Bachelor Thesis

CUSTOM COMPONENT DEVELOPMENT For a DIGITAL LIBRARY WITH JAVASER FACES

Che Maabo Jomia Blaise

Information technology

Technical University Hamburg Harburg

Supervised by:

Prof. Dr. Ralf. Möller

Dipl.-Inf. Patrick Hupe

Software Systems Department

Technical University Hamburg-Harburg

March 2007

Acknowledgements

I sincerely would like to thank Prof. Dr. Ralf. Möller for the interesting subject and the supervision of this work.

Special thanks go to Dipl.-Inf. Patrick Hupe for his outstanding support, he has been great and always took very much time for the supervision of this work.

Further I would like to thank all those who contributed to the success of this work.

Table of Contents

1 Introduction **1**

1.1 Motivation 1
1.2 Objectives 2
1.3 Structure 2

2 Application Architectural Models **3**

2.1 Single-tier model 3
2.2 Evolution towards multi-tier models 3
  2.2.1 Two-Tier Client/Server Architectures 4
  2.2.2 Three-Tier Client/Server Architectures 4
2.3 Introducing Component-Oriented Modelling 4
2.4 Component-Oriented Multi-Tier Architectures 4
  2.4.1 Presentation Layer 4
  2.4.2 Business Layer 4
  2.4.3 Data Access Layer 4
  2.4.4 Data Store 5

3 Presentation Layer Models and Frameworks **7**

3.1 Presentation Layer Models 7
  3.1.1 Model View Controller Pattern 7
  3.1.2 Model 1 8
  3.1.3 Model 2 9
3.2 Web Application Frameworks 9
  3.2.1 Application Frameworks 10
  3.2.2 User Interface Frameworks 10

4 The JavaServer Faces Component Framework **11**

4.1 Introduction 11
4.2 Design Goals 12
  4.2.1 User Interface Development 12
  4.2.2 Navigation 12
  4.2.3 Session and Object Management 12
  4.2.4 Validation and Error Feedback 12
  4.2.5 Internationalization 12
  4.2.6 Custom Component 13
4.3 MVC-Architecture of JavaServer Faces 13
4.4 Lifecycle and Event Model 14

4.5 UI Component Model 17
4.5.1 UI Component Classes 17
4.5.2 Events and Listeners 17
4.5.3 Converters 18
4.5.4 Validators 19
4.5.5 Renderer and Renderkit 20

5 Case Study: HMI Digital Library System 23

5.1 Analysis 23
5.2 Design 24
5.2.1 High-level solution 24
5.2.2 Technical solution using JSF 24
5.3 Creating a custom tag handler and a custom component 25
5.3.1 Create a custom tag that subclasses UIComponentTag 25
5.3.2 Extending UIComponent 25
5.4 Implementation 26
5.5 Evaluation 29

6 Summary and Outlook 31

6.1 Summary 31
6.2 Outlook 31

References 33

CHAPTER 1

Introduction

In the last years, user-centric web applications have grown from small useful applications to indispensable and well established entities in the Internet. Server side programming has been leveraged by a diversity of programming language and scripts. Over the years server side technologies have been battling to get the favour of web developers, Perl, Python, PHP, Ruby and of course Java just to name a few. No wonder that with the increase in the number of technologies the development of high dynamic web-application have become even more complex.

Based on the Java Platform, Sun Microsystems provides specifications that define uniform standards for application development. This has lead to a good number of web applications that nowadays exist on Intranet and Internet. These web applications vary from simple applications such as a calculator to complex business processes such as a Bookstore.
Java is widely used for server-side Web development and Servlets and JavaServer pages (JSP) allow you to create dynamic data-driven Web applications.

The Servlet technology and JavaServer Pages (JSP) are traditional technologies for building Java Web applications. JavaServer faces doesn't only take over the challenges that are to be programmed in a Servlet or JSP application but also defines a component model for User Interface element for Java Web applications. JavaServer Faces as defined in its specification is to be regarded as the de facto standard for Graphical User Interface (GUI) for Java Web applications. It is the modern, component based approach, which aim is to significantly ease the burden of writing and maintaining applications that run on a Java application server and render their User Interfaces back to a target client.

As the Model-view-Controller (MVC) design pattern still prevails in the development of graphical user interfaces and is doesn't come as a surprise that JSF also implements this design pattern in UI component development. This is similar to Swing that defines a component model for local Graphical User Interfaces (GUI). Thus components exist for user input, selection or command user interfaces. These components analogue to swing can also be hierarch ally ordered

1.1 Motivation

In fact, this idea has pushed developers to think of new technologies that provide a clear separation of business logic and presentation. Recently, new technologies and concepts appeared in the field of enterprise applications based on the Enterprise Edition of the Java 2 Platform (J2EE).
With the advance of the Internet and especially the World Wide Web in the early 1990s and its increasing importance in *Business-to-Customer* and *Business-to-Business* services, the

need to define standard methodologies to design and build efficient, robust and maintainable Web applications has become increasingly important.

Not only that, there is the need to build robust User Interface Components (UI) smart enough to manage their state and assume more complex functionally such as event handling and rendering. In this context Sun Microsystems offers JavaServer Faces technology (JSF), the state of the art Framework for User Interface development on the Java platform.

1.2 Objectives

This thesis introduces the JavaServer Faces component framework and its usage in the development of custom UI Components. This thesis goal is to apply the concept brought in by JSF in the HMI System. The HMI System is a digital library for high school teachers in Hamburg. The library's goal is to support the collection, classification, recommendation and reuse of high quality teaching material.
For that, analysis of the business process will be carried out in order to determine the requirements to the User Interfaces. Here the User Interfaces must meet the requirements resulting from business operations as well as consider technical concepts of the HMI System.

1.3 Structure

This document is organized into six chapters, each having their particular relevance according to the goal of understanding application architecture models and the JavaServer Faces framework in particular and developing user interface components.

The second chapter gives an introduction to the client/server model and its evolution from centralized architecture to a multi-tier, component-oriented architecture.

The third chapter is about the Java Web application, the MVC design pattern in Web applications, the different model approach to building web applications and an introduction into Web application Frameworks.

The fourth chapter introduces the JavaServer Faces technology, its design goals, the architecture, its lifecycle and its component model.

The fifth chapter introduces the HMI System as an application for JavaServer Faces. It shows how UI component implement business processes in the HMI System with the help of JSF custom component.

The sixth chapter finally gives a summary of the Thesis as well as an outlook on further development using JavaServer Faces.

CHAPTER 2

Application Architectural Models

This chapter gives an introduction to software application architectural models and the evolution from centralized models to multi-tier models. It further briefly describes the component oriented model and its reflection in a modern application architecture.

2.1 Single-tier model

In the beginning of computing era, the model was pretty much a single tier model in which dumb terminals are directly connected to a mainframe. Here the mainframe is centralized point of computing intelligence in which presentation, business logic, and data access are all intertwined in a single monolithic mainframe application. In this tier model, because the clients are dumb terminals and do not have any processing logic what so ever, no client side management is required. Data consistency is easy to achieve since data access logic is in complete control of the mainframe application.
The drawback of the one tier model is that there is no separation among presentation, business logic, and data access. This doesn't facilitate code re-use because all functionality is mixed together. Changing data or business logic may affect every part of the application, making changes (for example, adding new functionality or bug fixes) difficult task.

2.2 Evolution towards multi-tier models

A first step away from the drawbacks of centralized architectures was the introduction of the client/server model. The *client/server model* is a distributed system architecture model where functionality is split into a client and a server application part. The client/server software architecture was intended to improve scalability as compared to the centralized model as several clients can work independently and access a single server when required. A single machine can be both a client and a server depending on the software configuration.
Functionality of an application is split into one or more tiers or layer. Typically, the presentation logic layer, the business logic layer and data layer are being identified as the main application layer.
Typically, data is being managed by the server part and provided to one or more client(s) upon request. Business logic and presentation logic can be handled either by the server completely (which results in ‘dumb clients’, e.g. terminal client system), or by the client completely (which makes the server a simple data-server). In most cases though, presentation logic is handled by the client and the business logic is distributed among client(s) and server. The models will be described in detail in the following sections.

