Struts2JSF:
Framework Migration in J2EE
using Framework Specific Modeling Languages

by

Aseem Paul S. Cheema

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2007

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

Java 2 Enterprise Edition is a portable, robust, scalable and secure platform for enterprise software development based on Java technologies, and embraces open standards through the Java Community Process (JCP). J2EE development is not very productive because of the complexity of the platform and the lack of good tool support. Object-Oriented Frameworks are a reliable design and code reuse approach. Many frameworks have emerged since J2EE's release to ease development. Struts has become the de-facto standard, while JavaServer Faces (JSF) is a new framework, which has been included in the J2EE specification and hence standardized. Both Struts and JSF frameworks are based on Model-View-Controller design pattern. JSF takes a similar approach to Struts for the controller component, but adds to it by providing user interface components with server-side state for the view component.

This work deals with the problem of migrating an application based on the Struts framework to the new JSF framework. The software migration task is divided into view and controller migration. Controller migration is semi-automated using Antkiewicz's *Framework-Specific Modeling Languages* (FSML) approach. Guidelines are provided for view migration, which boils down to the problem of componentization. JSF and Struts frameworks can also be used together where JSF supports the view component while Struts supports the controller component. Merits and demerits of this approach are also discussed.

Acknowledgements

I would like to thank my supervisor, Prof. Andrew J. Malton for his continuous guidance and support at each step of the process. His knowledge and experience in the field of Software Migration, Software Engineering and his ability to convey his understanding clearly to students, helped me get a clear perspective of the problem. Without his guidance, I would have strayed without accomplishment.

I thank Prof. Krzysztof Czarnecki and Prof. Steve MacDonald for taking the time to review my work and to serve on the committee.

I thank Michal Antkiewicz for introducing me to framework-specific modeling and Eclipse plugin development. He went to the extent of pair programming with me to give me a head start on designing a FSML and implementing an Eclipse plugin.

I thank Mr. Mike Goel from Taxwide Inc. for allowing me to use the TOAST application source code for the case study. The case study was used for evaluation of the prototype developed as a part of this work.

I thank all SWAG group members for their support and valuable advice regarding my research work. I specifically thank them for providing a wonderful work environment with the right mixture of work and play.

I thank my family for being by my side throughout and for their continuous encouragement to pursue higher studies and to accomplish the goals I set for myself.

Contents

List of Tables

List of Figures

Trademarks

Java, Java 2 Standard Edition, Java 2 Enterprise Edition, Java Enterprise Edition 5.0, JavaServer Pages, JavaServer Faces, Servlets, Enterprise JavaBeans, Java Transactions API, JavaBeans, Java Naming and Directory Interface, Java Messaging Service, JavaBeans Activation Framework, Sun Java Studio Creator are registered trademarks or trademarks of Sun Microsystems, Inc.

Active Server Pages, ASP .Net, Microsoft Visual Studio .Net, Microsoft Visual Studio 2005 are registered trademarks or trademarks of Microsoft Corporation.

Chapter 1

Introduction

As a large, high-stake software system becomes old and the underlying technologies become obsolete, the maintenance costs tend to rise. One way of handling this problem is to migrate the software system to a newer environment. The environment may be the programming language, the underlying operating platform, the database system or a framework. This is referred to as *Software Migration*. This work deals with the problem of Framework Migration, specifically concentrating on frameworks in Java 2 Enterprise Edition (J2EE) [15] domain.

In this work, we study migration strategies for migrating an application based on the Struts [39] framework to the new JSF [45] framework. We break the migration task into two separate migration tasks involving the view and the controller components. The two components are loosely coupled by configuration files. The controller component migration is semi-automated using a Framework Specific Modeling Language (FSML) [64], discussed in Chapter 3 and Chapter 4. A step-by-step guide to migrating the view component of an application is discussed in Chapter 5.

The rest of this chapter is organized as follows: Section 1.1 discusses research work related to Software Migration in general, Section 1.2 discusses Object-Oriented Frameworks,

and Section 1.3 discusses the need for Framework Migration and the challenges posed by the problem.

1.1 Software Migration

Software Maintenance [52] is defined as

> "Modification of a software product after delivery to correct faults, to improve performance or other attributes, or to adapt the product to a modified environment".

Software Maintenance tasks are classified as corrective, adaptive, perfective and emergency [52]. A slightly different classification of Software Maintenance tasks is as corrective, adaptive, perfective and preventive as in [63].

Software Maintenance can be the most expensive phase of the Software Development Life Cycle. As the code becomes older, the maintenance costs tend to increase for a number of reasons. Unavailability of original developers, manpower shortages for a particular skill-set, obsolete technologies, continuously changing business requirements and design flaws originating from quick-fix maintenance activities are some of the most prominent reasons for increased maintenance costs. In Parnas' words [67], the software system tends to undergo an aging process because of prolonged maintenance.

Software Migration is deemed a Software Maintenance activity. Malton defines software source migration in the *Software Migration Barbell* [65] as

> "The re-engineering task of deploying existing software in a new environment, by significant modification of the source code."

As the stake in a software system increases, it becomes more important than ever to migrate the legacy system of critical importance to adopt new technologies. Migrating

large software systems manually is a daunting task. A manual migration is harder than writing a new system from scratch because of the constraints the legacy system imposes and hence is not feasible. Many attempts have been made to automate and/or semi-automate the migration activity. Most use ad-hoc migration processes that are specific to the technologies under consideration. These processes tend to follow a pattern: reverse engineering followed by forward engineering.

Terekhov and Verhoef [69] give an account of their experiences with language conversions in *The Realities of Language Conversions*[1]. It is highlighted that "easy conversion is an oxymoron" and the difficulties of language conversions are often underestimated resulting in failures. A three step process is suggested for language conversion: restructuring in the original program, syntax swap and restructuring in the target program. Also, the requirements for the conversion tools are listed.

Kontogiannis *et al.* [61] report on experiences with PL/IX to C++ language conversion. The importance of the Abstract Syntax Tree (AST) representation is highlighted and used throughout the process. The process used is specific to the languages under consideration and consists of the following steps: data structure transformations, generation of supporting utilities to handle the different language constructs and generation of the new system guided by AST traversal.

Martin and Muller [66] present the *Ephedra* approach for C to Java migration and provide account of their experiences using this approach on three case studies. *Ephedra* is a three step approach: insertion of C function prototypes, object-orientation with the removal of multiple inheritance and transliteration of source code. *Ephedra* is specific to C/C++ to Java migration.

[1]Migration, conversion and transformation have been used synonymously in the literature, and in this work. All three refer to software source migration.

Malton proposes a three step systematic approach for Source Migration called the *Barbell Model* [65] in *The Software Migration Barbell*. In this work, Malton also classifies Software Migration as follows:

- *Dialect Conversion* implies Software Migration from one version of a programming language to another. This is usually a result of evolution of compiler technology. Java to Java 2 conversion is an example of Dialect Conversion. This is the easiest to automate.

- *API migration* refers to source migration that is a result of a change in the API of external libraries or a framework.

- *Language Conversion* implies Software Migration between two different languages and is the hardest of the three because of a paradigm shift. C to Java migration is an example of Language Conversion.

The *Barbell Model* consists of three phases namely, source normalization, blind translation, and target optimization. The model is depicted in Figure 1.1.

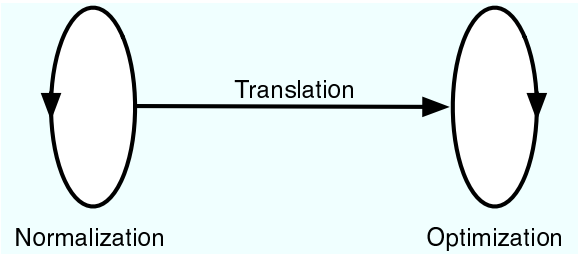

Figure 1.1: The Software Migration Barbell[65]

Normalization means modifying code in the source environment to ease the translation process. The automation tools usually make some assumptions about the source. Translation depends on the automation tools and hence the assumptions. If the assumptions fail,

the translation may be error prone. Normalization can be used to assure the validity of assumptions on which the translation process depends.

Translation represents the actually migration step or process. During this phase, the code is a mixture of source and target environments and hence cannot be compiled. We lose all the support of tools like compilers, IDEs, *etc.* during this phase. Malton insists that the translation should be as blind and as fast as possible because of lack of support during this phase.

Optimization refers to code modification in the target environment to achieve the goal of maintainability. At the end of translation, the target code still has "bad smells" from the source code. The aim of optimization is to remove these bad smells, and make the code as native to the target environment as possible.

1.2 Object-Oriented Frameworks

The Object-Oriented (OO) paradigm promotes reuse by data abstraction, encapsulation and inheritance [60]. Encapsulation brings the data and the methods together in one module (class) promoting modularization and enabling reuse of the module. Inheritance also promotes reuse of code, as other classes can extend base classes thereby inheriting the code of the base class. The component of reuse in OO is a class.

Johnson and Foote [60] define a framework[2] as "a set of classes that embodies an abstract design for solutions to a family of related problems". A framework is therefore a partial design and implementation of an application in a given problem domain. This promotes reuse of design and architecture as well as the lower level components like classes. Frameworks provide a semi-complete application that is completed by extending the frame-

[2] A framework is a very general concept. This work uses the term framework to refer to Object-Oriented Application Framework.

work at hot spots or extension points. This enables reuse at a higher abstraction level.

The primary difference between a framework and a library is the "Inversion of Control" concept [12]. The framework controls the execution sequence and calls the application specific code. While using libraries, the application code controls the execution and calls library methods.

Most academic research work related to OO Frameworks dates back to late 90's. In [56], Fayad and Schmidt provide an overview of OO Application Frameworks. In this work they identify modularity, reusability, extensibility and Inversion of Control as the primary benefits of OO Application Frameworks. They provide framework classification based on a framework's scope as follows:

- *System infrastructure frameworks* simplify the development of infrastructure software like operating systems, database management systems, *etc.*
- *Middleware integration frameworks* simplify the development of distributed applications.
- *Enterprise application frameworks* are domain specific frameworks for application development.

Based on the above classification, most of the frameworks for J2EE web development are Enterprise application frameworks.

Another classification of frameworks is provided based on the ways to extend a framework. This classification is a rather popular classification. It classifies frameworks as follows:

- *Whitebox Frameworks* use inheritance as the primary extension technique.
- *Blackbox frameworks* use object composition as the primary extension technique.

Based on the above classification, most of the frameworks for J2EE web development are "Graybox frameworks" as they use mixture of inheritance and object composition as their extension techniques. Fayad and Schmidt [56] explain how Whitebox frameworks require deep knowledge of framework structure for application development and provide a tightly coupled system, but are easier to develop. On the other hand, it is easier to develop applications with Black Box frameworks and they provide loosely coupled software systems, but Black Box frameworks are harder to develop.

Frameworks are also compared to the other reuse techniques like design patterns and class libraries [56]. Design patterns are solutions to recurring problems but are not concrete implementations. Frameworks on the other hand, use many combinations of design patterns and provide concrete implementations of semi-complete application for a particular domain. Frameworks also complement class libraries by providing semi-complete application and by implementing inversion of control.

Most of the other research work related to Object-Oriented frameworks is experience reports on framework development and does not address the problem of Framework Migration. Harinath *et al.* [58] describe their experiences with the design and implementation of an Object-Oriented framework for distributed control applications. Commercial off-the-shelf (COTS) products were used and integration challenges were discussed. They mention that using COTS software does not immediately solve all the problems claimed to be addressed. Weinand *et al.* [70] present the architecture of ET++, an Object-Oriented Application Framework based on MacApp [27] and its seamless integration in a Unix environment with a conventional window system. They emphasize how the use of Object-Oriented concepts increased productivity and a complex system was implemented by just two programmers in one year. Srinivasan and Vergo [68] report their experiences with the development of an Object-Oriented framework for speech recognition applications. They

report that the initial learning curve associated with the framework is overshadowed by the productivity gains for application development.

1.3 Framework Migration

Since the Object-Oriented paradigm became main stream, many Object-Oriented frameworks have emerged. These frameworks provide code reuse in the large, hence facilitating design and architecture reuse. As the use of frameworks has increased, a large number of software systems have been developed on top of base frameworks, thereby increasing the dependence on the base frameworks. The application code contains the business logic, while the framework stitches it to the underlying technology.

Situations arise when the underlying framework does not evolve and becomes a constraint on application evolution. This is particularly true when the frameworks are open source or commercial off-the-shelf (COTS) frameworks. This gives rise to the need to migrate the application code to a different, more advanced framework, that supports the new features required by the application. This is where Framework Migration comes into play.

Framework migration is a specialization of Software Migration, where the environment is the framework. The goal is to remove an application's dependency on one framework, and create dependency on another equivalent framework. Figure 1.2 depicts this scenario.

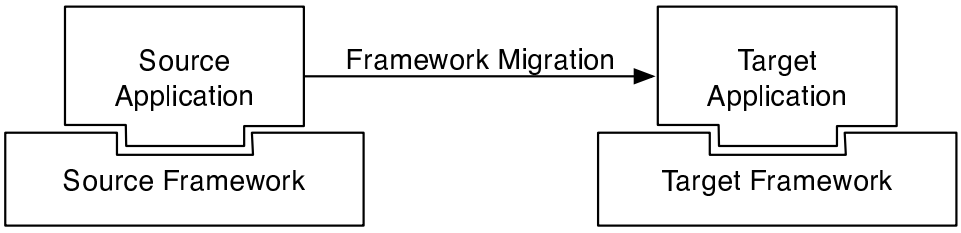

Figure 1.2: Framework Migration

Protocol Mismatch is a challenging problem in Framework Migration. A Framework is a semi-complete application, which is extended at hot spots or extension points to implement the application specific functionality. This is also referred to as *Framework Completion*. The framework hot spots or the framework's interface to the application represents a completion protocol, which is a contract the application must follow in order to use the framework. When two frameworks have a different framework completion protocol, we face the problem of Protocol Mismatch. Protocol Mismatch is the hardest problem in Framework Migration. Migration between two frameworks with same completion protocol is relatively simple and can be compared to API Migration.

There is not a great deal of work in the area of Framework Migration. Chi [55] used a virtualization technique for Framework Migration. A virtual framework is designed that has a similar interface to the source framework. The virtual framework uses an adaptor to talk to the target framework and hence the application can be used with both source and target frameworks with minimal code modification.

Using virtualization technique has the benefit of requiring minimal code changes, but faces three problems. Firstly, because of virtualization, the performance in the target environment (framework) will be affected as the application code will not be native to the target environment. Secondly, a separate maintenance activity will be required to maintain the application code and the virtual framework code, thereby increasing the costs and complexity. Thirdly, if the source and target frameworks use disparate interfaces, virtualization might not even be possible. In this work, the virtualization technique is not used for Framework Migration because of disparate framework completion protocols of source (Struts) and target (JSF) frameworks. A semi-automated code migration strategy is used in this work using a FSML. The migration strategy is discussed in detail in Chapter 3.

