In Cordeiro et al. [7], pp. 109–116.

[5] BROY, M., AND ZAMULIN, A. V., Eds. *Perspectives of Systems Informatics, 5th International Andrei Ershov Memorial Conference, PSI 2003, Akademgorodok, Novosibirsk, Russia, July 9-12, 2003, Revised Papers* (2003), vol. 2890 of *Lecture Notes in Computer Science*, Springer.

[6] BULLARD, V., SMITH, K. T., AND DACONTA, M. C. *Essential XUL Programming*. Wiley, 2001.

[7] CORDEIRO, J. A. M., PEDROSA, V., ENCARNAÇÃO, B., AND FILIPE, J., Eds. *WEBIST 2006, Proceedings of the Second International Conference on Web Information Systems and Technologies: Internet Technology / Web Interface and Applications, Setúbal, Portugal, April 11-13, 2006* (2006), INSTICC Press.

[8] CZARNECKI, K., AND EISENECKER, U. W., Eds. *Generative and Component-Based Software Engineering, First International Symposium, GCSE'99, Erfurt, Germany, September 28-30, 1999, Revised Papers* (2000), vol. 1799 of *Lecture Notes in Computer Science*, Springer.

[9] CZARNECKI, K., EISENECKER, U. W., GLÜCK, R., VANDEVOORDE, D., AND VELDHUIZEN, T. L. Generative Programming and Active Libraries. In Jazayeri et al. [17], pp. 25–39.

[10] DA SILVA, P. P., GRIFFITHS, T., AND PATON, N. W. Generating user interface code in a model based user interface development environment. In *Advanced Visual Interfaces* (2000), pp. 155–160.

[11] DAVID, J.-L., RYAN, B., DESERRANNO, R., AND YOUNG, A. *Professional WinFX Beta: Covers "Avalon" Windows Presentation Foundation and "Indigo" Windows Communication Foundation*. Wrox, 2005.

[12] DIX, A., FINLEY, J., ABOWD, G. D., AND BEALE, R. *Human Computer Interaction*. Prentice Hall, 2003.

[13] FOWLER, M. Inversion of Control Containers and the Dependency of Injection pattern. http://www.martinfowler.com/articles/injection.html (January 2004).

[14] GAMMA, E., HELM, R., JOHNSON, R., AND VLISSIDES, J. M. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, October 1994.

[15] GOTTLOB, G., BENCZÚR, A. A., AND DEMETROVICS, J., Eds. *Advances in Databases and Information Systems, 8th East European Conference, ADBIS 2004, Budapest, Hungary, September 22-25, 2004, Proceesing* (2004), vol. 3255 of *Lecture Notes in Computer Science*, Springer.

[16] JAVA COMMUNITY PROCESS. JavaServer Faces Specification. *http://www.jcp.org/en/jsr/detail?id=127*, 2006. Java Technology Specifications.

[17] JAZAYERI, M., LOOS, R., AND MUSSER, D. R., Eds. *Generic Programming, International Seminar on Generic Programming, Dagstuhl Castle, Germany, April 27 - May 1, 1998, Selected Papers* (2000), vol. 1766 of *Lecture Notes in Computer Science*, Springer.

[18] JOHNSON, R. *Expert One-on-One: J2EE Design and Development*. Wrox, 2002.

[19] JOHNSON, R. Introduction to Spring Framework. http://www.theserverside.com/tt/articles/article.tss?l=SpringFramework (July 2005).

[20] JOHNSON, R., ET AL. Spring Java/J2EE Application Framework. *http://www.springframework.org/*, 2006. Framework Specification.

[21] KUNII, H. S., JAJODIA, S., AND SØLVBERG, A., Eds. *Conceptual Modeling - ER 2001, 20th International Conference on Conceptual Modeling, Yokohama, Japan, November 27-30, 2001, Proceedings* (2001), vol. 2224 of *Lecture Notes in Computer Science*, Springer.

[22] MANN, K. D. *JavaServer Faces in Action*. Manning, 2004.

[23] MERRICK, R., WOOD, B., AND KREBS, W. Abstract User Interface Markup Language. In *Workshop on developing User Interfaces with XML: Advances on User Interface Description Languages* (2004).

[24] MOFOR, G. N. Modeling of User Interfaces for Conceptual Content Management Systems. Projektarbeit, STS, TU Hamburg-Harburg, August 2006.

[25] OPENSYMPHONY. WebWork Web Application Framework. *http://www.opensymphony.com/webwork/*, 2006. Framework Specification.

[26] SAVOLSKYTE, J. Conceptual Content Management Application Development by means of Storyboarding. Master thesis, TU Hamburg-Harburg, August 2006.

[27] SCHMIDT, J. W., AND SEHRING, H.-W. Conceptual Content Modeling and Management. In Broy and Zamulin [5], pp. 469–493.

[28] SEHRING, H.-W. Compiler Framework and Generator Development Guide. *http://www.sts.tu-harburg.de/ hw.sehring/cocoma/projs/compiler/Compiler _Framework.pdf*, September 2004. Framework Specification.

[29] SEHRING, H.-W. *Konzeptorientierte Inhaltsverwaltung Modell, Systemarchitektur und Prototypen*. Doctoral thesis, TU Hamburg-Harburg, 2004.

[30] SEHRING, H.-W. Java Code Generation Toolkit. *phhttp://www.sts.tu-harburg.de/ hw.sehring/codegentk/doc/index.html*, 2006.

[31] SEHRING, H.-W., AND SCHMIDT, J. W. Beyond Databases: An Asset Language for Conceptual Content Management. In Gottlob et al. [15], pp. 99–112.

[32] SMARAGDAKIS, Y., AND BATORY, D. S. Scoping Constructs for Software Generators. In Czarnecki and Eisenecker [8], pp. 65–78.

[33] SUN MICROSYSTEMS. JavaServer Faces. *http://java.sun.com/javaee/javaserverfaces/*, 2006.

[34] SZEKELY, P. A., SUKAVIRIYA, P. N., CASTELLS, P., MUTHUKUMARASAMY, J., AND SALCHER, E. Declarative interface models for user interface construction tools: the MASTERMIND approach. In Bass and Unger [3], pp. 120–150.

[35] THALHEIM, B., AND DÜSTERHÖFT, A. Sitelang: Conceptual Modeling of Internet Sites. In Kunii et al. [21], pp. 179–192.

[36] WIEDERHOLD, G. Mediators in the architecture of future information systems. *IEEE Computer 25*, 3 (1992), 38–49.