2.2.1 Two-Tier Client/Server Architectures

In two-tier model, fat clients are talking to backend database server by using some database access protocol. The clients are called "fat" clients because the clients have to maintain presentation logic, business logic, and have to have detailed understanding on data model of the backend data and how to access it. In this model database independence compared to one-tier model and a standard database access protocol using SQL is achieved.
Yet it had significant disadvantages. The presentation, business logic and data model are now intertwined now at the client side thus making maintenance and updates difficult.

2.2.2 Three-Tier Client/Server Architectures

The three-tier architectural model takes it a step further. In a three-tier model (also referred to as multi-tier model), the client implements the presentation logic only (thin client). The business logic is implemented on an application server(s) and the data resides on database server(s).

2.3 Introducing Component-Oriented Modelling

Component-oriented modelling is based on object-oriented modelling and shares its principles of encapsulation and separation of responsibilities. It furthermore introduced the notion of building software from re-usable units, the *components*, which provide clearly defined sets of functionality and dependencies. Components expose their functionality through clearly defined interfaces and thus implementations should theoretically be interchangeable. Examples for components are XML-Parsers and search engines. Components can be hosted in a so-called 'container' that is responsible for managing the components' lifecycle, and facilitates communication between components.

2.4 Component-Oriented Multi-Tier Architectures

Component-oriented modelling can be applied to multi-tier architectural models: Applications following a component-oriented model structure decompose layer functionality into components that have clearly defined responsibilities and interact with one another.

Nowadays, many enterprise applications follow the sketched architecture. In enterprise application systems, the following components can typically be found:

2.4.1 Presentation Layer: The presentation layer contains components needed to interact with the user of the application. This layer usually consists of a GUI or Browser type of interface. Presentation layer components are web page and content item templates, rich-client forms, user interaction process components, e.g. user input validators.

2.4.2 Business Logic Layer: This layer contains the definition of the application domain, consisting of business entities and business processes that run on them. Components are process execution engines (workflow engines), search engines. The business components are generally front-ended by a service interface that acts as a facade to hide the complexity of the business logic from the presentation layer.

2.4.3 Data Access Layer: The data access layer abstract from the underlying data access technology thus allowing the business layer to focus on business logic. Each component

typically provides methods to perform Create, Read, Update, and Delete (CRUD) operations for business entities.

2.4.5 Data Stores: Applications store their data in one or more data stores. Relational databases, XML databases and file systems are very common types of data stores.

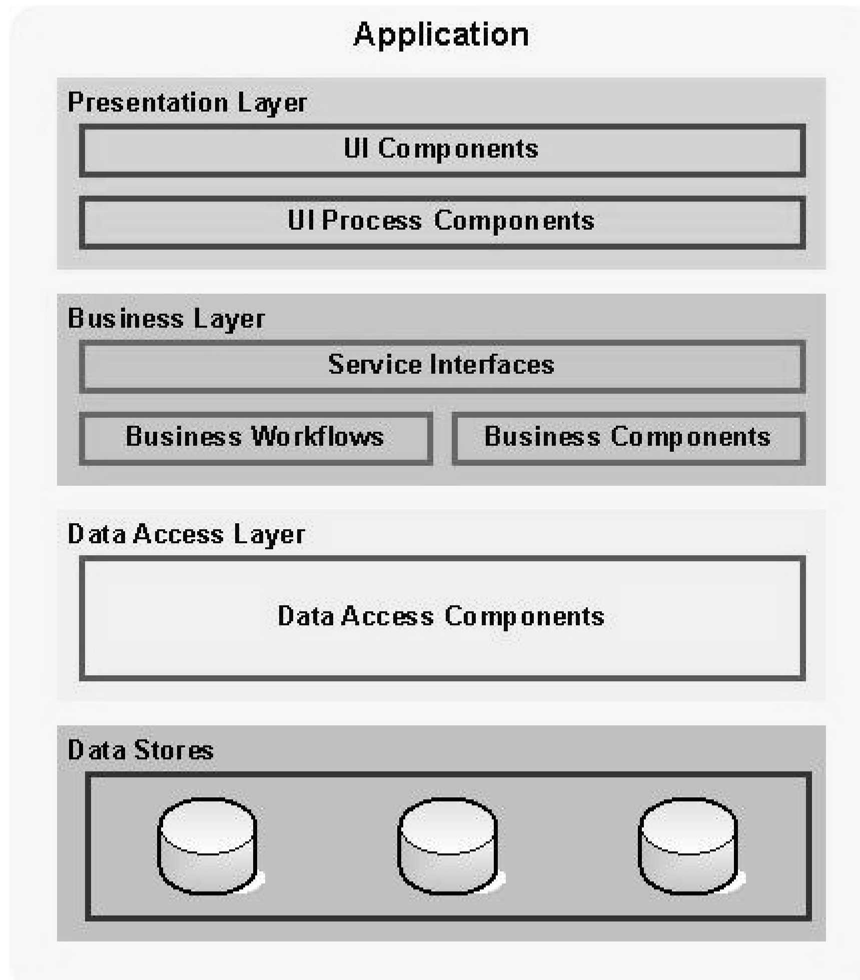

Figure 2.1: Component-oriented, multi-tier application architecture

CHAPTER 3

Presentation Layer Models and Frameworks

This chapter introduces Java-based web applications presentation layer models and frameworks. The goal of this chapter is to provide a basic knowledge about Java web application presentation models and common architectural patterns for presentation layer models is given. Restrictions of known Java presentation layer models are discussed, and an introduction to Java-based web application frameworks is give which help developers overcome model shortcomings.

3.1 Presentation Layer Models

Two main approaches for designing the presentation layer of Java-based web applications have been evolved in the past. The traditional design is called the Model 1 approach [Sb03]. Current web applications follow the Model 2 design [Sb03], which is an adapted version of the Model-View-Controller (MVC) design pattern [GHJV95] for server applications. Since Model 2 applications are complex to implement, web application frameworks provide a Model 2 implementation to relieve the developer from the task of implementing a complete Model 2 architecture.

3.1.1 Model-View-Controller Pattern

The Model-View-Controller (MVC) architecture, which has its background in the Smalltalk environment, provides design patterns for developing Graphical User Interfaces (GUIs). Application of this architecture leads to a strict separation of the following components:

The model represents the business logic and the underlying data model of the system. It is usually a software approximation of a real-world process. A clear separation of the model from the presentation logic is essential in order to be able to use different viewing technologies on the same data.

A view is responsible for rendering the model and displaying data to the user. Additionally it has to keep the displayed data consistent to the underlying data model. Therefore the model notifies the view of changes in the model.

The controller acts as the mediator between model and view. It is in charge of processing user events and it maps these events to updates on the model. Based on user actions or results of model updates, the next view is selected by the controller.