1.4 Thesis Organization

This thesis is organized as follows: Chapter 2 discusses the technical background necessary to understand the Struts to JSF framework migration problem and presents the business case for Struts to JSF migration; Chapter 3 discusses migration strategies and the process of using an FSML for Framework Migration; Chapter 4 presents the prototype implementation of a special FSML and the migration process for Struts to JSF controller migration; Chapter 5 provides a guide for the manual migration of the view component; Chapter 6 reports on experiences with a migration activity using the prototype developed; Chapter 7 concludes this work and discusses future directions.

Chapter 2

Technical Background

With the advent of the Internet in the day-to-day lives of people, more and more enterprise systems are moving to web-enabled software systems. Java 2 Enterprise Edition (J2EE) [14] is Sun Microsystem Inc.'s solution [41] for developing web-enabled enterprise solutions. J2EE is the platform of choice for enterprises for its portability, open standards, scalability and security. But J2EE is also well known for its complexity and hence is not one of the most productive solutions.

Object-Oriented Frameworks are a proven code and design reuse approach as discussed in section 1.2. In the recent past, as the J2EE platform has matured, many application development frameworks have emerged for J2EE application development [39, 36, 9, 49, 50, 33, 31, 48, 1]. These frameworks aim at making J2EE development easy and productive, and exploit known Design Patterns and best practices. Over fifty such frameworks exist for J2EE development and more than thirty of these frameworks have serious, large community support.

In the midst of this mesh of frameworks, Java community realized the need of a standard framework for application development. This gave birth to JavaServer Faces (JSF) [45], a framework for component based user interface development. JSF is similar to Struts [39] in

some aspects while different in others. Struts is a de-facto standard for J2EE application development and has strong community support.

This chapter gives technical background required to understand the techniques and technology used in the rest of this work. The chapter is organized as follows: Section 2.1 discusses Java web development and challenges, Section 2.2 discusses the role of frameworks in J2EE application development, Section 2.3 discusses the Model-View-Controller (MVC) Design Pattern, Section 2.4 gives an overview of the Struts framework, Section 2.5 gives an overview of the JSF framework and Section 2.6 discusses the rationale behind migrating from Struts to the JSF framework.

2.1 Java Web Development

Since its introduction in 1996, the Java platform has evolved significantly and is now used to solve many different problems. The Java platform comes in three different flavors: Java 2 Standard Edition (J2SE) [18] is used for console-based and desktop development, Java 2 Enterprise Edition (J2EE) is used for distributed enterprise applications and Java 2 Micro Edition (J2ME) [16] is used for embedded system development.

All Java platforms are based on the Java Virtual Machine (JVM) [26], which is responsible for executing the *bytecode*. Java programs are compiled into bytecode, and not executable code. The bytecode is executed by the JVM, which is the interpreter for the bytecode. This brings portability to the Java platform. Java code compiled to bytecode can be run on any hardware platform for which a JVM implementation exists. Portability is the one great feature that brings Java into the limelight.

The J2EE platform is Java's solution to the problem of distributed enterprise application development. It takes an approach of dividing the application into components and deploying the components on different tiers, hence making it a multi-tired model. The most

commonly used multi-tier approach in J2EE development is to break down the application functionality into four tiers [13], shown in Figure 2.1.

- Client Tier
- Web Tier
- Business Tier
- Enterprise Information System (EIS) Tier

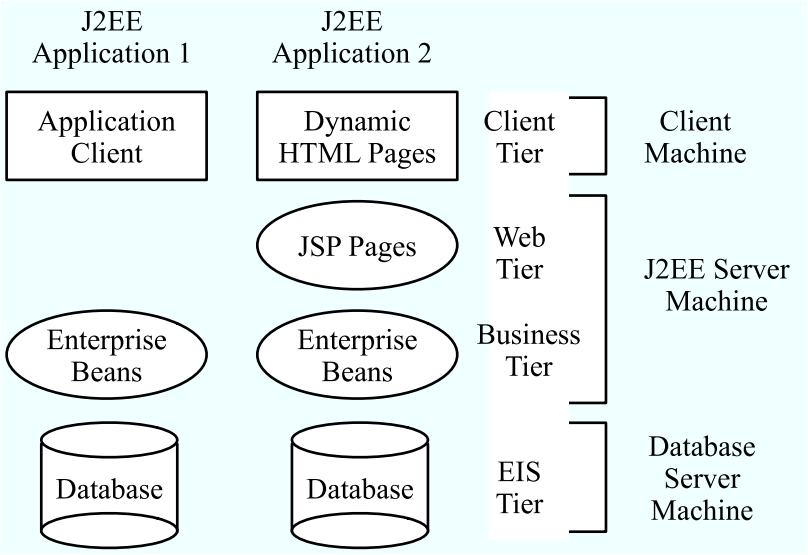

Figure 2.1: Tiers in multi-tiered J2EE applications. [13]

The Client tier is represented by the components that run on the client side, such as Java Applets [2], browsers and Java Desktop applications. The Web tier is composed of JavaServer Pages (JSP) [25] and Servlets [17] that execute in the J2EE servlet container environment on the J2EE server machine. These web components are managed by the

J2EE servlet container and are responsible for response generation and navigation. The Business tier consists of Enterprise Java Beans (EJB) [8] that are also deployed on J2EE Server. These are responsible for business logic. The Enterprise Information Systems (EIS) tier represents other enterprise systems that communicate with J2EE components. These may include Web Services [62], databases or legacy systems.

The J2EE platform provides a rich set of APIs for the development of Web tier components and Business tier components. It is a portable framework because of its underlying JVM technology that offers scalability and security. One of the negative points about J2EE is that it is a very complex framework and hence affects developer productivity negatively.

2.1.1 Challenges

J2EE provides a portable, robust, scalable and secure platform for web development. But, J2EE application development can be challenging due to the complexity of the platform and unavailability of a good Integrated Development Environment (IDE).

Complexity

J2EE is known for its complexity. The specification document itself is 229 pages long [14]. It is made up of more than thirty different technologies that have their own specifications and acronyms. Any beginner in J2EE is overwhelmed with the amount of learning required to get a simple application running. Some people would argue that J2EE is not for simple web applications, but the majority of web applications using J2EE are simple, and do not use enterprise level features like Enterprise JavaBeans (EJB), Java Transactions API (JTA) [20], *etc.* The complexity of the J2EE platform is one of the biggest challenges adopting the technology, and a serious development challenge.

Tools

J2EE is a specification and not an implementation. This ensures that there is never a vendor lock-in situation. The specification discusses the standards that every implementation must follow to be certified. This has created a large market for J2EE application servers. On the other hand, the J2EE specification itself is completely silent about any Integrated Development Environments (IDE)ial. Other J2EE competitors like Microsoft's .Net platform [28] have great tools for Rapid Application Development (RAD) based on the platform. IDEs are known to improve productivity. But in the J2EE world, there is no one IDE that provides comprehensive coverage of all J2EE features and frameworks, and hence J2EE application development is still a difficult task.

There are a number of tools that aim at developer productivity improvements by providing IDEs for J2EE development. Some of them are as follows: Oracle JDeveloper [35], NetBeans IDE [32], Sun Java Studio Creator [19], IntelliJ IDEA [11], MyEclipse [30], Eclipse Web Tools Platform [7], *etc.* Another interesting tool that takes a different approach is RoadMapAssembler [54]. It provides a tool for design-pattern based incremental development of J2EE applications, thereby improving developer productivity and code quality.

2.2 Frameworks To the Rescue

In recent times, a number of frameworks have been proposed for J2EE web development. Some of these frameworks solve specific problems, but most offer ease of development by enabling developers to write a minimum of code. Presently there are more than thirty different frameworks that have a serious community of developers supporting them.

Frameworks provide a robust solution to the challenging problem of J2EE application

development. Most frameworks use J2EE best practices and hence the code quality is better using the framework solution. Inversion of Control frameworks take control of the execution of code and provide an architecture of the J2EE application, to which application specific code segments are added.

Frameworks usually solve a specific problem. Hence, choosing an off-the-shelf framework makes development less complex because we code to the framework rather than the J2EE platform. This brings in the application framework layer between J2EE and the application and hides the complexity of the J2EE platform.

Some frameworks enable ease of development of RAD tools, like JSF. This ensures better productivity.

Most of the J2EE frameworks use proven Design Patterns [57]. Model-View-Controller (MVC) is the most common architectural level Design Pattern used by the frameworks. Section 2.3 gives an introduction to the MVC Design Pattern.

2.3 Model-View-Controller

Mixing business logic code and User Interface code in web applications, that involve user interaction, results in a number of development and maintenance problems. Such high coupling results in code duplication, discourages code reuse because of strong inter-dependencies and hence causes ripple effects when a change occurs. Such mixing of code should be avoided under all circumstances. This is a well studied problem, and has been tackled using the Model-View-Controller (MVC) Design Pattern depicted in Figure 2.2, which has its roots in Smalltalk-80 [57]. This section discusses the problem in the web context, explains MVC Design Pattern briefly and discusses the consequences of using MVC.

Enterprise applications need to support multiple types of users with different interfaces.

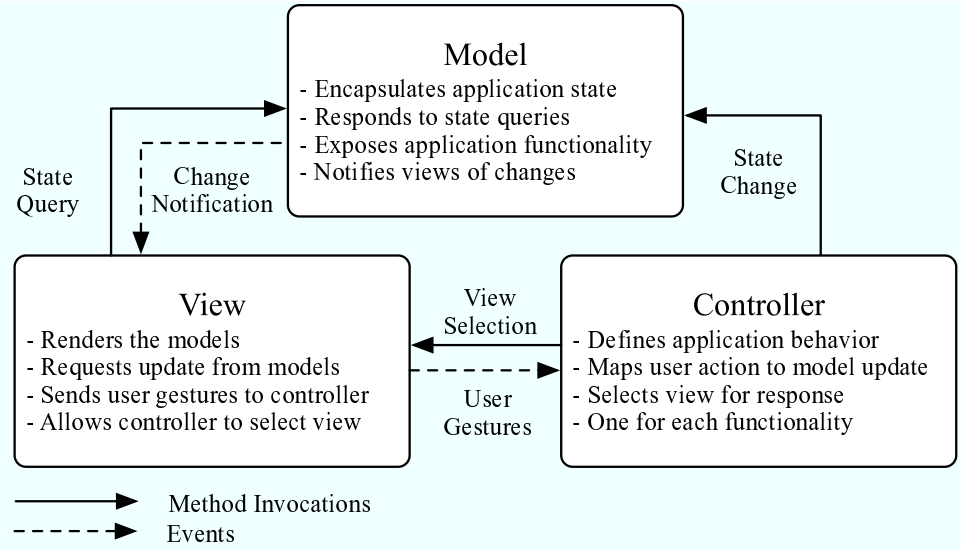

Figure 2.2: Model-View-Controller [29]

The most common interfaces used in the web context are a browser that requires an HTML front end, wireless devices that require a WML front end and web services that require an XML front end. In this context, it is inappropriate to mix business logic code and User Interface code. Such an approach will result in code duplication in each application and will pose serious maintenance problems. The MVC Design Pattern solves this problem by drawing a clear separation between business logic code and User Interface code. The business logic code can hence be reused in all different kind of applications.

A Design Pattern is a description of communicating objects and classes that is customized to solve a general design problem in a particular context [57]. In the MVC Design Pattern, we have three major components. The model is responsible for business logic and data access code that might communicate with other information systems. The view is responsible for rendering the model data to the user. It accesses information systems through the model and presents the state of the model to the user. The controller is the in-

termediary between the model and the view. It translates user interactions in the interface into actions to be performed by the model.

In the Java web development world, MVC is also known as Model 2 [29]. The controller component is implemented using Servlets and is responsible for request processing and data validation. The view is largely JavaServer Pages (JSP) pages and HTML, and is responsible for response generation. The model is Java code and is responsible for business logic and communication with databases, Enterprise Information Systems, Web Services, *etc.*

Using the MVC architectural pattern, the code reuse and maintenance problems discussed earlier are solved. Firstly, as we make the model component independent of view and controller components, the model can be reused for different applications. The web based application, the WML based application for wireless clients and the XML based web services can all use the same model component, for example, and hence proper code reuse is facilitated. Secondly, because of a clear separation of concerns, it becomes easier to add new types of clients without effecting the existing application. Thirdly, each application can be modified without affecting any other applications. Hence, the MVC architectural pattern facilitates code reuse and avoids maintenance difficulties.

Both frameworks under consideration use MVC Design Pattern. Section 2.4 and Section 2.5 give an overview of the Struts and the JSF framework respectively, from the MVC perspective.

2.4 Struts Overview

Apache Struts is a free, open source framework for creating Java based web applications [39]. It has become the de-facto standard for enterprise applications based on J2EE technologies because of its popularity and strengths. Struts framework is based on the MVC architectural pattern. This section discusses the features and components of the Struts

framework that are critical to understanding the migration strategy from the MVC perspective, as depicted in Figure 2.3.

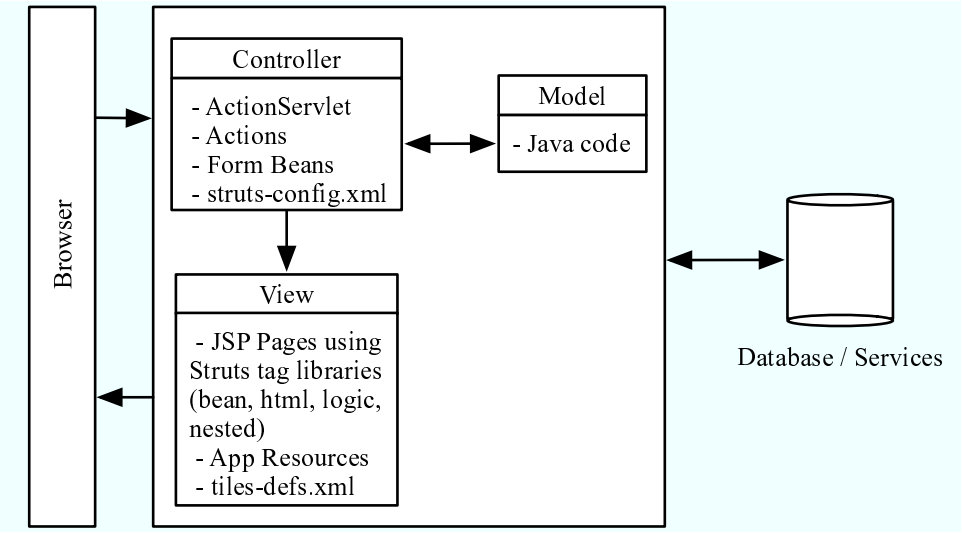

Figure 2.3: Struts Overview

2.4.1 Controller

ActionServlet

The controller component serves as the central point of access for all requests from clients. ActionServlet is a Servlet provided by the Struts' base framework and is the core of the controller component of Struts. It follows the Front Controller Design Pattern [53]. All HTTP requests are received by the ActionServlet, which is responsible for invoking corresponding actions and using proper views for response generation. ActionServlet is also responsible for initializing the Struts framework by reading the configuration file. The information about what actions and views to use is present in the main configuration file. This configuration file also contains values for configuration parameters for this Servlet.