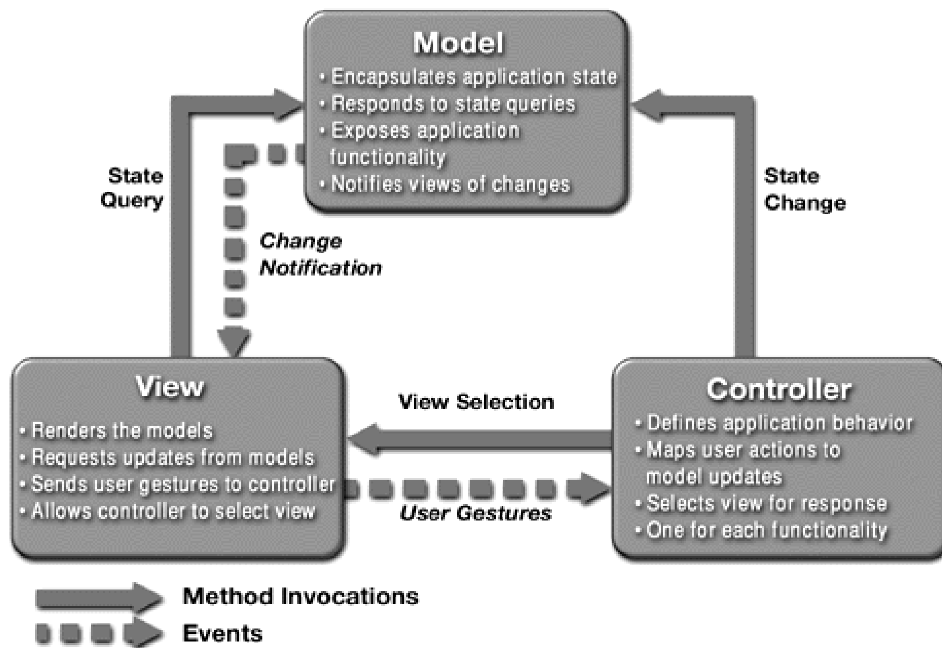

Figure 3.1: This picture shows MVC pattern, the Model, the View, and the Controller

This leads to an application consisting of loosely coupled components. They have very few dependencies that allow separate development and unit testing of model, view and controller. Better maintainability and scalability are also benefits of this approach.

3.1.2 Model 1

The Model 1 approach [Sb03] shown below is the traditional design for Java web applications. All business logic is hard coded into the JSP in form of JavaBeans, Scriptlets or expressions. A web browser accesses the JSP directly and the JSP use JavaBeans, which represent the model, to access the data.
The current view (or more specific the current JSP) determines the next view.

The page navigation is controlled through hyperlinks or request parameters inside each JSP. There is no centralized controller where page navigation is controlled or the request parameters are processed, therefore Model 1 is a ”decentralized” or ”page centric” approach. Each JSP has to process its own request data, which leads to a lot of Java code inside the JSP.
This model is usually used for small web applications with few user interaction and simple page navigation

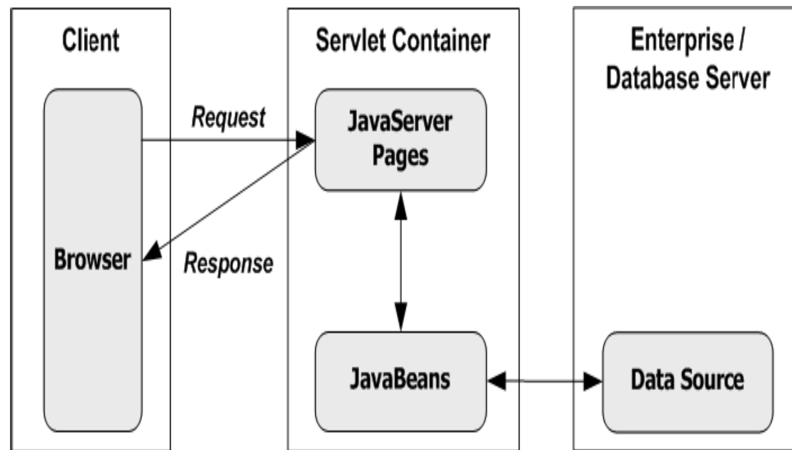

Figure 3.2: Model 1 Architecture

3.1.3 Model 2

The Model 2 design [Sb03] introduces a front controller Servlet as a centralized component that processes the clients' requests. It processes the request data sent by the client and selects the next view according to request parameters or state of the model. Components of the view do not refer directly to each other. Since the front controller Servlet is a single point of entry into the web application, it may implement security and logging functions as well. The view can be represented by any presentation technology like JSP or a XML/XSLT approach. JavaBeans or Enterprise JavaBeans (EJB) can for example represent the model. This design is also called a "centralized" or "Servlet centric" approach.
Most Java web application frameworks are based on the Model 2 architecture shown in 3.3 or on a slight variation.

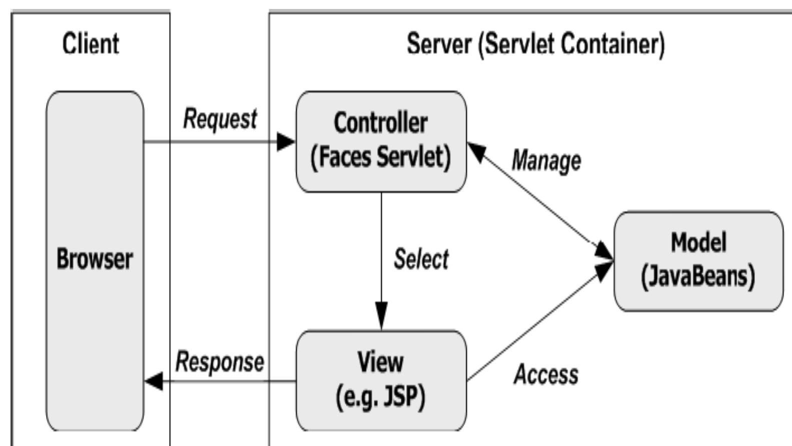

Figure 3.3: Model 2 Architecture

3.2 Web Application Frameworks

A web application developer has to face many complex tasks, especially if the web application is developed using a pure Servlet or JavaServer Pages approach. Many of these tasks cannot be handled well with this approach and result in a lot of coding work which also

repeats itself frequently, for example handling of form data or localization. For small web applications this might be bearable.
But the situation gets worse when the applications become larger and more complex.
Many Java web application frameworks evolved to simplify Java web development with providing a set of common features. The following tasks give an introduction to the functionality of web application frameworks.

Most of the development work of a web application results from form processing. The HTML pages of a web application contain HTML form elements such as text fields, checkboxes or comboboxes. When a form is submitted the developer has to write code that retrieves the values from each form element and verifies that the entered data is valid. Additionally, the application has to interact with the data model of the application.

Another point is the management of the page structure, since the JavaServer Pages technology does not provide a good support for building a page out of several reusable parts.

Providing a localized web application according to the client's language preference is not natively supported by Servlets or JavaServer Pages. The developer has to take care that every text output is rendered according to the right language. Some frameworks provide this feature as well as the JavaServer Pages Standard Tag Library.

3.2.1 Application Frameworks

The goal of an application framework, also referred to as foundation framework, is to support the development of a complete web application. Consequently it does not care about specific details of the application, for example how the user interface is rendered. It therefore requires a separate presentation layer technology like JavaServer Pages for rendering a response to the client. Application Frameworks do not distinct between user actions that only affect the user interface, for example a selection in drop-down Listbox results in displaying an additional text field, and actions that result in calls to the backend code of the application. They provide features including simplified form processing, an architecture based on the Model 2 or a slight variation, integration of data sources and control of all application resources in a centralized configuration file. Although they provide a lot of features, they do not mask the underlying stateless request-response model of HTTP to the developer. Examples of this category of frameworks include Webwork and the very popular Struts framework.

3.2.2 User Interface Frameworks

User interface frameworks focus on user interface details and do not focus on the implementation of complete applications like application frameworks do. In contrast to the application frameworks, user interface frameworks hide the underlying stateless request-response model of HTTP by providing a stateful component and event model that brings web user interfaces closer to the programming model introduced by Java Swing or visual development tools like Delphi or Visual Basic. A user interface framework therefore significantly simplifies the entire programming model.

Especially the processing of form data is simplified, since the developer does not have to handle the request parameters on the level of the Servlet API.
Examples for user interface Frameworks are Tapestry and JavaServer Faces, whose architecture is described in detail in chapter 4.

CHAPTER 4

The JavaServer Faces Framework

After discussing the fundamental technologies, related problems of Java web applications presentation layer models and frameworks; this section focuses on a concrete web application framework, JavaServer Faces (JSF) [SM04a]. The explanations given in this section are also the fundamental knowledge needed to understand the implementation of custom UI Components which is introduced in chapter 5 as an example of custom component development with JavaServer Faces.
The first part describes the JavaServer Faces and its design goals. It also addresses the architecture and explains the way in which the MVC architecture introduced in the previous section is applied. This section then introduces a detailed explanation of the request processing lifecycle. The lifecycle defines several steps in processing a client's request and is a major concept of JSF. Finally, the key components and features of the framework along with the problems that are to be solved by the particular components is given.

4.1 Introduction

JavaServer Faces is a user interface framework that has been designed to significantly ease the development of web based user interfaces by providing an event-driven component model. JSF defines a number of events that are close to the events provided by a traditional GUI. JSF still deals with the stateless HTTP protocol on a low level, but masks this as much as possible to the developer by creating different events automatically according to the request data received. JSF provides support for rich, powerful user interface components tied to a well-defined request processing lifecycle.
JSF UI components are stateful, which means that components keep their state and value across multiple client requests. When using HTML/JSP as the viewing technology for JSF,
UIcomponents are mostly a one-to-one mapping to the HTML form elements. This component model also opens up the opportunity for the appearance of RAD-Tools that allow the development of pages simply by arranging user interface components in a visual development tool. Therefore the page designer can work with components that encapsulate the user interface without the need of embedded Java source code.
Another benefit of the Model 2 or MVC architecture that JSF uses is the clear separation of presentation, logic and data. If modifications to one of these parts need to be made, they do not necessarily lead to changes in the other parts. This is also a significant advantage when a team of developers is working on a web application, since every developer can focus on his/her core competence. So a page designer can focus on the page layout and does not have to implement Java code.
It is important to mention that JavaServer Faces is a specification provided by Sun Microsystems that may result in several implementations by different vendors. The specification was officially released in March 2004.

It is a server side user interface component framework for Java technology based Web applications. It is server side instead of client side framework. What this means is that, in JSF architecture, many things that are related to UI management are handled at the server instead of at client side. The prime example of client side UI framework is Swing. Added to what we have just mentioned, JSF is a UI component framework, meaning under JSF architecture,

UI is handled by a set of UI components. The concept of UI component is very important in understanding JSF.
In more technical terms, JSF is a specification and reference implementation for Web application development framework. And the specification defines various things such as
UI component model, event and listener model, validator model, back-end data integration model.

4.2 Design goals

JavaServer Faces promises to bring rapid user-interface development to server-side Java. It allows developers to painlessly write server-side applications without worrying about the complexities of dealing with browsers and Web servers. It also automates low-level, boring details like control flow and moving code between web forms and business logic.
Java server Faces (JSF) simplifies the construction of the Presentation layer of web applications. Developers can put together reusable UI Components to generate web pages. The JSF framework aims to unify techniques for solving a number of common problems in Web application design and development, such as:

4.2.1 User interface development

JSF allows direct binding of user interface (UI) components to model data. It abstracts request processing into an event-driven model. Developers can use extensive libraries of prebuilt UI components that provide both basic and advanced Web functionality.

4.2.2 Navigation

JSF introduces a layer of separation between business logic and the resulting UI pages; stand-alone flexible rules drive the flow of pages.

4.2.3 Session and object management

JSF manages designated model data objects by handling their initialization, persistence over the request cycle, and cleanup.

4.2.4 Validation and Error feedback

JSF allows direct binding of reusable validators to UI components. The framework also provides a queue mechanism to simplify error and message feedback to the application user. These messages can be associated with specific UI components.

4.2.5 Internationalization

JSF provides tools for internationalizing Web applications, supporting number, currency, time, and date formatting, and externalizing of UI strings. JSF is easily extended in a variety of ways to suit the requirements of your particular application. You can develop custom

components, renderers, validators, and other JSF objects and register them with the JSF runtime.

4.2.6 Custom Component

Component developers can develop sophisticated components that page designers simply drop into their pages.

4.3 The MVC-Architecture of JavaServer Faces

The JavaServer Faces web application framework is based on the Model 2 design, but provides a richer MVC environment that is closer to a real MVC GUI application than most of the available web application frameworks based on Model 2.
Since user events are a main point in GUI applications, the major drawback about most model 2 frameworks results from the fact that HTTP is a stateless protocol. The only event that is recognized by a web application is the HTTP request, while GUI applications can fire more fine-grained events like a changed value in a textfield or a clicked button. A web application has to process all request parameters in order to discover which event occurred.
JSF provides a stateful component model in addition to fine-grained events for building event driven web applications. This simplifies the development of web applications, since it hides the stateless nature of the HTTP protocol and related problems.
The controller in the JSF architecture is consisting of the front controller Servlet called FacesServlet, the centralized configuration file and a set of action handlers for the web application. The front controller Servlet is responsible for receiving every request and performs the steps according to the request processing lifecycle to create a response for the client.
The configuration file named faces-config.xml is a centralized point for managing resources of the web application. It defines for example the navigation according to results of processing in the backend and therefore determines which view is to be rendered next. The event listeners respond to events generated from the JSF event model and manipulate the model or invoke other backend code for example.
The application model in a JSF environment is implemented as a set of server-side JavaBeans that hold values of the model, and define methods on these values. These JavaBeans may also be persisted through an underlying persistence layer and a database through the use of Java
Data Objects (JDO), Enterprise JavaBeans or an object-relational mapping implementation like Hibernate.
The main part of the view is the component tree that contains the stateful user interface components. Components can be rendered in different ways according to the type of the client. The view delegates this work to separate renderers, each taking care of one specific output type, HTML or WML for example.

Additional delegates are validators and converters that can be attached to specific components in order to validate and convert the values entered by the user. Converters are used because the client always submits strings as values for input fields and these may need to be converted to a numerical datatype in case these values need to be used in a calculation. Validators check if values delivered by the client are syntactically correct, for example if the length of a string is correct. The view also features resources, which are for example used for localization of the web application.

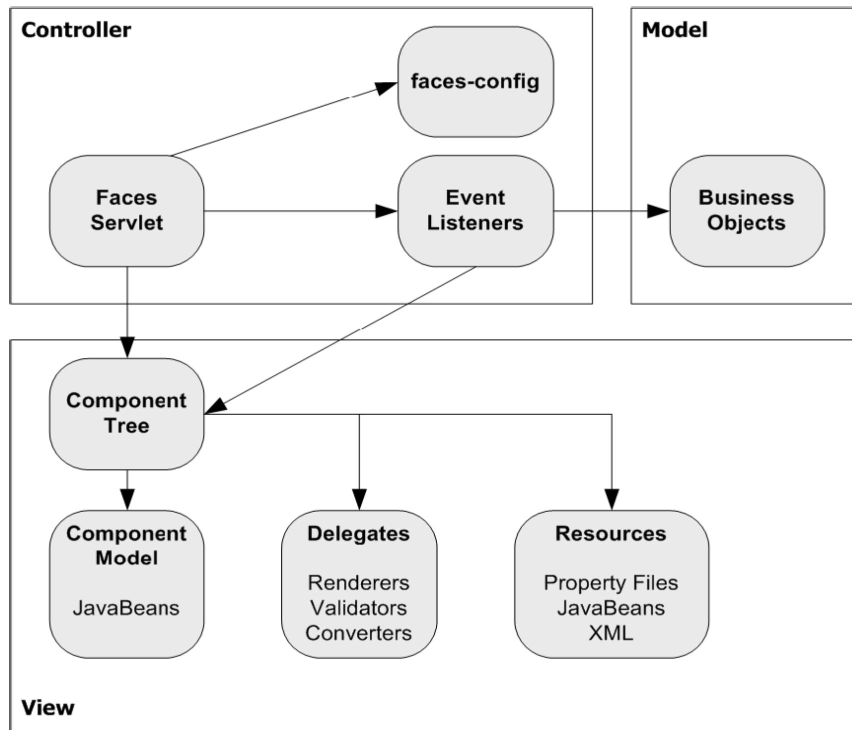

Figure 4.1: The MVC-Architecture of JavaServer Faces

4.4 Lifecycle and Event Model

The JSF specification defines six distinct phases. The normal flow of control is shown with solid lines; alternative flows are shown with dashed lines.