Except for configuring the values of some parameters, the application developer has no other responsibilities when using the ActionServlet.

Actions

The application developer provides the Action classes. They must extend the Action abstract base class in the package org.apache.struts.action, which requires the implementation of the execute method. This is where the Struts framework code ends and the application code begins. The Action classes are responsible for communication between the view and the model. These classes transfer data from the view to the model and then present the results back to the view. Each Action class is configured in the configuration file, and an ActionMapping object exists for each action at runtime. Actions are mapped to the URL patterns in the configuration file and this information is utilized by the ActionServlet, which is responsible for invoking the Actions. Actions should not have any business logic code. Their sole responsibility is data communication between view and model, and the navigation logic. The Struts framework provides many built-in actions that provide a library of commonly used actions. Some of the most commonly used built-in actions are as follows:

- DispatchAction provides a mechanism for modularizing a set of related functions in a single action by providing an abstract class.
- LookupDispatchAction is an implementation of DispatchAction included in Struts.
- ForwardAction provides a mechanism for forwarding to a specified URL.
- IncludeAction provides a mechanism for including JSP content.
- LocaleAction provides a mechanism for setting a locale and forwarding to a specified URL.

ActionForms

ActionForms are data containers. At implementation level these are regular Java beans with getter and setter methods. They must extend the org.apache.struts.action.ActionForm abstract base class. They are responsible for transferring data from the view component to the controller component. Hence, the data from the HTML forms is transferred to the Java beans by the controller layer. Also, form beans are used for transferring data from the model component to the view component, by the controller component. The controller in Struts is responsible for initializing and populating the ActionForms. Application developers write ActionForm classes and declare these as form beans in the Struts configuration file.

Two methods in ActionForm classes are *reset* and *validate*. The reset method is called just before the controller layer populates the form beans with data from HTML forms. The validate method is called after the form bean has been populated to check the validity of the data. Overriding these two methods for each ActionForm is the application developer's responsibility.

DynaActionForm is a special kind of ActionForm that does not need a corresponding Java bean to be written by the application developer. It is declared in the Struts configuration file and uses standard Java data types. Each property for the DynaActionForm is also declared in the declaration.

2.4.2 View

Tag Libraries

The Struts framework provides a JSP tag library that helps in the development of JSP pages. These tag libraries are used to create HTML forms and are also used to display form bean data in the web pages. These libraries also provide tags for conditional logic,

iteration, displaying nested object data, *etc.* There are four different tag libraries in Struts:

- The HTML tag library is used to generate HTML forms. These forms have elements that are associated with form bean properties using tag libraries. The Struts API and base framework take care of transferring data from these form elements to the form bean objects and vice versa. The HTML tag library example shown below generates an HTML input element with text as its type and associates it with the bankName property of a form bean associated with this HTML form. The *html:form* associates a form with an Action which has an associated form bean.

```
<html:form action="/AddSignatory"/>

<html:text property="bankName" size="20" maxlength="128"/>
```

- The Bean tag library is used to access the data in form beans. It is also used to display data from Java property files for internationalization purposes. The example below shows how the bean tag library is used to display a property label.user from a Java properties file. The Java property file is associated at an application level in the Struts configuration file.

```
<bean:message key="label.user"/>
```

- The Logic tag library is used for simple conditional logic in JSP. The example below shows how to use the logic tag library to see if any message is present in the message queue to be displayed to the user.

```
<logic:messagesPresent message="true"> </logic:messagesPresent>
```

- The Nested tag library is used to nest different tags from the Struts tag libraries, which do not work together, otherwise. It is also used to access the nested object properties.

The Struts tag libraries provide tags that are the fundamental building blocks for the view component.

Tiles Plug-in

Tiles [46] is a template system that exploits JSP includes and moves the JSP include programming tasks to XML based declarations in configuration files. This enables reuse of view components. Tiles started as an independent product that is now fully integrated with Struts. It can still be used independently though.

Validation Framework

Struts provides a declarative validation framework. Both client-side JavaScript based validations and server side Java based validations are supported by the framework. It is used for validation of user input data.

2.5 JavaServer Faces Overview

J2EE is a popular framework for enterprise application development and is known for its stability and performance. But it is a well known fact that J2EE does not support rapid application development of enterprise systems due to the complexity of the framework and the lack of good Integrated Development Environments (IDE). J2EE also lacks User Interface development features.

The promise of JavaServer Faces (JSF) is to bring rapid application development to server side J2EE development by providing a User Interface component framework and enabling easy development of IDEs for server side J2EE programming [45]. JSF has a very similar architecture to Struts framework, except that it provides a set of User Interface components that have server side state as shown in Figure 2.4.

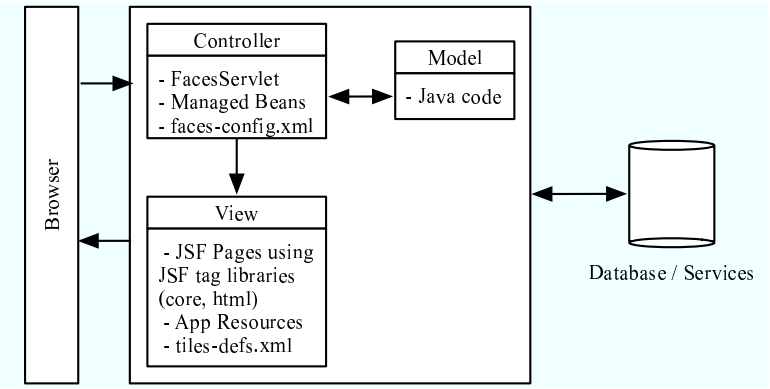

Figure 2.4: JSF Overview

2.5.1 Controller

Managed Beans

Managed Beans are JavaBeans [21] that have properties and methods related to data and events on the user interface side. The UI Component values are bound to Managed Bean properties using the component tag's value attribute, as shown below.

```
<h:outputText value="#{BeanName.propertyName}"/>
```

The methods in Managed Bean are bound to the UI Component events, such as action listeners. The JSF core framework is responsible for converting User Interface actions into Managed Bean method invocations and also setting, getting the properties of Managed Beans.

2.5.2 View

User Interface Components

A JavaServer Faces implementation is required to provide a basic set of User Interface components that have server side state. The JSF base framework is responsible for main-

taining the state and mapping the state to the markup (HTML) elements. There are two fundamental actions that occur behind the scenes to enable server side state handling of User Interface components: decoding and encoding [59]. Decoding refers to parsing of incoming requests parameters to extract and set the state of components that may have been modified by the user. Encoding implies converting the state of the components to markup tags (usually HTML, though encoding depends on the specific rendering plug-in being used).

All JSF UI components consist of two parts. These are the component and the renderer [59]. The component is responsible for the state and behavior of the UI component. The renderer dictates how the state will be read from request parameters and how the markup will be generated from the state. The JSF component model is extensible. New components can be defined by programming the component and the renderer.

2.5.3 The JSF Request Life Cycle

The JSF Request Life Cycle consists of six phases as depicted in Figure 2.5:

- The first phase is the Restore View phase. As mentioned earlier, the JSF core framework maintains server side state of UI Components. During this phase, if a component tree does not exist for a view on the server side, it is created.

- The second phase is the Apply Request Values phase, which is when the values from the request parameters are applied to the components.

- The third phase is Process Validations phase, which applies data conversions (*eg.* string to date, string to number) and then validates the input values. If validation fails, the response is rendered without other phases being invoked.

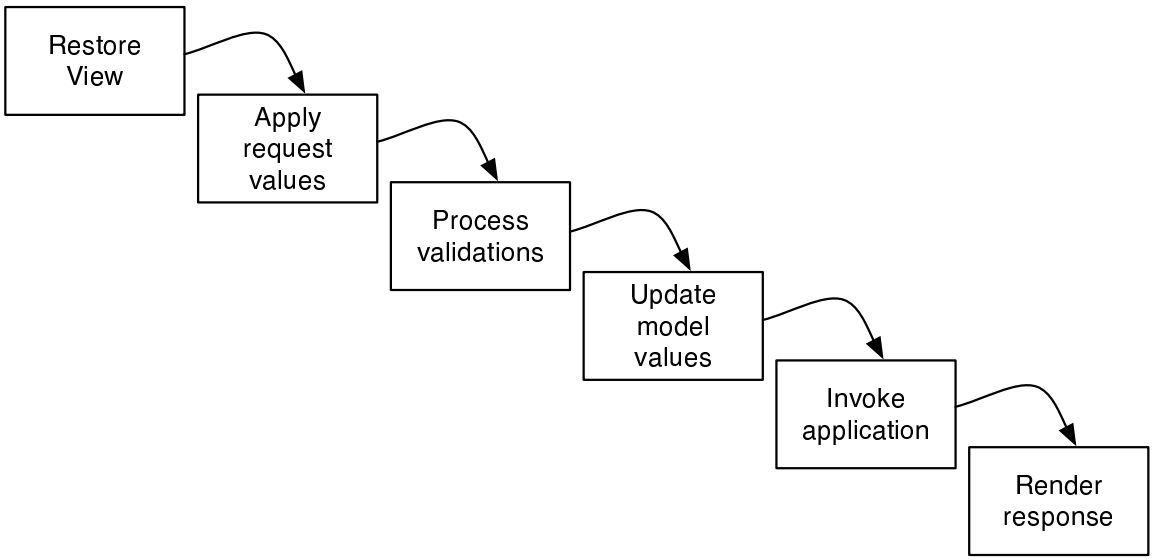

Figure 2.5: JSF Request Life Cycle

- If validation succeeds, then it is followed by fourth phase, the Update Model phase, which is responsible for copying the values to the backing beans. These backing bean properties are bound to the UI components in the JSF component tags.

- The fifth phase is Invoke Application phase, which invokes the listeners and actions. The JSF components are bound to methods of the backing bean using listeners and actions. The listener method is invoked before the action method.

- The sixth phase is Render Response phase. This phase is responsible for rendering the response to the client by loading the next view.

Each phase of the life cycle except for the first and the last phase, is followed by Process Events phase.

2.6 Why migrate to Struts to JSF?

Struts is the de-facto standard for developing Java based web applications. It is a stable framework that provides many excellent features and hence simplifies development. It has a strong and loyal community supporting it. Then why bother migrating to JSF? We need to look at what JSF offers to answer this question. There are many technical and other reasons for migrating to JSF, discussed in this section.

2.6.1 Technical Reasons

Event-Oriented Approach

While using the basic Java web development technologies (JSP, Servlets), we have to think in terms of HTTP requests and responses. This requires a familiarity with the HTTP protocol. While some features of Struts allow us to think in terms of objects and properties, others still require to work at request/response level. For example, for the binding of a form element to an object property, we just provide the name of the property and hence the protocol details are abstracted away. But, when actions are invoked in Struts, the developer must think in terms of HTTP requests.

On the other hand, JSF uses an event-oriented approach. Various events in the User Interface invoke application methods on the server side. This abstracts away the communication protocol details for the developer. Hence, the event-oriented approach provided by JSF is a natural and better way for User Interface development.

UI Component Framework

While using the Struts framework, we use the Struts tag libraries to generate the HTML pages. This strategy is similar to JSP. Hence, the User Interface is created using tags that generate HTML.

In JSF, the User Interface is created using a set of components. The events are generated by these components. The JSF specification provides a basic set of User Interface components like text fields, password fields, command buttons, radio button, check boxes, *etc.* But the real power of JSF is its support for third-party components. This creates a new market place for components. There are already many commercial and open source component libraries available that provide complex components like tree views, data grids, date input fields, *etc.* Hence, the User Interface in JSF is an assembly of components and thus reduces development costs.

Multiple Client Device Support

While Struts tags in the User Interface generate HTML, the JSF tags in the User Interface represent a component-renderer pair on the server side. The component is responsible for the state while the renderer is responsible for rendering. This renderer architecture is pluggable, as the renderer for the same component can be replaced to generate different markup. Hence, the application can be modified with minimal coding effort to support different client devices. Presently JSF components can be displayed using HTML, WML and XML. Also, because of this pluggable renderer architecture, it is possible to create renderers that generate DHTML, JavaScript, XML and AJAX code for a richer client experience.

Tool Support

It is a well known fact that good tools can greatly enhance productivity. The popularity of ASP.Net [3] for example, is largely due to the presence of some good RAD (Rapid Application Development) environments like Microsoft Visual Studio .Net. There are about fifty different frameworks available for Java web development. The tool vendors are therefore

reluctant to support any one framework. The inclusion of JSF in Java Enterprise Edition 5.0 implies standardization on a web development framework. This encourages tool vendors to provide support for JSF. A large number of tools already support JSF for RAD.

JSF provides core framework for development of custom components. All JSF components extend from the core components and this makes it possible for tool vendors to write IDEs that can handle different kinds of JSF components in a similar manner. Hence, JSF is designed with tool support in mind and facilitates development of such tools.

Flexible Controller Architecture

While using Struts, the actions are tightly coupled to the Struts API by extending Action classes. In JSF, action methods can be implemented as Plain Old Java Objects (POJO). JSF uses Dependency Injection (Inversion of Control) as it instantiates and initializes beans of any type. Struts on the other hand has a restriction that Action and ActionForm classes should be of certain type from the Struts API.

Moreover, in Struts there are action objects and form beans. Form beans represent data while action objects represent the logic. This is not good practice in an Object-Oriented setting that is based on the concept of encapsulating logic and data in one unit. Hence, JSF provides a more flexible controller architecture that is more intuitive as well, as it removes the unnecessary mediator layer in between the model objects and form beans that represent User Interface data.

Basic Features

Apart from bringing a completely new perspective of User Interface components and event handling to Java web development, JSF also supports all of the basic features that most other frameworks provide. JSF provides a basic validation mechanism that can be ex-

tended. Declarative navigation handling is at the heart of JSF and is more flexible than navigation handling in Struts because it works at the page level. Internationalization and localization is supported by both JSF and Struts equally well. JSF also provides a basic data conversion mechanism along with an extensible architecture for data conversion.

2.6.2 Other Reasons

Java Community Process

JCP [44] is a community of Java developers from around the world that develop and evolve Java technology specifications. JSF was born and evolved as a JCP JSR - 127 [22] (Java Specification Request). Apache Software Foundation [43], Borland Software Corporation [4], IBM Corporation [10], Oracle Corporation [34] and Sun Microsystems Inc. [41] are some of the many expert group members for JSF. This encourages developers and tool vendors to trust the framework. Struts on the other hand, does not enjoy such a support from industry leaders.

Strong Industry Support

Because JSF is built with RAD tools in mind, industry responded quickly by providing support for JSF in all of the major IDEs for Java web development. Moreover, the market for User Interface components that was created furthered industry's interest in JSF. Many commercial and open source components are already on the market and competing with other implementations with the same features. Struts does not enjoy such industry interest.