The six phases show the order in which JSF typically processes a form GUI. The list shows the phases in their likely order of execution with event processing at each phase, but the JSF lifecycle is hardly set in stone. You can change the order of execution by skipping phases or leaving the lifecycle altogether. For example, if an invalid request value were copied to a component, the current view would be redisplayed, and some of the phases might not execute. In this case, you could issue a `FacesContext.responseComplete` method invocation to redirect the user to a different page, and then use the request dispatcher (retrieved from the request object in the `FacesContext`) to forward to an appropriate Web resource. Alternately, you could call `FacesContext.renderResponse` to re-render the original view.

The JSF lifecycle structures the developer's development efforts without feeling completely tied to it. The developer can alter the lifecycle when needed without fear of breaking his application. Forms have to be validated before any application logic can be executed, and field data has to be converted before being validated. Sticking to the lifecycle frees the developer up to think about the details of validation and conversion, rather than the phases of the request process itself.

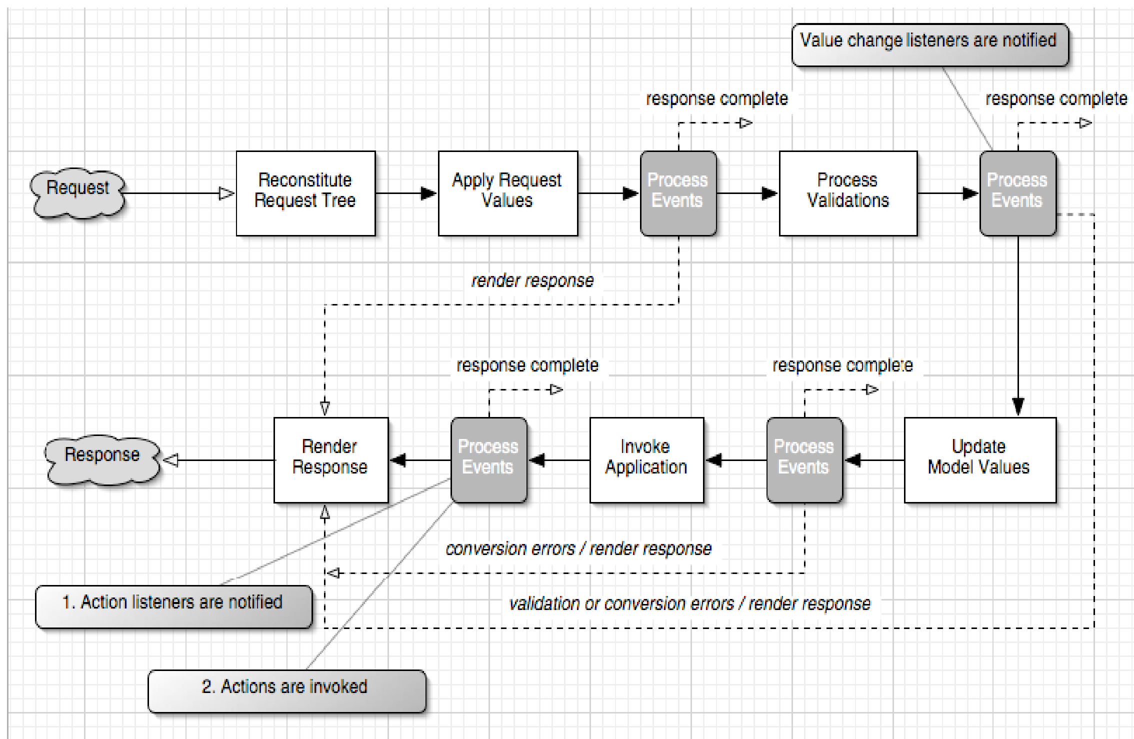

Figure 4.2: JSF Request-Response lifecycle

Phase 1: Restore view

In the first phase of the JSF lifecycle a request comes through the `FacesServlet` controller. The controller examines the request and extracts the view ID, which is determined by the name of the JSP page. The JSF framework controller uses the view ID to look up the components for the current view. The Restore View phase retrieves the component tree for the requested page if it was displayed previously. If the view doesn't already exist, the JSF controller constructs a new component tree and displays it for the first time.

This phase of the lifecycle presents three view instances: new view, initial view, and postback, with each one being handled differently. In the case of a *new view*, JSF builds the view of the `Faces` page and wires the event handlers and validators to the components. The view is saved in a `FacesContext` object. The `FacesContext` object contains all the state information JSF needs to manage the GUI component's state for the current request in the current session. The `FacesContext` stores the view in its `viewRoot` property; `viewRoot` contains all the JSF components for the current view ID.

In the case of an initial view (the first time a page is loaded), JSF creates an empty view. The empty view will be populated as the user causes events to occur. From an initial view, JSF advances directly to the render response phase.

In the case of a postback (the user returns to a page she has previously accessed), the view corresponding to the page already exists, so it needs only to be restored. In this case, JSF uses the existing view's state information to reconstruct its state. The next phase after a postback applies request values.

The restore View phase retrieves the component tree for the requested page if it was displayed previously or constructs a new component tree if it is displayed for the first time. If the page

was displayed previously, all components are set to their prior state. This means that JSF automatically retains form information. For example, when a user posts illegal data that are rejected during decoding, the old inputs are redisplayed so that the user can correct them. If the request has no query data, the JSF implementation skips ahead to the Render Response phase. This happens when a page is displayed for the first time. Otherwise, the next phase is the Apply Request Values phase.

Phase 2: Apply request values

The purpose of the *apply request values* phase is for each component to retrieve its current state. In this phase, the JSF implementation iterates over the component objects in the component tree. Each component object checks that request values belong to it and stores them. The components must first be retrieved or created from the `FacesContext` object, followed by their values. Component values are typically retrieved from the request parameters, although they can also be retrieved from cookies or headers.

In addition to extracting request information, the "Apply Request Values" phase adds events to an event queue when a command button or link has been clicked. Events can be executed after each phase. In specialized situations, an event handler can "bail out" and skip to the Render Response phase or even terminate request processing altogether.

Phase 3: Process validation

The first event handling of the lifecycle takes place after the apply request values phase. At this stage, each component will have its values validated against the application's validation rules. The submitted string values are first converted to "local values," which can be objects of any type. However, when conversion or validation errors occur, an error message is added to `FacesContext`, and the component is marked invalid. If a component is marked invalid, the JSF implementation invokes the Render Response phase directly, redisplaying the current page so that the user has another chance to provide correct inputs. If there are no validation errors, JSF advances to the update model values phase.

Phase 4: Update model values

This phase of the JSF application lifecycle updates the actual values of the server-side model namely, by updating the properties of your backing beans (also known as managed beans). Only bean properties that are bound to a component's value will be updated. We should note that this phase happens after validation, so you can be sure that the values copied to your bean's properties are valid (at least at the form-field level; they may still be invalid at the business-rule level).

Phase 5: Invoke application

In the Invoke Application phase, the JSF controller invokes the application to handle `Form` submissions. The `action` method of the button or link component that caused the form submission is executed. The component values will have been converted, validated, and applied to the model objects, so you can now use them to execute the application's business logic. That `action` method can carry out arbitrary application processing. It returns an outcome string that is passed to the navigation handler. The navigation handler looks up the next page.

Phase 6: Render response

Finally, the Render Response phase encodes the response and sends it to the browser. In this phase the JSF engine renders the UI components in the component tree persisted in the FacesContext. When a user submits a form, clicks on a link, or otherwise generates a new request, the cycle starts anew.

4.5 UI Component Model

UI Components are the centrepiece of JSF and are used as building blocks to create user interfaces that range from simple to complex. Of all the frameworks out there, very few of them incorporate the concept of reusable user interface components (beyond the concept of a Web page). JSF is just such an attempt, and it is intended to be for Web user interfaces what Swing is for more traditional user interfaces.

As Swing [Fp05] did for rich user interface clients, JSF provides a standard user interface component framework for the Web. This standardization promises more powerful and visual development environments for Web applications and libraries of rich user interface components like calendars, trees, and tables.