Specification vs Implementation

As mentioned earlier, JSF was born as a JSR (Java Specification Request). It is a specification for which many implementations can exist. Presently the Reference Implementation

from Sun and MyFaces from Apache Software Foundation are two most popular JSF implementations. Any JSF implementation must provide the basic User Interface components in addition to the core APIs. But most implementations also provide some extra features in terms of better components for ease of User Interface development. This encourages competition in the JSF world and hence we can expect some great innovations in the area. Struts on the other hand, is an implementation itself, and hence only one Struts exists.

JSF is the Standard

Since the release of Java Enterprise Edition 5.0 (Java EE) [15], JSF has been included in the specification. This means that all container implementations for Java EE will have to provide a JSF implementation. Hence, JSF will be ubiquitous. Struts on the other hand, is not a part of Java EE specification.

2.6.3 Summary

In this section we discussed some technical and non-technical reasons to migrate from Struts to JSF. Although Struts provides a stable framework for web development, JSF has many features that make it attractive. In this work, we will discuss the issues that arise while carrying out this migration activity. We will discuss strategies to solve this migration problem. Some tasks that can be automated in this migration will be identified and algorithms for automating these tasks will be presented.

2.7 Chapter Summary

In this chapter we discussed the technical background necessary to understand the Struts to JSF migration problem and presented the business case for the migration. Chapter 3

discusses various migration strategies and presents the process of using framework-specific modeling for framework migration purposes.

Chapter 3

Migration Strategies

Both the Struts and the JSF frameworks implement the MVC design pattern and provide an application development framework for the view and the controller components. Both frameworks largely rely on XML based configuration files to combine the two components. Both frameworks play no role in the development of the model component. Hence, the migration activity involves controller migration, view migration and the migration of configuration files. The model component of the application can be reused without any code modifications, provided that the MVC design pattern is not violated by introduction of dependencies of the model on the controller or the view components.

In some small scale applications based on the Struts framework, a clear distinction cannot be made between the controller and the model components. In such applications the controller component is largely responsible for performing model functions. The migration strategies discussed in this work are still applicable because the part of the controller component that is responsible for model related tasks will be unaffected by migration. Also, the migration activity does not affect the application logic that is encapsulated in the model or the controller component.

Because of the close resemblance of the source and the target frameworks, facilities exist

for the co-existence of the two frameworks [38]. These facilities allow the developer to use the JSF user interface components in the view component, while still using the Struts based controller component. The Struts community has provided libraries to facilitate communication between the two frameworks in a co-existence scenario. Co-existence will be discussed briefly in Section 3.2.

Based on the goal of migration, the migration strategies can be classified as *Complete Migration* and *Partial Migration*. Figure 3.1 gives an overview of partial and complete migration and highlights the differences between them. The Complete Migration strategy produces an application based solely on the JSF framework, completely removing all application dependencies on the Struts framework. Both the controller and the view components are migrated to the JSF framework. The Complete Migration strategy, its advantages and its disadvantages are discussed in further details in Section 3.1. The Partial Migration strategy produces an application based on both the JSF and the Struts frameworks. The view component is migrated to the JSF framework while the controller component and the configuration files are still based on the Struts framework. Struts Faces Integration Library (SFIL) [38] makes this possible. The Partial Migration strategy, its advantages and its disadvantages are discussed further in Section 3.2.

Both strategies are incremental in nature in the context of the view migration. This implies that the JSP scripts constituting the view component can be migrated individually. Hence, both strategies support a working application during the migration process.

3.1 Complete Migration

The Complete Migration strategy aims at a complete migration to JSF, and a complete removal of application dependence on the Struts framework. It involves the migration of the controller component, the view component and the framework configuration files. The

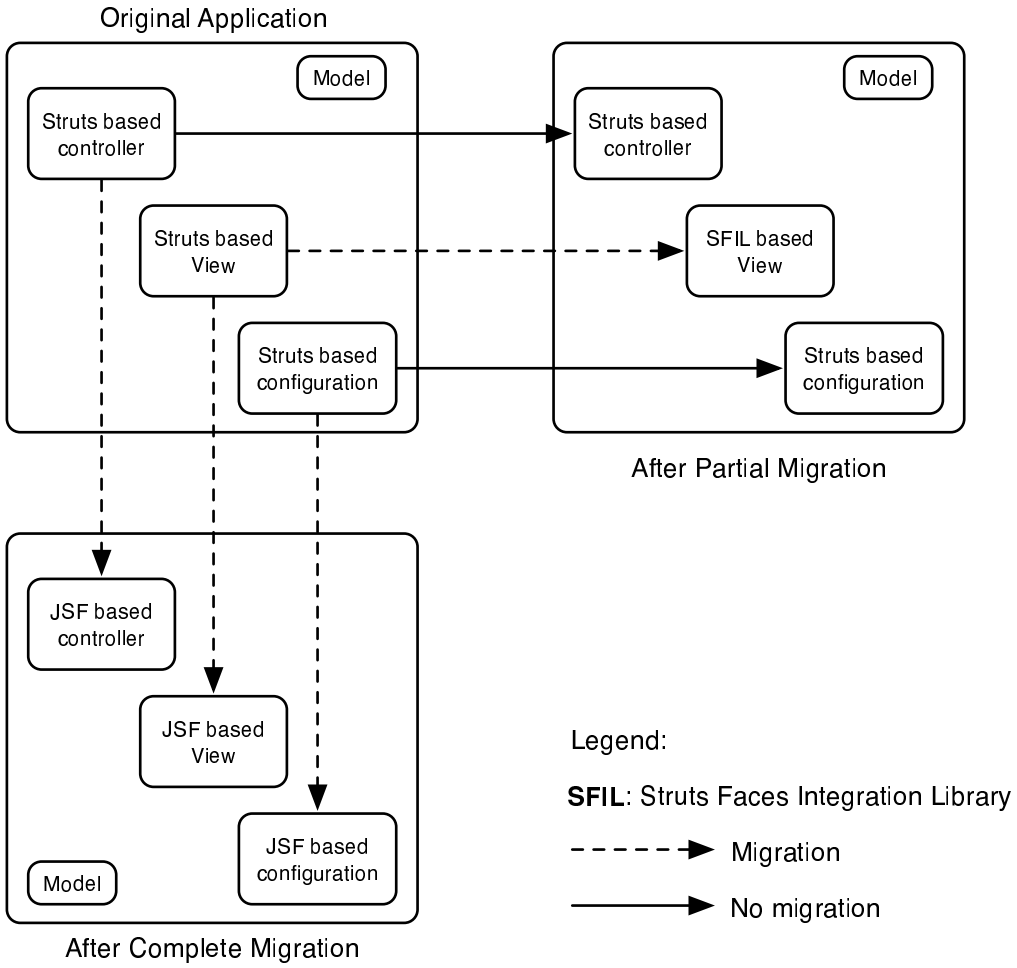

Figure 3.1: Partial/Complete Migration Overview

migration should be carried out in the following order:

1. Controller component
2. Configuration files
3. View component

3.1.1 Controller Migration

The controller component migration should be carried out before any other component migration. The controller component is composed of Java source code. It is dependent on the model component but is independent of the view component according to MVC principles. Hence, the Struts based controller component can be migrated to a working JSF controller component without view migration. The configuration files can be considered as part of the controller component because these configuration files have parameters for the runtime instance of the controller component. Controller migration involves Java API migration and the merging of Java classes.

The source code in the Struts Action classes can be classified as model invocation code and Struts framework invocation code. The model invocation code encapsulates the application logic while the Struts invoking code is responsible for communicating with the base framework. The goal of the code migration will be to replace the Struts invocation code with equivalent JSF invoking code, without affecting the model invocation code.

Another important aspect of controller migration is the merging of the Action and the Form Bean classes. The Struts framework keeps data (Form Bean) and the operations (Actions) in two separate classes. This is not a good Object-Oriented practice. In the JSF framework, the data and the operations reside in one class called a Managed Bean. Hence the related Actions and Form Beans are merged together to form Managed Beans.

The controller migration involving API migration and the high level class restructuring is semi-automated using an FSML. The general process of using an FSML for framework migration is presented in Section 3.3. The details of using the process presented in Section 3.3 for the specific migration task of Struts to JSF controller migration are presented in Chapter 4.

3.1.2 Configuration Migration

Both the Struts and the JSF frameworks use an XML based configuration file to allow for declarative customizations. This simplifies framework development because changes to the applications can be made declaratively by modifying the configuration file. The formats of XML files are governed by the Document Type Definition (DTD) [51] file.

DTDs specify a set of tags that can exist in a configuration file and also the order in which the tags are present. It dictates the structure of the XML document. Both the Struts and the JSF configuration files must declare conformance to their corresponding DTDs.

- `<!DOCTYPE struts-config PUBLIC`
  `"-// Apache Software Foundation//DTD Struts Configuration 1.1//EN"`
  `"http://struts.apache.org/dtds/struts-config_1_1.dtd">`
- `<!DOCTYPE faces-config PUBLIC`
  `"-//Sun Microsystems, Inc.//DTD JavaServer Faces Config 1.1//EN"`
  `"http://java.sun.com/dtd/web-facesconfig_1_1.dtd">`

The root element of the Struts configuration file is “struts-config” and the root element of JSF configuration file is “faces-config”. The top level XML elements of interest during migration in the Struts configuration file are the following:

- form-beans

- global-forwards
- action-mappings
- message-resources

The JSF configuration file elements of interest during migration are the following:

- managed-bean
- navigation-rule
- navigation-case

This section will discuss all of the XML elements of Struts configuration file and their corresponding XML elements in the JSF configuration file.

form-beans*, *action-mappings* to *managed-beans

The *form-beans* element is a collection of *form-bean* elements. These *form-bean* elements represent data transfer objects, and each bean has an associated *name* attribute. The *action-mappings* element is a collection of *action* elements. The *action* element represents the operations to be carried out on the *form-bean* element. Each *action* has a *name* attribute which associates the *action* with a *form-bean*.

During controller migration, Form Beans and Actions in Struts are merged into Managed Beans. From the configuration perspective, *form-bean*s in Struts are declared as *managed-bean*s in JSF. The *action*s in Struts are not declared anywhere in the JSF configuration file. This is because the JSF framework provides a low level mapping of user interface components to the managed bean properties and methods to give it an event-oriented context. This also makes JSF's configuration file much smaller compared to the Strut's configuration file.

global-forwards*, *forward* to *navigation-rule*, *navigation-case

An *action* in Struts also contains a collection of *forward* elements. This collection collectively represents the set of navigation options from the *action* invocation. On the other hand, in JSF the navigation is declared at the view level. This means that forward elements in Struts map to *navigation-case*s and *navigation-rule*s in JSP or JSF pages.

message-resources

The Struts framework provides support for internationalization. This is accomplished using a Java properties file for each *locale* to support. The XML element *message-resources* is used to identify these Java property files. The Struts framework requires this declaration be made in the configuration file. All tags utilizing messages using Struts tags will use these Java property files for internationalization. JSF takes a different approach to internationalization. JSF still uses the Java properties files as its base technology, but it does not require the declaration in the main configuration file. Each JSP or JSF page can individually load a properties file and display messages using JSF tags. This means more work in the view tier, but at the same time, more flexibility in terms of making changes declaratively at run-time. Please note that any changes to the configuration file require re-instantiation of the controller servlet and hence a re-deployment of the application.

3.1.3 View Migration

View migration is not automated in this work. The problem of View migration boils down to the problem of componentization. The Struts view consists of various combinations of Struts tags that generate HTML markup code to generate responses. The JSF view consists of user interface components. This migration problem involves some straightforward tag migrations from Struts tags to JSF tags.

The JSF tag library is not as extensive as the Struts tag libraries. In particular the logic tags provided by Struts do not have an equivalent in the JSF world. This brings the challenging problem of componentization in the picture. Various combinations of the Struts tags are used to generate some complex user interface components that do not have an equivalent in the JSF component library. There are two ways to solve this migration problem. One way is to use JavaServer Pages Standard Tag Library (JSTL) [24] that encapsulates as simple tags the core functionality common to many JSP based web applications. It provides logic tags that can be used to replace the Struts logic tags along with the JSF tags. This approach works only when the JSTL tags are not mixed within JSF components. When mixed together with no clear line of separation, the result is often unpredictable because the two tag libraries are processed during different phases of JSF's request processing life cycle. Hence this approach is not reliable. The second, more reliable approach is to create some custom JSF components and use these components to replace the Struts tag library combinations. Chapter 5 discusses view migration in greater detail. The Struts to JSF tag library map is presented and the combination of the Tiles feature of Struts and the JSF tag library is described. Tiles is independent of the Struts framework, although it is distributed as a package with the Struts binaries. It is compatible with JSF and is the preferred templating engine for JSF.

3.1.4 Pros and Cons of Complete Migration

The advantages of the Complete Migration strategy come from the fact that all dependence on the Struts framework is removed. Firstly, the new migrated application source is native JSF source in every sense of the term and hence has performance as well as space advantages. Only the JSF front controller servlet instance exists at run time as compared to both the JSF and the Struts front controller in the case of partial migration. In partial

migration, the requests are received by the Struts controller and then passed to the JSF controller, hence affecting performance. Secondly, the migrated application source is not constrained by the Struts framework and can exploit all of the features in JSF. For example the managed beans are no longer a specialization of Struts' abstract base classes and this allows for more controller flexibility. Complete Migration avoids the complexity that comes from the integration of two independently developing frameworks.

For large critical applications, the Complete Migration strategy involves more risk as it takes an all or nothing approach to migration. For such migration activities, Partial Migration can serve as a proof of concept and if successful, can be followed by Complete Migration.

3.2 Partial Migration

JSF is primarily a user interface component framework. Although JSF does provide a flexible event oriented controller architecture, but it also is extensible and can be used as a complimentary framework to other frameworks. Using the JSF framework to complement the Struts framework is a particularly common scenario because of the similarities between the two frameworks. The view component of an application can be migrated to JSF components while the controller can still be based on Struts.

The Struts-Faces integration library [38] (SFIL) is of crucial importance in this particular scenario. SFIL translates the user interface events from a JSF view component to Struts actions and events. The controller will have both a Struts ActionServlet instance and a JSF FacesServlet instance. The requests will be generated from the JSF components, but SFIL will translate these requests to Struts' request processing phases and execute related application controller code, which invokes the business logic code.

The Partial Migration strategy does not require migration of the controller component

or the configuration files. Only the view component is migrated. All Struts tags are replaced by the SFIL tags. In this work, we do not discuss the coexistence of Struts and JSF in further details.

3.2.1 Pros and Cons of Partial Migration

This approach provides an incremental way of migrating from Struts to JSF and hence is useful for large, complex applications where Complete Migration is considered risky. But it comes with a number of drawbacks.

Firstly, the most significant drawback of this approach is that it does not take advantage of the marketplace of JSF user interface components because of the dependence on the SFIL. Only the most basic user interface components are supported by SFIL and that restricts the use of advanced JSF user interface components.

Secondly, some of the new features of JSF in the controller component are not used, because the controller component is primarily based on Struts. These features include declarative navigation handling and backing beans that encapsulate the related data from view and methods that operate on data.

Thirdly, this migration requires knowledge of the Struts framework, the JSF framework and STIL. Dependence of the application on the two frameworks and the STIL will translate into more maintenance tasks to keep up with both framework updates and new features. It also means that the developers require a larger skill set.

Hence, it is suggested that Partial Migration should only be used as a proof of concept of migration and should be followed by Complete Migration to completely remove the dependence on the Struts framework.

3.3 FSMLs for Framework Migration

Section 3.3.1 provides an introduction to Framework-Specific Modeling Languages and discusses the benefits of using an FSML for framework completion and round-trip engineering. Section 3.3.2 discusses the process of using a FSML for framework migration instead.

3.3.1 Introduction to FSML

Antkiewicz and Czarnecki [64] defined Framework-Specific Modeling Languages (FSML) and discussed the usefulness of FSML for Round-Trip engineering:

> A Framework-Specific Modeling Language (FSML) is a Domain-Specific Modeling Language that is designed for a specific framework, called its base framework. A FSML consists of an abstract syntax, a mapping of the abstract syntax to the framework API, and, optionally, a concrete syntax.

It is particularly well-suited for model driven development on top of frameworks. Using framework-specific modeling based on an FSML, the task of framework completion can be simplified. The authors describe the task of framework completion as a mixture of *concept configuration* and *open-ended programming* with restrictions. Concept configuration refers to the instantiation of framework-provided concepts in the application code by making implementation choices, which are called *features*. Open-ended programming refers to application code that is not framework specific, but is mixed with the framework specific code to implement the differences between the framework and the application. It also contains the business logic or the code responsible for invocation of business logic residing elsewhere. Open-ended programming is restricted in the sense that it should not violate framework's general rules of engagement.

The framework provided abstractions and implementation choices are formalized in terms of an abstract syntax of an FSML that defines the framework concepts and features. A concrete syntax of an FSML defines rendering details for comprehension of models created using an FSML, and is optional. The mappings can be classified as *Forward Mappings* that define the code generation from model or *Reverse Mappings* that define model generation from code. A complete FSML with all these parts can be used for round-trip engineering. Figure 3.2 shows a birds-eye view of using an FSML for round-trip engineering.

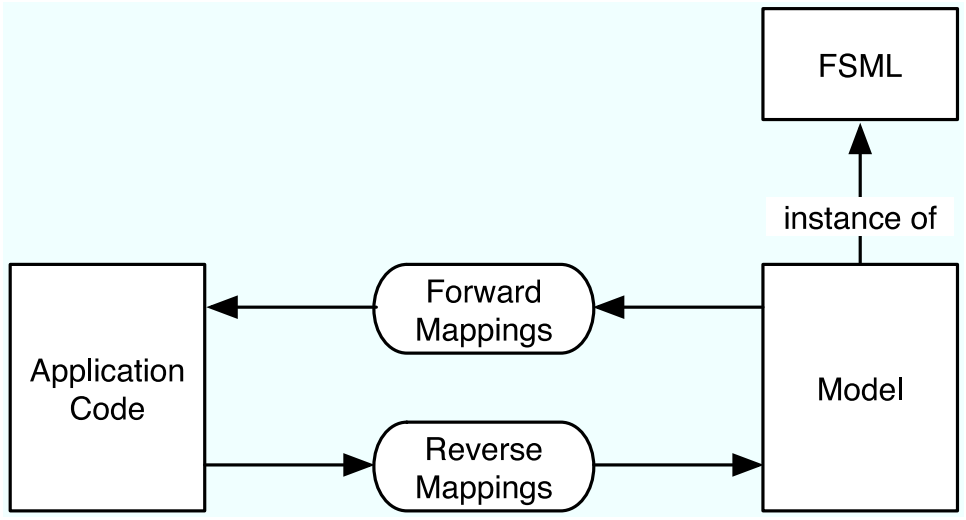

Figure 3.2: FSML for Round-Trip Engineering

In this work we used an FSML for a completely different purpose: the automation of source code migration by using Forward Mappings to generate code for the target framework completion. Section 3.3.2 provides the details of this process.

3.3.2 The Process

In this work we have used framework-specific modeling to automate source code migration between frameworks. This is accomplished by designing and implementing a special purpose FSML common to both the source and the target frameworks. Figure 3.3 gives a high level view of the process of framework migration using a common FSML for the source and the target frameworks. The process involves the following steps:

- *Define* a common FSML.
- *Analyze Code* to generate the model.
- *Migrate Code* using the model and the application source code.

Each of these is discussed in further detail in this section.

Define a common FSML

The first and the most critical task while using framework-specific modeling for framework migration is designing and implementing a common FSML. By the definition of FSML, it is specific to a framework. In this work, an FSML is used as a framework specific modeling language for two frameworks, the source framework and the target framework. The FSML based model will be extracted from the application code in the source framework and then used to migrate the code to the target framework. The model extraction step in the process is done by *Code Analysis* which is analogous to *reverse-mapping* and the target source code migration is done by *Code Re-writing* which is analogous to *forward-mappings* [64].

The technique used in this work is possible only if the two frameworks are sufficiently similar. The two frameworks must have identical or similar concepts and features. Minor differences in the two frameworks concepts and features can be handled by the *forward-*

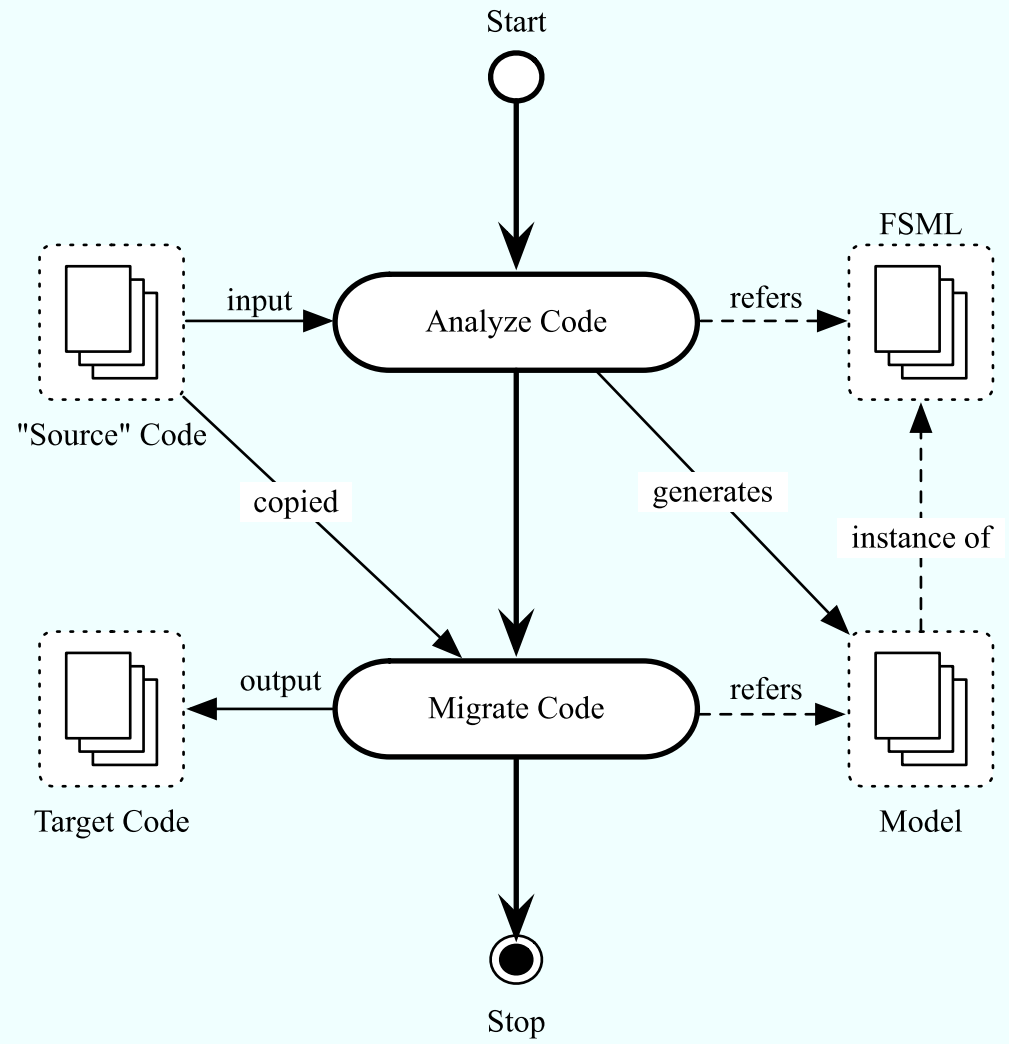

Figure 3.3: FSML for Framework Migration - The Process

mappings that will generate code for the target framework. This assumes that this migration is one directional process, and we will always be migrating from the source framework to the target framework. To solve the problem in the other direction, the FSML will have to be reconstructed, and will be significantly different.

Another approach can be taken if the frameworks do not possess similar concepts and features. Model transformations can be used to transform the model based on the source FSML to the target FSML and code can be generated thereafter. This work does not take this approach and hence this approach is not discussed any further.

Analyze Code

This is a reverse engineering activity. The source application code is parsed and searched for framework concept instances. The model is generated that is expressed in terms of framework concepts as an instance of the FSML. This model presents an overview of the application from the framework's perspective.

The input to the process is the source application. The process refers to the FSML while parsing the source code. The output of the process is a model, which is expressed using the common FSML. This model is referred to by the next step in the process, Migrate Code.

Migrate Code

The final step of this migration process is code migration. Using the model extracted from the source, target application code is generated. This is similar to Forward Mapping in [64], but the migrated code is based on the target framework rather than the source framework.

The source code is copied from the source application and modifications are then made

to the code. This is different from generating code from scratch, which is not what this process does. Source code cannot be generated from the model from scratch because the model does not capture information about open ended programming, which is an important part of the application. Open ended programming encapsulates the business logic code. This code is not modified in framework migration. Only the framework-specific code that is captured in the model is migrated.

To implement this code replacement, the model needs to save pointers to the source application code concept instances. We will refer to this as *Code Marking* in this work. This is implementation specific and will be discussed further in our prototype implementation in Chapter 4.

3.4 Chapter Summary

Complete and partial migration strategies, and their pros and cons were discussed in Section 3.1 and Section 3.2. Section 3.3 introduced Framework Specific Modeling Languages and presented the process of using an FSML for framework migration. Chapter 4 will discuss the prototype implementation of the process discussed in this chapter.

Chapter 4

The Prototype FSML

By the definition of an FSML, it consists of an abstract syntax, a mapping of the abstract syntax to framework API and optionally a concrete syntax. Chapter 3 discussed the process of using a special FSML for framework migration. This chapter gives the details of a prototype implementation of an FSML for the Struts to JSF migration scenario. Section 4.1 presents the abstract syntax of the FSML, Section 4.2 presents the reverse mappings used in *Analyze Code* to identify concept instances in the source code, Section 4.3 presents forward mappings used in *Migrate Code* to migrate to JSF code and Section 4.4 discusses the implementation environment.

4.1 Struts2JSF FSML

This section presents the abstract syntax of the Struts2JSF FSML that captures framework concepts and features necessary for the Struts to JSF migration.

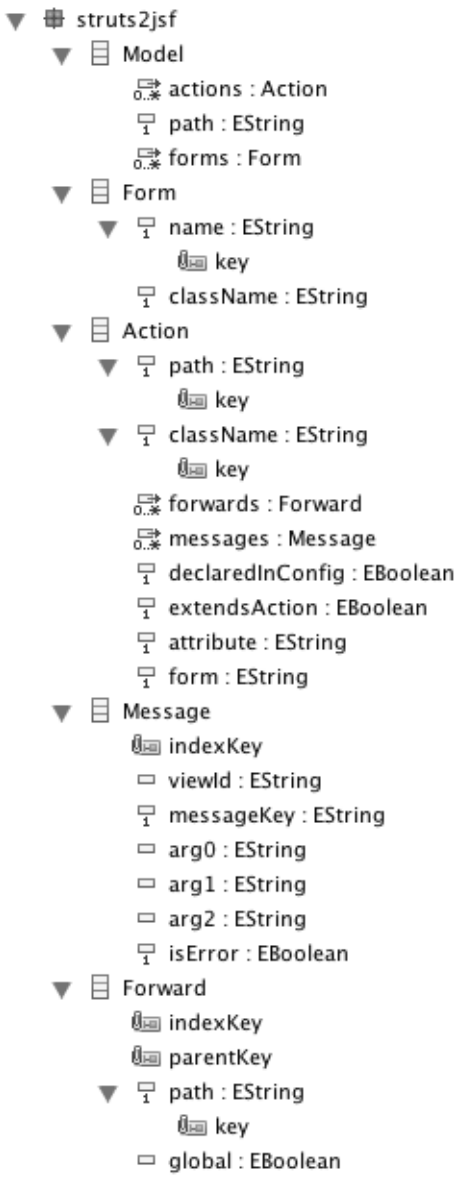

Figure 4.1: Struts to JSF FSML Definition

4.1.1 Model

Figure 4.1 shows the hierarchical structure of Struts2JSF FSML. The top-level concept is the *Model*. This represents the whole application based on the Struts framework. The first class concepts in the *model* are *actions*, *path* and *forms*.

The concept *actions* represents a collection of zero to many instance of the *Action* concept. The concept *path* represents the file system path that represents the Struts' main configuration file. The concept *forms* represents a collection of zero or more instances of the Form concept.

4.1.2 Form

The concept *Form* represents a subclass of org.apache.struts.action.ActionForm. Instances of *Form* exist as data transfer form objects. These objects are instantiated by the Struts base framework with the request data. They can exist in request scope, session scope, or application scope. Request scope form objects are created and destroyed for each request. Session scope form objects live through the lifetime of a user session and are shared by all requests associated with the session. Application scope form objects live throughout the application lifetime and are shared by all requests and sessions.

The *Form* has a *name* associated with it. *Name* is a string uniquely identifying a form.

The *Form* also has a *className* associated with it. This *className* is the fully qualified Java class name associated with this *Form* and must be a subclass of ActionForm in the Java package org.apache.struts.action.

4.1.3 Action

The concept *Action* represents a subclass of the class org.apache.struts.action.Action. This action is also declared in the main Struts configuration file. Every HTTP request originat-

ing from a client is parsed by the Struts front controller servlet to identify the associated *Action*. This *Action* is invoked if data validation succeeds.

The feature *path* represents a string starting with a forward slash (/). The HTTP request is parsed for this string to associate a request with an *Action*. This string is unique and hence is used as an identifier of the *Action* in the Struts framework as well as in the FSML we defined.