JSF also provides a number of other standard component types that play a supporting role to UI components. These include Converters, Validators, Renderers, and others. Like UI components, these components are interchangeable and can be reused throughout the same application and a host of others.

4.5.1 UI Component Classes

User interface components are the basic reusable entities for developing user interfaces using the JavaServer Faces framework. The components are build on top of the JavaBeans specification, and therefore have properties, methods and an event model. Unlike Java Swing components, JSF user interface components are situated on the server-side. This fact leads to a major difference between user interface components from standard desktop clients and web application user interface components. The latter do not interact directly with the client, so that every interaction between client and server requires a complete HTTP request-response cycle.
The user interface component itself defines only its functionality. A renderer creates the appearance to the user. Thus, this architecture separates the functionality from the presentation, which allows a flexible handling of different client devices.
All user interface components are derived from the base class UIComponent, which defines methods for the navigation in the component tree, interaction with the backing data model and managing validation, conversion and rendering.
To simplify the development of custom user interface components a subclass of UIComponent named UIComponentBase provides default implementation for all methods, so that a developer only has to extend necessary methods without writing the whole component from scratch.

4.5.2 Events and Listeners

Events represent user interactions with the user interface and therefore provide the main mechanism for the user interface components to propagate user actions on the interface to

other application components. Clicking a button or changing a value in an inputfield of the user interface for example triggers events.
The event model of JavaServer Faces is based on the event model of the JavaBeans specification, which is also the basis for the Java Swing GUI toolkit. An event object encapsulates a particular type of event and also additional information about the source of the event. The events are broadcasted to listeners, which take care of executing the appropriate code if a specific event has been fired.
Event driven development is a standard for client based GUI applications, but it is new to server-side web applications. The event model brings the development to a higher abstraction level, removing the need to operate directly on level of the underlying stateless request-response model of HTTP. Thus, integration of application logic is not more than assigning listeners to user interface components that generate events appropriate to the listeners.
User interface components may fire an arbitrary number of events and may have an arbitrary number of interested listeners associated to it. JavaServer Faces defines four standard events:

Action events are fired when the user interacts with a component that represents a command, for example, when the user clicks on a CommandButton component. The counterpart to an action event is the ActionListener, which takes care of handling the fired events. Action listeners are divided into two types whether they affect the navigation or not. If navigation is affected, the listener performs some processing and returns a logical outcome, which is used by JSF for selecting the next view that is to be sent to the client. Action listeners that do not affect the navigation usually manipulate components in the current view or the backing data model. When the execution of such a listener is finished, the current page is simply redisplayed.

Value changed events are generated by UIInput components, when the value of the component has changed, for example if the user has modified the value of an inputfield.
Value-changed listeners handle these events.

4.5.3 Converters

Web applications receive data from the client as HTTP request parameters that have a string based representation while the backend data is represented through Java objects. In other words, a web application has to handle two different viewpoints:

The model view represents the server-side, where the application handles the data in form of Java objects that hold the data as native Java datatypes. For example a java.util.Date Object that represents a specific date.

The presentation view is regarded as the client-side, where data is presented in a human readable form that gives the user the opportunity to manipulate the data. This is usually the string-based representation. A date, for example, may be represented as a single string or separate strings for year, month and day. Additionally, localization and formatting is performed on the level of the presentation view.

In order to transform the data between the model and presentation view, JavaServer Faces provides a conversion model for type conversion between the usually string based representation of the data in the presentation view and the representation based on Java objects in the model view. Pluggable converter components can be associated to each UI component that is able to hold a local value, for example an input component. An associated

converter is in charge of converting the component's data between the two viewpoints of the application, from a string to a Java object or vice versa.
Converter components implement the javax.faces.convert.Converter interface defined by the JavaServer Faces specification. It consists of two methods, each one in charge of converting in a particular direction:

The `getAsObject()` method converts the presentation view, usually a string, into the corresponding model view represented through Java objects. The method is called during "Decoding" in the"Apply Request Values" phase at the beginning of the request processing lifecycle.

The `getAsString()` method converts from model to presentation view, so that data held as Java objects is converted to strings which can be used to render a response to the client during "encoding" in the "Render Response" phase at the end of the request processing lifecycle.
Since converters are pluggable components that can be associated to many components, they provide an optimal reusability among several components. JSF already defines a set of converters, for example for date, time and numerical conversions.

4.5.4 Validators

Validation of the data users have entered into forms of the web application is critical to the operability, since unexpected types of input may cause the application to behave in a completely unexpected way or may leave the systems' data in an inconsistent state. Additionally many of the security problems web applications have nowadays, result from insufficient validation or even direct usage of the data that is submitted from the client.
In order to ease the development and reusability of validation logic, JavaServer Faces provides validation through validator components that perform the validation of input data on the server-side. One or more validators may be associated to a UI component that implements the EditableValueHolder interface defined in the JSF API. This means that the component can hold a value, like for example input fields can. A validator associated with a component checks the local value of the component during the"Process Validations" phase of the request processing lifecycle.
Similar to rendering, validation can be implemented in two ways:

Direct validation allows the validation logic to be implemented through overriding the `validate()` method of the UIComponent superclass. The validation logic is hard coded inside the UI component, and is therefore not reusable among other UI components.

Delegated validation separates the validation logic from the component and therefore allows several UI components to share the same validation logic. JSF supports two types of delegated validation. The first method is to delegate the validation to a method of a JavaBean through a method binding expression. The other method is to use a separate validator class (implementing the Validator interface) that may also be associated to a component. Since this approach results in the clearest separation, it allows reuse of validators among other UI components.

During the"Process Validations" phase JSF invokes the associated validation logic. The validation logic throws an exception if validation of the component's value fails. This exception leads to the fact, that JSF marks the UI component as invalid and skips the rest of the Request Processing Lifecycle, because it is a potential risk for the data integrity of the

application when the client's submitted data is invalid. On a failed validation, a message is stored in the FacesContext and later submitted to the client and displayed to the user.
It is also possible to validate the data with embedded Java Script code that runs on the clientside. As an advantage, this method reduces the number of HTTP request-response cycles, since the client can produce validation error messages directly. But the major drawback about client-side validation is, that a lot of clients do not support Java Script. So it is not sure for the application, that the data has been validated on the client. Client-side validation code has to be placed in the Renderer for the specific component or in the `encode()` methods of the component itself. But actually none of the standard UI components of the JSF reference implementation use client-side validation.

4.5.5 Renderer and Renderkit

Web applications usually send a response in HTML format to the client's web browser. But what happens when the client device is a mobile phone or PDA that does not provide a HTML browser and therefore requires the web application to respond in another markup language? It is also problematic to add support for a different client device later on. It would require a large number of changes within the web application.
JSF technology is a user interface framework for building Java-based web applications that run on the server side, and render the user interface back to the client. The user interface code runs on the server, responding to events generated on the client. A renderer produces the output for one specific component associated to it. A set of renderers is organized into a renderkit. A renderkit has a particular type of output its registered renderers produce. A renderkit is derived from the abstract superclass javax.faces.render.RenderKit.
The rendering model of JSF describes two methods of component rendering:

Direct rendering encapsulates the rendering logic directly into the components, so there is no clear separation of functionality and presentation. This method is used when a component is only intended to work with a particular client device. It allows implementing a component in one single class, which results in an efficient and compact solution. But this comes at the price of poor maintainability and reusability. To implement direct rendering of a component, the component has to override the `encodeBegin()`, `encodeChildren()`, `encodeBegin()` and `decode()` methods to produce the response to the client itself and handle the data received from the client.

Delegated rendering uses a separate renderer. It delegates the encoding and decoding to a renderer class. This leads to the separation of functionality and presentation and allows the use of different client types, localisation, look & feel and so forth.
When using this method, switching between different clients is just a matter of replacing the renderkit.