The feature *className* represents the fully qualified Java class name that is associated with the *Action*. It has an execute method that contains the application logic and Framwork-Specific code to be executed on the Action invocation.

The feature *declaredInConfig* is a boolean flag that indicates if the Action is declared in the main Struts configuration file. This is not of particular importance from the migration perspective, but can be used as a validation criterion. The framework can invoke only those actions that are declared in the configuration file. Any subclass of org.apache.-struts.action.Action not declared in the configuration file as an Action is not known to the framework and hence the framework can never invoke it.

The feature *extendsAction* is also a boolean flag that indicates if the class associated with this Action is a subclass of org.apache.struts.action.Action. This is also used as a validation criterion. Every action class should be a subclass of the above mentioned framework specific class.

The features *attribute* and *form* represent a *Form* associated with the *Action*. The *Form* is a data transfer object that contains the request parameters. Some *Actions* do not have any associated *Form* because they do not require any data from the client side to process the request. A Logout *Action* is an example of such an *Action* because it just invalidates the user session on the server side and does not need any other information. The feature is a string that uniquely identifies the *Form* concept associated with this Action.

The two features are used for the same entity because different versions of Struts used the two names to identify same framework concept.

The feature *messages* is a collection of instances of the concept *Message*, discussed in Section 4.1.4. Each *message* represents a message to be displayed to the client after the execution of the *Action*. The collection holds a message for each possible result of this *Action*.

The feature *forwards* is a collection of instances of the concept *Forward* discussed in Section 4.1.5. The concept *Forward* represents a navigation case when a user is forwarded to a different view based on the application logic. The *Action* contains the code to navigate to different *Forwards*. A collection of the possible forwards associated with the *Action* are represented by this feature.

4.1.4 Message

The concept *Message* represents an information message, error message or any other kind of message to be displayed to the user. The Struts base framework keeps track of a collection of messages that have not yet been displayed to the user. Once a message is displayed to the user, it is removed from the collection.

This collection of messages is of type org.apache.struts.Action.ActionMessages. The concept *Message* represents each message in this collection. Every *Action* has a number of *messages* associated with it. These *messages* represent the text to be displayed for different outcomes of the *Action*. The *Action* is responsible for adding messages to the collection which are then displayed to the user.

A *Message* has a feature called *viewId* that represents the string that can be used to access this *message* in the view component.

A *Message* has a feature called *messageKey*. With the aim of providing international-

ization and localization features, Struts provides a facility to store all messages in a Java properties file. This feature represents the key from the Java properties file that can be used to retrieve the text associated with a *message*. One Java properties file will exist for each locale supported. Each properties file has the same keys, but the values of the keys will be different for different languages.

A *Message* has three features called *arg0*, *arg1* and *arg2*. These provide a way to personalize the messages being displayed to the user. They are just strings and store the Java code used to get the personalized string to be displayed.

A *Message* has a feature called *isError*. This is a boolean which is true if the message is of type org.apache.struts.action.ActionError that is a subclass of the ActionMessage class from the same package.

4.1.5 Forward

The concept Forward represents a navigation case from an *Action*. It has a feature *path* associated with it, which represents a string starting with a forward slash (/) that uniquely identifies a view to navigate to. *Forward* also has a feature *global* which is a boolean. If global is true then any *Action* can navigate to this *forward*. Global forwards exist independent of the *Action* in the *model*.

4.2 Analyze Code

In this section we discuss the code segments, that the source code is parsed for, to identify different framework concept instances that exist in application code based on Struts. Code analysis is performed with the purpose of generating a model, based on the FSML, of the source application. This model guides the Migrate Code step in the migration process. The

code segments that represent each of the concepts discussed in Section 4.1 are presented using an example application based on Struts.

4.2.1 Model

As discussed in Section 4.1.1, a *model* is the top level concept in the Struts2JSF FSML. The source application code is searched for the main Struts configuration file, called "struts-config.xml" by default. A different name can also be given to the main configuration file, which is configured in the web application deployment descriptor, where the Struts ActionServlet is configured.

```
 1 <servlet>
 2   <servlet-name>action</servlet-name>
 3   <servlet-class>
 4      org.apache.struts.action.ActionServlet
 5   </servlet-class>
 6   <init-param>
 7     <param-name>config</param-name>
 8     <param-value>/WEB-INF/struts-config.xml</param-value>
 9   </init-param>
10   <init-param>
11     <param-name>debug</param-name>
12     <param-value>3</param-value>
13   </init-param>
14   <init-param>
15     <param-name>detail</param-name>
16     <param-value>3</param-value>
17   </init-param>
18   <load-on-startup>0</load-on-startup>
19 </servlet>
```

Figure 4.2: The J2EE Component Deployment Descriptor - "web.xml"

The deployment descriptor for the J2EE web component must be called "web.xml". Figure 4.2 shows the part of the deployment descriptor where the Struts front controller servlet is configured. At lines 3 through 5, the servlet class is declared as org.apache.-

struts.action.ActionServlet. If a need arises to subclass the main servlet, then this class is subclassed, and the new subclass is used as the class for the main servlet. This is particularly a useful extension point for security enhancements for the whole application.

At lines 6 through 9, the parameter "config" is given a value "/WEB-INF/struts-config.xml". This is the relative path to the main Struts configuration file. The Action-Servlet is an instance of the main Struts configuration file in the form of a Java object that is managed by the base framework. The *path* feature of the *model* concept is given the value of the relative path of the main Struts configuration file.

4.2.2 Form

As discussed in Section 4.1.2, forms are data transfer objects. The Struts base framework is responsible for instantiating and managing forms. The forms are configured in the main Struts configuration file as shown in Figure 4.3.

```
.  .....
7  <form-beans >
8    <form-bean
9     name="userForm"
10    type="com.taxwide.toastadmin.struts.form.user.UserForm" />
11   <form-bean
12    name="loginForm"
13    type="com.taxwide.toastadmin.struts.form.LoginForm" />
14   <form-bean
15    name="userList"
16    type="com.taxwide.toastadmin.struts.form.user.UserList" />
17   <form-bean
18    name="locationList"
19    type="com.taxwide.toastadmin.struts.form.location.LocationListForm"/>
20 </form-beans>
.  .....
```

Figure 4.3: Struts Configuration File - Forms

The *form-beans* XML element in the configuration file represents by the *forms* feature

of the *model* concept. It is a collection of *forms*, each of which is represented by the *form-bean* XML element. Each *form* concept has two features associated with it. The *name* feature is the *name* attribute of the *form-bean* XML element. The *className* feature is the *type* attribute of the *form-bean* XML element. The configuration file is parsed to retrieve the *forms* available to the application, which are added to the *model*.

4.2.3 Action

As discussed in Section 4.1.3, each HTTP request is parsed to identify a corresponding *Action*. Each *Action* has an associated class in the source code and is declared in the Struts configuration file. Some features associated with this concept are retrieved from the configuration file, while others are retrieved by parsing the source code.

The source code is searched for classes that subclass the Action class in the Struts base framework. Each of these classes represents a *action* concept. Figure 4.4 shows an example of an Action subclass. The source code is parsed to instantiate many features of the *action* concept. The features that are instantiated by parsing the source file are *className*, *extendsAction*, *messages* and *forwards*. The *className* represents the fully qualified class name of the Action class. The *extendsAction* is set to true for each *action* concept. Lines 93 through 95 represent an instance of the concept *message*. The feature *messages* is a collection of the *messsage* concepts in the *action*. This will be discussed in further detail in Section 4.2.4. Each of the lines 62, 71 and 97 in Figure 4.4 represents a concept instance of *forward*. A collection of concept instances of *forward* that belong to the *action* is represented by the concept *features*. This is discussed in further details in Section 4.2.5.

For each *action* concept, the configuration file is consulted to instantiate rest of the features. Figure 4.5 shows a part of the configuration file where the Actions are declared.

```
 1  package com.taxwide.toastadmin.struts.action;
 2
 3  import org.apache.struts.action.Action;
 .  .....

32  public class LoginAction extends Action {
 .    .....
45    public ActionForward execute(ActionMapping mapping,
46           ActionForm form, HttpServletRequest request,
47           HttpServletResponse response) {
 .      .....
58      if(.....)
59        {
 .          .....
62         return mapping.findForward("main");
63        }
64      else if(.....)
65        {
 .          .....
71          return mapping.findForward("limitedMain");
72        }
 .      .....
93      ActionErrors errs = new ActionErrors();
94      ActionMessage msg = new ActionMessage("error.invalidLogin");
95      errs.add("invalidLogin", msg);
96      addErrors(request, errs);
97      return mapping.getInputForward();
 .      .....
110   }
 .    .....
140 }
```

Figure 4.4: Struts Action

```
 .   .....
 4   <struts-config>
 .    .....
30    <global-forwards>
31     <forward name="home"
32       path="/ShowHomePage.do"/>
33     <forward name="main"
34       path="/ShowMainPage.do"/>
35     <forward name="accessDenied"
36       path="/ShowAccessDeniedPage.do"/>
37     <forward name="limitedMain"
38       path="/ShowLimitedMainPage.do"/>
39    </global-forwards>
 .    .....
47    <action-mappings>
 .     ......
59     <action
60      attribute="loginForm"
61      name="loginForm"
62      path="/Login"
63      input="HomePage"
64      scope="request"
65      validate="true"
66      type="com.taxwide.toastadmin.struts.action.LoginAction"/>
 .     .....
238   </action-mappings>
      .....
280 </struts-config>
```

Figure 4.5: Struts Action Configuration

The actions from the source code are mapped to the actions from the configuration file by comparing the full qualified class name of the source class with the XML attribute *type* of the *action* XML element, contained within *action-mappings* XML element. If a match is successfully made, the feature *declaredInConfig* is set to true. The values of features *attribute*, *form* and *path* are set equal to the attributes *attribute*, *name* and *path* of *action* XML element respectively.

4.2.4 Message

As discussed in Section 4.1.4, the concept *message* is a message to be displayed to the user. A *message* can only exist within an *action*. A collection of all the *message* instances within an *action* are represented by the feature *messages* in *action*.

```
ActionMessages msgs = new ActionMessages();
msgs.add("statusChanged",
         new ActionMessage("msg.statusChanged",
                           cashback.getTwCashbackId(),
                           prevStatus, status));
saveMessages(request, msgs);
```

(a) Message in Code

```
msg.statusChanged = Change of Status for Cashback {0} from previous
                    status {1} to new status {2} succeeded.
```

(b) Message Text in Java Property File

Figure 4.6: Struts Message

Figure 4.6(a) shows the code used to identify the concept instance for *message*. This code is parsed to instantiate various features of the *message*. The method *add* is the entry point for a message into a collection of messages, that are queued in the framework to

be displayed to the user by the method *saveMessages*. We are interested only in the *add* method and its arguments to instantiate the *message* concept. The first argument is used to set the value of the feature *viewId*.

The second argument is an instance of ActionMessage. The arguments passed to the constructor of ActionMessage are used to set the rest of the features of the *message*. The first argument is represented by the feature *messageKey*. This is the key from the Java properties file that stores the text corresponding to the message. The other arguments are used to set the features *arg0*, *arg1* and *arg2*, as the need be. These are used to personalize the messages for the user.

Figure 4.6(b) shows the section of Java properties file related to the corresponding key. The curly bracketed indexes (0, 1, 2) are replaced with the string values returned by *arg0*, *arg1* and *arg2* respectively to personalize the messages for the user.

4.2.5 Forward

Each of the *return* statements at lines 62, 71, 97 in Figure 4.4 represents a concept instance of *forward*. The source code is parsed to find *return* statements. Each of the return statements used the *findForward* method which takes a *String* argument. This argument is used to set the value of the *name* feature of the *form* concept instance.

4.3 Migrate Code

The next major step in the process is when the actual source transformation takes place. The *model* created by source code analysis in last step is used to guide the migration process. The *model* does not capture the application specific source code that contains the application logic. Hence, it is not possible to generate the target application code from scratch. This problem is solved by copying the source application code and making the

necessary modifications only to the code specific to the Struts framework. This leaves the application code, that encapsulates the application logic, unmodified. This is achieved by storing references to the source code in the model. These references map concept instances to the location in the source code. Section 4.3.1 gives more details on source code referencing. The algorithm for the source code migration at a high abstraction level is discussed in Section 4.3.2. Section 4.3.3 and Section 4.3.4 give details of forward mappings used to migrate code to the JSF framework.

4.3.1 Concept Instance References

In the last step we analyzed code to identify concept instances. In this step we will be re-writing the code corresponding to these concept instances so that we replace the dependence on the source framework with the dependence on target framework. This is done while leaving the application specific code as it is. This requires a mapping of the concept instances in the code. This mapping can be achieved by storing references to the source code along with the concept instances in the model.

A reference should facilitate navigation from the model element to the source code related to a concept instance. Hence the reference should provide the source file and the character offset for the concept instance related source code. As we re-write the concept instance's source code, these offsets becomes invalid. Hence, with each re-writing the offsets of all concept instances related to the source file being modified have to be re-calculated.

The FSML base framework provides a utility to achieve this code referencing from the model using *Markers*. These are based on code marking facilities provided by the Eclipse framework to provide for code referencing. Source code modifications do not affect these *Markers* and hence re-calculations do not have to be performed at each step. The FSML base framework extends this functionality for forward mapping purposes. *Markers* were

used extensively in this prototype implementation.

4.3.2 Algorithm

Figure 4.7 show a high abstraction level migration algorithm. This section discusses the algorithm in further details. Section 4.3.3 and Section 4.3.4 discuss the individual steps in the code migration algorithm in further details.

Line 4 declares a collection of compilation units. This collection will hold all the new compilation units that will be created during the migration process that are added to the target application source based on the JSF framework.

Lines 5 through 11 loop over the collection of forms in the model. For each form, a new compilation unit is created in the JSF code which is based on the source of the form's compilation unit.

Lines 12 through 33 loop over the collection of actions in the model. For each action, if a form is associated with the action, then a compilation unit has already been created. The data members and methods of the action are copied to the new compilation unit. Section 4.3.3 discusses this merging of actions and forms in further details. If a form is not associated with the action, the new compilation unit is created for the action, and the data members and the methods are copied to this new compilation unit. The new compilation unit is also added to the target application source code. After this, the code in the *execute* method is rewritten. All actions are required to implement the *execute* method. The Struts-based code is rewritten to JSF-based code. There are three different kinds of code rewrites processed. All statements involving form variables are modified to use the data members of the new compilation unit, and hence the *this* keyword is used. All code related to messages is migrated to use JSF-specific code to generate user messages. All return statements are modified to return Java Strings. All three code migrations are

```
1  migrate(Struts2JSFModel model, SourceCode strutsCode,
2                                 SourceCode JSFCode)
3  {
4       Collection newCUs;
5       for each form in model
6       {
7            Create a new compilation unit in JSFCode,
8                  based on the form's compilation
9                   unit from strutsCode;
10           Add new compilation unit to newCUs collection;
11      }
12      for each action in model
13      {
14           if form is associated with action
15                CompUnit = Find the compilation unit for
16                                        the form in newCUs;
17           else
18                CompUnit = Create new compilation unit
19                                      and add it to newCUs;
20
21           Add all data members and methods of action
22                     to the CompUnit;
23
24           Rewrite form variable to this in execute method;
25
26           for each message in action
27                rewrite message code in execute method;
28
29           for each forward in action
30                rewrite return statement code in CompUnit;
31
32           Rename the execute method in CompUnit;
33      }
34 }
```

Figure 4.7: Migration Algorithm

discussed in further detail in Section 4.3.4.

4.3.3 Actions and Form Beans

Struts has Actions and Form Beans. Actions encapsulate the code that processes the controller logic, while Form Beans are data transfer objects. The Actions execute on the Form Beans and the Form Beans are passed to the Action methods. This is not a very good Object Oriented (OO) approach. One of the most important concepts of OO is to encapsulate data and related methods together in a class. JSF provides a more intuitive way of doing this. JSF only has Managed Beans. Managed Beans have data members that represent request/response data and methods that encapsulate controller logic.

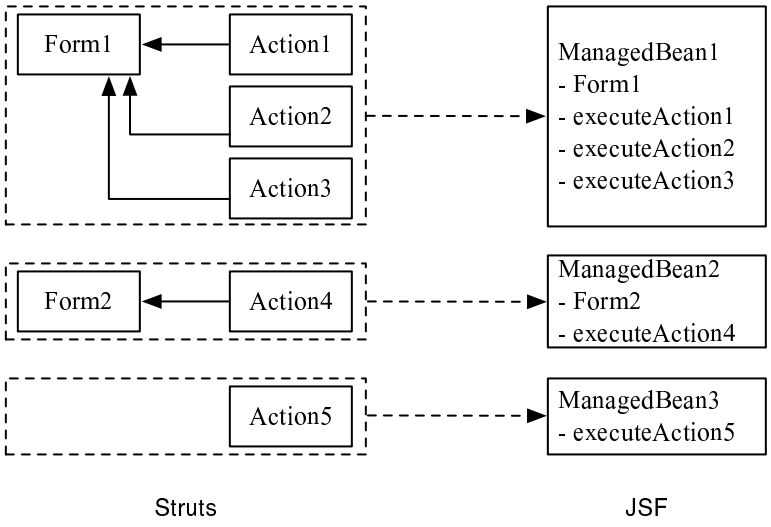

Figure 4.8: Merging Actions and Forms

Hence Struts to JSF code migration requires merging Actions and Form Beans into Managed Beans, as depicted in Figure 4.8. In Struts, one Form Bean is used as a data transfer object for many Actions. Form Beans have data members, while Actions have

methods. Hence, for each Form Bean, a Managed Bean is created in JSF code. After this, the methods from the Actions and their data members are transferred to the Managed Beans. As all Actions have an execute method, this method has to be renamed each time an Action is added to a Managed Bean.

If an Action is not associated with any Form Bean, an independent Managed Bean can be created. This Managed Bean will have exactly same data members and methods as the Action.

4.3.4 Code Migrations

The execute method is the entry point into the Actions. But as many actions are merged together with a form, the execute method is renamed. Moreover, the execute method will be an action listener in JSF. This implies that this method should follow some stringent rules about the method signature. Action's execute method in the Struts framework has the following signature:
public ActionForward execute(ActionMapping mapping, ActionForm form, HttpServletRequest request, HttpServletResponse response)
On the other hand, the JSF action listener should return a String and should have no arguments. There is no condition on the name of the method because it is declaratively associated with the event in the view tier. Hence, the method is renamed by concatenating "execute" and the name of the action. This also guarantees uniqueness, because all Actions have unique names. The ActionMapping argument is not required in the JSF context, as this is just a Java instance of the Action declaration in the Struts configuration file. An ActionForm argument is also not required because the data and the methods have been encapsulated in the same class. The request and response are still required, but these can be obtained using a FacesContext in JSF as follows:

- request = FacesContext.getExternalContext().getRequest();
- response = FacesContext.getExternalContext().getResponse();

Hence, we add this code to the beginning of each execute method to provide a handle to the request and response objects. This facilitates the use of the rest of the code without modification as the same variable names are used as the method argument names.

Figure 4.9 shows an example of the execute method in a Struts action. Figure 4.10 shows the same execute method after migration to JSF. These diagrams will be used in the following sections to explain the code migrations.

Rewrite Form Variable

Unlike Struts, we have data and the related methods in the same class in JSF. In Struts, the execute method had a ActionForm argument, which is type casted to the appropriate form before it is used. Line 11 in Figure 4.9 depicts this type casting. On the other hand, in JSF this data is the part of the same class. Hence we can access these data members directly from within the class, with the use of *this* keyword. Two tasks are performed to carry out this form variable migration.

Firstly, the casting statement, which declares a new form variable and casts the form to the appropriate type, is removed from the code. Line 11 in the Struts code in Figure 4.9 is removed and there is no equivalent in JSF code in Figure 4.10. The variable name (contactForm) is temporarily remembered because it is required for the migrations in the second task.

Secondly, all uses of the form variable are replaced by the *this* keyword. For example, all the usages of *contactForm* in Figure 4.9 are replaced by *this* in Figure 4.10. This completes the form variable rewrites.

```
 1 package ca.uwaterloo.swag.cm.struts.action;
 2
 3 import javax.servlet.http.HttpServletRequest;
 4 .....
 5
 6 public class AddContactAction extends BaseAction {
 7     public ActionForward execute( ActionMapping mapping, ActionForm form,
 8          HttpServletRequest request, HttpServletResponse response) {
 9          if(!haveAccess(request.getSession()))
10               return mapping.findForward("AccessDenied");
11          ContactForm contactForm = (ContactForm) form;
12          ActionMessage commonMessage = new ActionMessage("msg.cmnMsg");
13          HttpSession session = request.getSession();
14          User user = (User) session.getAttribute(IConstants.USER);
15
16          Contact contact = new Contact();
17          contact.setUserID(user.getUserID());
18          contact.setName(contactForm.getName());
19          contact.setEmail(contactForm.getEmail());
20          contact.setTelephone(contactForm.getTelephone());
21          contact.setNotes(contactForm.getNotes());
22
23          if(contactForm.getDob()!=null &&
24                    contactForm.getDob().length() > 0)
25                    contact.setDob(Utility.toDate(contactForm.getDob()));
26
27          boolean contactAdded = dbUtility.addContact(contact);
28          if(contactAdded){
29               ActionMessages msgs = new ActionMessages();
30               ActionMessage msg = new ActionMessage("msg.addSuccess",
31                                                      contact.getName());
32               msgs.add("addContactSuccess", msg);
33               msgs.add("commonMessage", commonMessage);
34               saveMessages(request, msgs);
35               return mapping.findForward("mainPage");
36          }
37
38          ActionMessages msgs = new ActionMessages();
39          ActionMessage msg = new ActionMessage("msg.addFailure",
40                                                    contact.getName());
41          msgs.add("addContactFailure", msg);
42          saveMessages(request, msgs);
43          return mapping.getInputForward();
44     }
45 }
```

Figure 4.9: Struts execute Method Code

```
1 public String exAddContactAction() {
2
3       HttpServletResponse response = Struts2JSFUtility.getResponse();
4       HttpServletRequest request = Struts2JSFUtility.getRequest();
5       if(!haveAccess(request.getSession()))
6       {
7             return "AccessDenied";
8       }
9
10      HttpSession session = request.getSession();
11      User user = (User) session.getAttribute(IConstants.USER);
12
13      Contact contact = new Contact();
14      contact.setUserID(user.getUserID());
15      contact.setName(this.getName());
16      contact.setEmail(this.getEmail());
17      contact.setTelephone(this.getTelephone());
18      contact.setNotes(this.getNotes());
19
20      if(this.getDob()!=null && this.getDob().length() > 0)
21            contact.setDob(Utility.toDate(this.getDob()));
22
23      boolean contactAdded = dbUtility.addContact(contact);
24
25      if(contactAdded){
26            Struts2JSFUtility.saveMessage("addSuccess",
27                  "msg.addContactSuccess", contact.getName());
28            Struts2JSFUtility.saveMessage("commonMessage",
29                                          "msg.cmnMsg");
30            return "mainPage";
31      }
32
33      Struts2JSFUtility.saveMessage("addFailure",
34                  "msg.addContactFailure", contact.getName());
35      return null;
36 }
```

Figure 4.10: JSF execute Method Code

Rewrite Messages

JSF provides a queue for user messages, similar to Struts. But the protocol for interaction is different for the two frameworks. To bridge the gap between the two protocols, a Struts2JSFUtility class is provided. This utility class provides a number of static methods to bridge the different protocols of interaction.

Line 32 in Figure 4.9 queues a message in the Struts base framework. Line 26 in Figure 4.10 is the equivalent for the JSF framework using the Struts2JSFUtility class. In Struts, we first create an instance of ActionMessage and then that ActionMessage is added to the queue. Hence, we need to refer to the ActionMessage creation code, to call the Struts2JSFUtility equivalent. For Line 32 in Figure 4.9, the message is created in Line 30. All the statements to create a message (Line 30), add the message to the queue (Line 32), and save the queue in the framework (Line 34) are replaced by the single line of code, Line 26 in Figure 4.10.

Rewrite Forwards

The execute method in Struts returns an ActionForward instance. This ActionForward encapsulates the string to represent the navigation case, and stores the information on whether to use the redirect or forward to navigate to the next view. On the other hand, JSF moves this information to the configuration file. Hence, we need only the string representation of the navigation case.

The string is returned as it is in the JSF code. Hence, this code migration boils down to replacing the ActionForward instance in all return statements with the corresponding character strings.

Lines 10, 35 and 43 in Figure 4.9 are replaced with Line 7, 30 and 35 respectively. The call to *getInputForward* is replaced with return null statement. Return null implies that

the same view should be displayed to the user.

4.4 Prototype Implementation

As a part of this work, a prototype[1] for automated controller migration has been implemented. This section discusses the technical details of the prototype implementation.

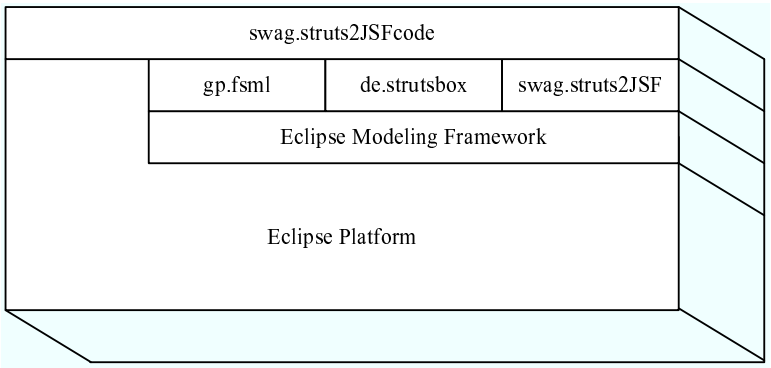

Figure 4.11: Prototype Implementation

The prototype is implemented as an Eclipse plugin based on the Eclipse Platform [5]. The prototype is built on top of many other components. Figure 4.11 shows the software stack used to implement the prototype.

The base of the prototype is the Eclipse Platform. Eclipse is an extendible open source development platform. It has built-in support for Java development, but it is being used for many other development purposes. This has been possible because of the extendibility of the platform. It has a large support community and is growing at a fast pace.

The Struts2JSF prototype consists of two Eclipse plugins: swag.struts2JSF, discussed in further detail in Section 4.4.1, and swag.strut2JSFcode, discussed in further detail in

[1]This section is based on the joint work of the author and M. Antkiewicz. The prototype implementation is based on the base FSML framework.

Section 4.4.2.

4.4.1 swag.struts2JSF Eclipse Plugin

The first step in the automated controller migration based on FSML is to define the FSML. The plugin swag.struts2JSF is used to define the FSML for migration and to generate the required code from the meta-model of the FSML. This has been implemented using the Eclipse Modeling Framework (EMF) [6], defined by the Eclipse Foundation [5] as follows:

> "Eclipse Modeling Framework (EMF) is a modeling framework and a code generation facility for building tools and other applications based on a structured data model."

EMF has three fundamental pieces:

- EMF - provides facilities for describing the meta-model on which the models are based. The meta-model is referred to as *ecore* model, and is the core of the EMF framework.
- EMF.Edit - provides base framework classes for building editors for models.
- EMF.Codegen - provides a code generation facility that generates Java code for modell manipulation.

The EMF core framework is used to define the meta-model for the Struts2JSF FSML in *Ecore* format. EMF.Codegen is used to generate the code required to manipulate the models based on the Struts2JSF FSML. The code generated is used extensively by swag.struts2JSFcode wizards to analyze application code to generate a model instance and to migrate the code to the target framework. EMF.Edit is used to create editors

to visually manipulate the generated model. Although we do not modify the models in this work, but this can be particularly useful if we want to customize the model before migration.

4.4.2 swag.struts2JSFcode Eclipse plugin

The two steps in automated controller migration are *Analyze Code* and *Migrate Code*. The swag.struts2JSFcode plugin provides two wizards that perform the code analyses and migration. The plugin depends on two other plugins: gp.fsml and de.strutsbox.

The de.strutsbox plugin [40] itself is based on EMF and it provides facilities to manipulate the Struts configuration files. These facilities have been used extensively in swag.struts2JSFcode for reading the Struts configuration files.

gp.fsml is the base framework for implementing FSMLs. The gp.fsml plugin for Eclipse provides many utility classes that facilitate code analysis, manipulation and marking. Both the code analysis and code migration components use these facilities extensively.

Code Analysis

Code Analysis is performed by the New Wizard, which uses the *NewWizards* extension point from the base Eclipse framework. The code in this section depends largely on the gp.fsml plugin as it extends the Queries base class provided by the plugin. The Queries base class provides facilities for searching for all the method invocations of a given method, retrieving the arguments being passed to the method, generating different IDs for concept instances, getting Abstract Syntax Trees of compilation units, etc.

Code Analysis is performed by selecting a Struts project and then selecting the Struts2JSF Analysis Wizard from the New menu. The wizard asks for the name of the model file, and generates a model based on the Struts2JSF FSML by analyzing the project code.

While analyzing the Struts code, the code is marked using the Marker utility provided by the gp.fsml plugin. This utility exploits the marker facilities provided by the Eclipse platform, and provides ways to uniquely identify and map each marker to the associated concept in the model.

Code Migration

Code Migration is performed by the Export Wizard, which uses the *ExportWizards* extension point from the base Eclipse framework. This wizard asks the user to select the project in which to create the new JSF code. The code generation largely utilizes the facilities provided by the CodeTransformations class in gp.fsml. CodeTransformations provides facilities for creating new compilation units, adding new types to the compilation units, adding import declarations, adding extends declarations, etc.

The code marking facilities of gp.fsml are also utilized extensively to map concept instances to the source code.