A renderer is derived from the abstract superclass javax.faces.render.Renderer and implements the same methods a UI component would use for direct rendering. The renderer is a translator between the client- and server-side. If a request is received, the renderer performs the decoding. This means, that the renderer extracts the appropriate values from the request parameters and sets the UI components' values accordingly. In other words, the previous state of the component is reestablished. When JSF is preparing to send a response back to the client, the renderer performs the encoding through creating a representation of the component that the client is able to understand. For example the HTML representation of an input field

CHAPTER 5

Case Study: HMI Digital Library System

This section discusses the application of Java Server Faces in a case study, a digital library system for teaching materials. Java Server Faces shall be applied to its user interface to simply user interface modelling and design.

5.1 Analysis

The HMI system [HM05] is a digital library application with heterogeneous content that concretely supports teacher's teaching units. The library manages higher quality material collected, classified and annotated by community. The library also provides services to seamlessly use external services such as book search.

Problem
The HMI System is an application that contains simple as well as complex structured documents. The User interface should as well support (multimedia) material play-out and facilitate navigation in and retrieval of material using taxonomies. User interface should display document using different format. To achieve such task, responsive user interface for simple material collection and classification are required. As requirements become more complex, sophisticated UI which can managed complex structured content are required.
Another point is the fact that UI are bound to a particular renderer. This is very undesirable especially for a web application whose presentation logic is prone to changes to support different display. Also coupling a rendering format to a UIComponent requires developer knowledge of Mark-up language and web designer programming logic knowledge. In the case of our library the UI contains code handles both presentation and business logic and it's explicitly based on HTML-based web application. Separation of presentation logic from business logic is not fully achieved. Code reading and UI extension is difficult.

UIs are stateless, non-manipulable objects. The developer of a JSP page containing one or more HTML forms must manage the form's UI state and build a mechanism for dispatching incoming requests to an object specific event handling method. This becomes very difficult to manage with object increasing functionalities. Dispatching of client events back to specific **application services (e.g. artefact manipulation)** usually leads to long lines of code. Coding error are more like and debugging becomes very difficult. Severe errors could occur and data model for instance could be populated with invalid data leaving the data inconstant in the model.
Lack of support for Internationalization and localization and this reduces the application to one language limiting the amount of application users.

5.2 Design

At this stage of our project, we focus on the implementation of the system's requirements, its architecture and the components that interact in the implementation.

5.2.1 High-level solution

There should be a strict and clearly defined separation between application logic and presentation while making it easy to connect the presentation layer to the application code. This design enables each member of a web application development team to focus on his or her piece of the development process. This would provide a simple programming model to link the pieces together. UI objects should be responsible for their display. Since our projects primary goal is a browser presentation, a set of ready-made HTML form element, which can be pasted on JSP, will ease the job of any web designer with no programming knowledge.

GUI object should be wired to a data model that translates input events and later gets updated after form input has been successfully validated. GUI objects smart enough to manage interaction with a user, to dispatch events (notifying value or state change) of GUI and to retain object state between client requests. Each GUI object (if necessary) should be attached to its validation code for data conversion and validation and to error messages for error display. That indeed will reduce the programming plumbing that exists in our application code. Application should be made available in different languages.

5.2.2 Technical solution using JSF

While the previous sections provide a detailed introduction to the challenge found in building UI components, the section moves on to illuminate why the JSF technology can be seriously considered as an option to handle the above mentioned requirements. My implementation of custom JSF components is to elaborate on points that distinguish JSF from other Web application Framework. Namely state management, server-side validation, data conversion, component rendering, and custom tag library (that expresses UI components within a JSP page and wires components to server-side objects).

This choice is supported by the fact that JavaServer Faces provides a set of UI component classes that are managed on the server side as GUI components. These components specify all of the UI component functionality, such as holding component state, maintaining a reference to objects, event handling and rendering for a set of standard components. All components are derived from the simple set of base classes (standard GUI component framework) that can be used to define new components.

In addition to creating custom components, the idea of a custom JSF tag (or tag handler) that at runtime dynamically determines the rendering format of its component will be illustrated. This deduction is based on the fact that for component rendering, JavaServer Faces provides a render kit, that defines how component classes can be mapped to an appropriate component tags for rendering on a JSP page. The JavaServer Faces implementation includes a standard `RenderKit` class for rendering to an HTML client, which in this case is the rendering format we expect. Rendering is done dynamically by server-side GUI component. Components are responsible for their display or can be plugged to a corresponding renderer and that renderer is responsible for rendering of the component and that of its children.

A simple logging application will be constructed to illustrate the separation of the presentation logic from the business logic. Separation is achieved in JSF application by associating UI components with a server-side object data through the object data properties. This objects is a JavaBeans component, called backing bean. An application gets and sets the object data for a component by calling the appropriate object properties for that component.

5.3 Creating a custom tag handler and a custom component

This section shows how to implement custom components. Minimally, a tag for a JSF custom component requires two classes: A class that processes tag attributes and a component class that maintains state, renders a user interface, and processes input. The tag class is part of the plumbing. It creates the component and transfers tag attribute values to component properties and attributes.

5.3.1 Create a custom tag that subclasses UIComponentTag

This decision to create a custom tag is very simple. It is based on the fact that JSF components are not inherently tied to JSP. To bridge from the JSP world to the JSF world you need a custom tag that returns the component type. A tag handler needs to gather the attributes that are supplied in the JSF tag and move them into the component object.

Since we expect a Tag that isn't tied to a specific component, it will be clever to aggregate the specific component properties (attributes) in this tag. At run time, only the attributes of the particular component to be rendered will be passed to the component properties. My Tag handler is expected to return the renderer type , to return the component type, to set up properties that might use JSF expressions

javax.faces.webapp.UIComponentTag is the base class for all JSF component tag handlers. There are two abstract methods in UIComponentTag: `getComponentType()` and `getRendererType()`.

The `getComponentType()` method returns the component type for the component that is bound to the tag. The other method, `getRendererType()`, selects the Renderer to be used for rendering the component. The method `getRendererType()` can also well returns null, which means that the component manages rendering itself without using external renderers.

Tag handler must override a setProperties method to copy tag attribute values to the component. This method transfers tag attribute values to component properties, or attributes. Custom components must call the superclass `setProperties()` method to make sure that properties are set for the attributes a UIComponentTag supports that is binding, id, and rendered.

5.3.2 Extending UIComponent

Here I am extending standard classes of the components required. A UIComponentBase class defines the default behavior of a component class. All the classes representing the standard components extend from **UIComponentBase**. These classes add their own behavior definitions, as our custom component class will do. Our custom component class must either extend UIComponentBase directly or extend a class representing one of the standard components. These classes are located in the **javax.faces.component package** and their names begin with UI.

At run time the component to be rendered will be informed. It will render the user interface by encoding markup and process user input by decoding the current HTTP request. Component classes can delegate rendering and processing input to a separate renderer.

Note that in this implementation, rendering is not delegated. In other words, each component will provide it's rendering code. This is to keep things simple. The following are therefore to be achieved: create a class that extends UIComponent , save the component state , override component decode method, override component encode method, register the component with faces-config.xml.

A component class defines the state and behavior of a UI component. The state information includes the component's type, identifier, and local value. Some examples of behavior defined by component class are decoding (converting the request parameter and other information to the component's local value), encoding (converting a local value to some markup), or updating model objects.

The next thing to do is save the component state. JSF does the actual storage and state management, typically though a session, a hidden form field, cookies, etc. (This is usually a setting that you configure.). To save the component state, override the component's `saveState()` and `restoreState()` methods.

JSF calls the `decode()` method at the beginning of the JSF life cycle only if the component's renderer type is null, signifying that the component renders itself. The decode method decodes request parameters. Typically, components transfer request parameter values to component properties or attributes. Components that fire action events queue them in this method.

Next we override the `encode()` method. JSF calls this method in the "Render Response" phase of the JSF life cycle only if the component's renderer type is null, signifying that the component renders itself.

To register the component, we need to put a component entry into the application configuration file.

5.4 Implementation

While the previous sections discussed the theoretical background of JSF component development; this section introduces real components as an example of the previously explained technologies and concepts. The implementation provides a custom input component, a custom command component and a custom tag handler.

Custom composite component

The code fragment 5.1 illustrates an initial JSF custom input component that has been extended by combining the logic of several components. This implementation shows one of the main features of component development using JSF. This component combines label, text input and message functionality into one component and shows how output is encoded. Secondly it is a component that processes request parameters (Decoding) to show server-side data conversation and validation. Its functionality allows users to input text and also presents

an asterisk (*) to denote required fields. Functionalities for value bindings and respective component attributes are added. Most important for this component implementation is the use of a custom renderer for the component. It also shows the difference between direct implementation and delegated implementation of component rendering.

```
public class LabelfieldRenderer extends Renderer{

public void decode(FacesContext context, UIComponent component) {

        /* Grab the request map from the external context */

       Map requestMap = context.getExternalContext().getRequestParameterMap();

        /* Get client ID, use client ID to grab value from parameters */

       String clientId = component.getClientId(context);

       String value;

      value = (String) requestMap.get(clientId);

      UILabelfield labelfield_component = (UILabelfield) component;

          /* Set the submitted value */

  ((UIInput)component).setSubmittedValue(value);

      }

      public void encodeBegin(FacesContext context, UIComponent component)

    throws IOException {

    UILabelfield labelfield_component = (UILabelfield) component;

    ResponseWriter writer = context.getResponseWriter();

    encodeLabel(writer,labelfield_component);

    encodeInput(writer,labelfield_component);

    writer.flush();

}
```

Code fragment 5.1: Renderer extended to a LabelfieldRenderer

Custom CommandButton component

The code fragment 5.2 illustrates the functionalities of a CommandButton. It shows how action event are processed and how the action attribute is used when specifying page navigation.

```
public class ButtonTag extends UIComponentTag{

   private String action= null;

   public void setAction(String action){

             this.action = action;
       }
/* Return the component to be rendered */

       public String getComponentType() {

             return "UIButton";
       }

/* Component renders itself */

    public String getRendererType() {

             return null;
       }

/* Set attributes as properties of component */

    protected void setProperties(UIComponent component){

       super.setProperties(component);

       UIButton command = null;
       try {
            command = (UIButton)component;
         }
       catch (ClassCastException cce) {
            throw new IllegalStateException("Component " + component.toString() + "
not expected type.  Expected: UIButton.  Perhaps you're missing a tag?");
          }

if (action != null) {

            if (isValueReference(action)) {
                MethodBinding vb =
FacesContext.getCurrentInstance().getApplication().createMethodBinding(action,
null);
                command.setAction(vb);

            } else {
                final String outcome = action;
                MethodBinding vb = Util.createConstantMethodBinding(action);
                command.setAction(vb);
```

Code fragment 5.2: UIComponentTag extended to a ButtonTag

Custom selection handler

From the previously gained knowledge (see 5.3.1 and 5.3.2) this handler represents one or more selection custom components in a JSF page. This handler extends a UIComponentTag and contains attributes corresponding to properties of the respective components. The choice of the component to be actually displayed depends on a name parameters or a state parameter of a component.

This decision is made within the Tag handler in the `getComponentType()` method, which returns as string the name of the component associated to this tag. The backing bean properties are used for this purpose as they provide the parameters necessary to make decision upon which component should be rendered.

5.5 Evaluation

Section 5.4 shows how custom components can be created and integrated in an application. Furthermore custom component rendering can be achieved either with standard renderer or with a custom renderer. Our goal with our custom components was to make UI´s more flexible and extensible. We provide the HMI system with rich user interfaces that try to resolve the traditional challenges of User Interface such as event handling and data validation by increasing flexibility in render type selection and extending component by combining the logic of several components into a component.

But still developing JSF custom components is all but an easy thing to do as we have to make the decision as whether a component should render itself or should delegate rendering to another class renderer. Both cases are described in our above custom components examples.

CHAPTER 6

Summary and Outlook

6.1 Summary

This thesis discusses JSF component framework, the state of the art technology for component development. This framework gives a fundamentally different programming model from the Traditional request/response-based, which almost hides the underlying stateless nature of HTTP to a developer. In this path we applied the JSF UI component model in a Digital Library scenario (HMI) to enhance the UI definition process. We developed JSF based components with an inherent autonomy concerning the actual selection of component type and thus rendering result.
By this we were able to create UI components that autonomously select an appropriate rendering according to parameterizable strategies.

6.2 Outlook

Building robust and maintainable User Interfaces is a challenge to any component developer. With JavaServer Faces, we have a UI component technology at hand that provides us with a set of APIs for building rich user interfaces for Java applications. Thus flexibility in component type and component view selection can and should be extended in order to achieve components with personalized look and feel. Other UI components could be made more flexible e.g Data grid/scrollable data tables which dynamically grows in size depending on number of elements it contains.

REFERENCES

[Be04] Hans Bergsten: JavaServer Faces. O´Reilly Media, 2004

[Bo04] Andy Bosch: Das Standard-Framework zum Aufbau webbasierter Anwendungen. Addison-Wesley, 1st edition 2004

[Br03] Simon Brown: Pro JSP, Third Edition. Apress, 2003

[CDK03] George Coulouris, Jean Dollimore and Tim Kindberg: Distributed Systems: Concept and Design. Addison Wesley, 2003

[DL04] Bill Dudney, Jonathan Lehr: Mastering JavaServer Faces. Wiley Publishing, 2004

[El05] Timo Elverkemper. Diploma Thesis; A project management based on JavaServer Faces and Hibernate. Uni of Applied Sciences Oldenburg, 2005

[ES04] Exadel Studio: How to create your own JSF Component, http://www.exadel.com/tutorial/jsf/jsftutorial-customcomponents.html

[Fi05] Paul Fisher: Introduction to Graphical User Interfaces With Swing. Addison Wesley, 2005

[GH04] David Geary, Cay Horstmann: Core JavaServer Faces. Addison Wesley, 2004

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnsons, John Vlissides: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, 1995

[Gr02] Frank Griffel. Componentware. 2.edition dpunkt-Verlag, 2002

[HG06] Marty Hall, James Gosling: Coreservlets. http://www.coreservlets.com, last visited 2006

[HM06] Sven Haiges, Marcel May: JavaSever Faces,Web Development mit dem Standard-Framework. Entwickler, Press, 2nd Edition, 2006

[HM05] Web reference to HMI http://www.sts.tu-harburg.de/projects/HMI, 2005

[Ho06] James Holmes: JavaServer Faces Resources
http://www.jamesholmes.com/JavaServerFaces

[IB05] IBM: JSF Component Development
http://www-128.ibm.com/developerworks/java/library/j-jsf4

[JC06] JSF Central
http://www.jsfcentral.com, last visited 2006

[JT06] JSF tutorials Webpage:
http://www.jsftutorials.net

[Ma05] Kito Mann: JavaSever Faces in Action. Manning Publications, 2005

[MF06] MyFaces. The free implementation
http://www.irian.at/myfaces/home.jsf, last visited 2006

[Mü06] Bernd Müller: JavaServer Faces- Ein Arbeitsbuch für die Praxis.
Hanser verlag, 1st Edition, 2006

[Sc05] Chris Schalk: Building custom JavaServer Faces
http://www.theserverside.com/tt/articles/article.tss?l=BuildingCustomJSF

[SM05] Sun Microsystems: Java EE 5 Tutorial
http://java.sun.com/javaee/5/docs/tutorial/doc

[SM04a] Sun Microsystems: JavaServer Faces
http://java.sun.com/javaee/javaserverfaces

[SM04b] Sun Microsystems: JavaServer Faces Specification 1.1
http://java.sun.com/javaee/javaserverfaces/1.1/docs/api/index.html, 2004

[Sw06] Jeff Swisher: JavaServer Faces: Architecture in Practice
http://bdn1.borland.com/borcon2004/article/paper/0,1963,32269,00.html

[Zu05] Zukowski John: The Definitive Guide to Java Swing, Third Edition.
Apress, 2005

.