4.5 Chapter Summary

This chapter discussed the technical details of a prototype FSML implemented for migration purposes. This prototype was used for a migration case study. Chapter 5 discusses the migration of the view component. The case study is discussed in Chapter 6.

Chapter 5

View Migration

As discussed in Chapter 3 the migration activity involves controller migration, view migration and the migration of configuration files. Configuration file migration was discussed in Chapter 3. The controller migration process and the prototype for automated controller migration was discussed in Chapter 3 and Chapter 4. This chapter discusses view migration, which is a manual process.

Applications based on the Struts framework use tag libraries that generate the markup for generating responses. Applications based on the JSF framework use a similar approach by using tag libraries to represent user interface components. Markup code is generated by the *renderer* to generate responses. From an application developer point of view, the two approaches are semantically different because the JSF tags represent components that have a server side state. The migration process involves replacing the Struts tags with equivalent JSF tags. Some of the Struts tags have a one-to-one mapping with JSF tags, while others do not. Straight forward tag mappings are discussed in Section 5.1. When there is no one-to-one mapping, certain combinations of Struts tags are replaced by a single JSF tag. Particular patterns of Struts tag combinations represent JSF components. Hence this migration comes down to the problem of *componentization*. This is discussed in further

detail in Section 5.2.

Tiles is an built-in templating technology for the Struts framework. On the other hand, JSF does not have a similar templating technology. Tiles was born as an independent project [46], that was later included in Struts. As such, the Tiles plug-in is independent of the Struts base framework, and it can also be used with JSF. Since applications based on Struts use Tiles, it is recommended to use Tiles with JSF. We will discuss the problems arising from this and how to handle these problems in Section 5.3.

5.1 Struts2JSF Tag Library Map

Struts has four different kind of tag libraries: *html*, *bean*, *logic* and *nested*. Each Struts tag library tag is preceded by the name of the tab library (html, bean, logic or nested) plus a colon. JSF has two tag libraries: core and html. The core tag library tags are preceded by *"f:"* and the html tag library tags are preceded by *"h:"*.

Table 5.1: Struts2JSF Tag Library Map

Struts Tag	**Equivalent JSF Tag**
html:text	h:inputText
html:password	h:inputSecret
html:hidden	h:inputHidden
bean:message	h:outputText
html:select	h:select*
html:option	f:selectitem
html:optionsCollection	f:selectitems

Continued on next page

Table 5.1 – continued from previous page

Struts Tag	**Equivalent JSF Tag**
html:textarea	h:inputTextarea
html:button	h:commandButton
html:checkbox	h:selectBooleanCheckbox
html:image	h:graphicImage
html:form	h:form
bean:message	h:message

Table 5.1 shows a tag library map from Struts to JSF migration. It presents only the most basic tags. A detailed discussion follows for each of the tag mappings, along with a discussion of more complex features and attributes of each tag.

Migrating html:form to h:form

These tags are responsible for generating an HTML *form* element for user input. The html:form tag in Struts can be replaced by h:form tag, and is followed by the replacement of all the code (both raw HTML and Struts tags) within the html:form with code for equivalent JSF components.

The most important attribute from the migration perspective is the *action* of the Struts html:form tag. This associates an HTML form with a Form Bean instantiated and managed by the Struts base framework. In JSF, h:form cannot be directly associated with a Managed Bean. Instead each of the elements in the h:form tag have to specify the Managed Bean. This requires a manual mapping of a Form Bean in Struts to a Managed Bean in JSF, and

then adding the Managed Bean in the *value* attribute using standard JSP 2.0 Expression Language [23] in JSF.

Migrating html:text to h:inputText

HTML provides the *input* element to support basic user input. As the view tier in Struts and JSF is responsible for generating HTML-based markup, both provide tags and user interface components that are analogous to these HTML *input* elements. The HTML *input* element has a *type* attribute that can have any of the following values: *text*, *password*, *checkbox*, *radio*, *submit*, *reset*, *file*, *hidden*, *image* and *button*. There are Struts and JSF tags that represent all of these types of *input* elements.

The following types have one-to-one mapping in both Struts and JSF: *text* (html:text, h:inputText); *hidden* (html:hidden, h:inputHidden), *password* (html:password, h:inputSecret), and *button* (html:button, h:commandButton). These tags can be directly replaced.

The *property* attribute identifies the data member of the Form Bean in Struts that is associated with the value in the HTML element. On the other hand, JSF uses the *value* attribute to identify both the Managed Bean and the data member associated with the value of the HTML element. Therefore, migration requires a manual mapping of the Form Bean in Struts to the correct Managed Bean in JSF, based on the *action* attribute in the html:form tag in Struts.

Migrating bean:message to h:outputText

Both Struts and JSF provide localization support. This is achieved using these tags. Both tags emit HTML markup to generate verbatim text in the present locale. The Struts bean:message tag uses the *key* attribute to specify the key for the Java properties file. JSF uses the *value* attribute in the Expression Language to set the text to be displayed. The

value attribute and localization in JSF is discussed later in this section, in the subsection titled "f:loadBundle".

Migrating html:select to h:select*

These tags are used to generate the HTML *select* element. This element is used to display multiple or single selection menus, list boxes and checkboxes. For each option to be listed, the HTML *option* element is used. The Struts tags used to work with *select* element are the following: html:select and html:multibox.

The html:select tag has a *multiple* attribute to generate multiple selection list boxes. The html:multibox tag is used to generate a list of checkboxes to select from. JSF has a number of different selection tags, starting with h:select. The ones that are analogous to the Struts html:select tag are the following: h:selectOneListBox, h:selectManyListBox, h:selectOneMenu and h:selectManyMenu. The Struts html:select tag should be replaced by one of these tags depending on if the *multiple* attribute is set to true or false.

The html:multibox tag in Struts generates a group of checkboxes. This tag should be replaced by the h:selectManyCheckbox JSF tag. Another Struts tag related to selection is html:radio. It is replaced by the h:selectOneRadio tag in JSF.

The HTML *select* element contains many *option* elements. The Struts and JSF tags related to generating options for the *select* tag are discussed in next sub section.

Migrating html:option to h:selectitem

These tags are used by Struts and JSF to generate HTML *option* elements. The Struts tags related to this are: html:option, html:options and html:optionsCollection. The JSF tags related to this are: f:listitem, f:listitems. The Struts html:option tag has a straightforward migration to f:listitem, and the html:options and html:optionsCollection should be

migrated to f:viewitems tags.

This tag migration involves some controller modifications as well. The Struts tags can deal with any Java collection. In JSF, the f:listitem and f:listitems expect a Java Collection, array or Map of type SelectItem (in javax.faces.model). A Wrapper design pattern [57] can be used to accomplish this, to avoid modification of the existing controller code.

These *label* and *value* attributes of the Struts tags are analogous to the *itemLabel* and *itemValue* tags in JSF.

Migrating html:button to h:commandButton

These tags generate the HTML elements related to the input types button, submit and reset. The Struts tags related to these are: html:button, html:submit and html:cancel. JSF has only one tag to which all of these Struts tags should be migrated: h:commandButton. In Struts, the *action* attribute in html:form tag specified which *execute* method should be called on submission. But in JSF a method is associated with the *click* event of an h:commandButton using the *action* and *actionListener* tags. This is one of the major differences between the two frameworks. In JSF this provides for more flexibility in terms of method call backs.

Migrating to f:loadBundle

The f:loadBundle tag in JSF has to be used in the f:view tags to support localization and personalization using Java property files. In Struts, this is done at an application level, by declaring this in the configuration file. The f:loadBundle tag is used to load a bundle of properties from a Java properties file for the specified locale.

```
<f:loadBundle basename="java.properties.file" var="msg"/>
```

The *basename* attribute specifies the name of the Java properties file and *var* attribute specifies the name of the variable to be used by other tags to access the text from the properties file. The html:outputText uses this as follows:

```
<h:outputText value="#{msg.[key]}"/>
```

Hence, this f:loadBundle should be added at the top of each f:view JSF tag. The value of the *basename* attribute is read from the *message-resources* XML element in the Struts configuration file.

Migrating HTML Tables to h:panelGrid

Some of the JSF tags are incompatible with regular HTML tags. This is especially true for the HTML table related tags (<table>, <tr> and <td>) inside of the h:form component tag. This can be avoided by using h:panelGrid tag in JSF. The h:panelGrid provides facilities to generate HTML tables.

5.2 Componentization

The goal of view migration is to have JSP pages that generate identical looking responses to the user. Componentization, in Struts to JSF migration, refers to replacing combinations of Struts tags with a single or a combination of JSF components that represent the same markup element when responses are generated.

The JSF framework provides extension points to add new user interface components. This has created a marketplace for user interface components. Some of the most popular libraries for JSF user interface components are MyFaces [23], Oracle ADF Faces [24] and JScape's WebGalileo Faces [25]. Also, the JSF framework provides some basic components that are included in the base framework. Hence, componentization is largely dependent on

the choice of JSF implementation and component libraries. In this work, we discuss only the components that are available in the reference implementation of JSF by Sun Microsystems. It is highly recommended to identify opportunities to replace repeating patterns of Struts tags with equivalent custom JSF components [59]. Custom JSF components are beyond the scope of this work.

5.3 Tiles and JSF

Templating technologies provide for reuse of view code. In JSP this is accomplished using JSP includes. JSF does not provide a templating technology. Tiles, being an independent technology, can be used with JSF. The approach to using Tiles with JSF is again largely dependent on the choice of JSF implementation. Some implementations (*ex.* MyFaces) provide a view handler that is Tiles-aware, and hence simplifies the Tiles and JSF integration [38]. Most JSF implementations do not provide a built-in Tiles and JSF integration facility. JSF has a view handler extension point that can be extended to provide this facility. As JSF and Tiles integration is not a part of the JSF specification, and is important only for the Struts to JSF migration, this section discusses using Tiles with JSF when the implementation does not provide an integration facility. Sun Microsystem's reference implementation, for example, does not provide any such facility. This section discusses how to use Tiles with JSF without using any extension points of the JSF framework.

The Tiles plugin has two major components: the Tiles configuration file and the Tiles Servlet. In the migration from Struts to JSF, the following steps should be taken to use Tiles with JSF:

- Add a TilesServlet configuration in the JSF deployment descriptor (web.xml).
- Create a JSP page for each Tile.

- Migrate the layout to use the combination of JSF and Tiles.

With Struts, Tiles is used as a plug-in and is configured using the *controller* and *plug-in* XML elements in the main Struts configuration file as shown in Figure 5.1(a). These configuration element are not mirated to the JSF configuration file. Instead, we declare a TilesServlet in the application deployment descriptor as shown in Figure 5.1(b). This enables the use of Tiles in JSF.

```
......
<controller
      processorClass="org.apache.struts.tiles.TilesRequestProcessor"/>
.....
<plug-in className="org.apache.struts.tiles.TilesPlugin">
      <set-property property="definitions-config"
          value="/WEB-INF/tiles-defs.xml" />
</plug-in>
.....
```

(a) Tiles Configuration in Struts

```
......
<servlet>
  <servlet-name>tiles</servlet-name>
  <servlet-class>org.apache.struts.tiles.TilesServlet</servlet-class>
    <init-param>
      <param-name>definitions-config</param-name>
      <param-value>/WEB-INF/tiles-defs.xml</param-value>
    </init-param>
    <init-param>
      <param-name>definitions-parser-validate</param-name>
      <param-value>true</param-value>
    </init-param>
  <load-on-startup>1</load-on-startup>
</servlet>
.....
```

(b) Tiles Configuration in JSF

Figure 5.1: Example of Migrating Tiles from Struts to JSF

Because Struts has built-in support for the Tiles framework, it is possible to forward HTTP requests directly to the tile in the Tiles plugin. In JSF, navigation works at page level. It is not possible to navigate to a tile from the Tiles framework. To accomplish this kind of navigation, we need to create a new JSP page that includes a tile. This has to be done for each tile in the Tiles configuration file. This JSP page simply uses the insert tag from Tiles tag library to include the tile. The navigation graph can now be written in terms of these JSP pages. The code for such JSP pages will look similar to the following code.

```
<%@ taglib uri="/WEB-INF/struts-tiles.tld" prefix="tiles" %>
<tiles:insert definition="HomePage" flush="false" />
```

This JSP inserts a tile called HomePage into the JSP page. In Struts this would not be required as the request can be directly forwarded to the HomePage tile. A simple script can be used to generate each page by reading all the tile in the Tiles configuration file. A Java program was used for this purpose in the case study in this work.

The most critical part of using Tiles and JSF is to modify the layout JSP files. The power of the Tiles framework comes from the fact that it allows us to not only reuse the JSP page contents, but also allows us to reuse the layout by defining the layout in separate JSP files. The view migration applies to all the JSP pages that hold the content, whereas layout migration here refers to the migration of the JSP pages responsible for the layouts in Tiles. Figure 5.2 shows the code of such a layout JSP in Struts and in JSF.

The HTML *table* shown in Figure 5.2(a) and the related markup tags have been replaced by the h:gridLayout tags in JSF shown in Figure 5.2(b). Each of the tiles:insert statement should be enclosed in a separate f:subview. This means that an independent component tree exists for each tile being inserted on the server side.

```
<body>
  <tiles:insert attribute="css"/>
      <table width="100%" border="1">
            <tr>
                  <td colspan="2" align="center">
                        <tiles:insert attribute="header"/>
                  </td>
            </tr>
            <tr valign="top">
                  <td align="left" width="20%">
                        <tiles:insert attribute="navigation"/>
                  </td>
                  <td align="left">
                        <tiles:insert attribute="main"/>
                  </td>
            </tr>
            <tr>
                  <td colspan="2" align="center">
                        <tiles:insert attribute="footer"/>
                  </td>
            </tr>
      </table>
  </body>
```

(a) Tiles Layout File in Struts

```
<f:view>
<f:loadBundle
   basename="com.taxwide.toastadmin.faces.ApplicationResources"
   var="msgs"/>
<tiles:insert attribute="css" flush="false"/>
<h:panelGrid columns="2">
  <f:facet name="header">
    <f:subview id="header">
      <tiles:insert attribute="header" flush="false"/>
    </f:subview>
  </f:facet>
  <f:subview id="navigation">
    <tiles:insert attribute="navigation" flush="false"/>
  </f:subview>
  <f:subview id="main">
    <tiles:insert attribute="main" flush="false"/>
  </f:subview>
</h:panelGrid>
</f:view>
```

(b) Tiles Layout File in JSF

Figure 5.2: Example of Migrating a Tiles Layout

5.4 Chapter Summary

This chapter presented guidelines for migrating a view component based on Struts to a view component based on JSF by migrating the JSP tags. The process discussed in Chapter 3 and Chapter 4 and the view migration technique discussed in this chapter was used to carry out a migration case study. Chapter 6 discusses the case study and presents the lessons learned from the migration task. Chapter 7 concludes this work.

Chapter 6

Migration Case Study - ToastAdmin

Taxwide, Inc. [42] is a tax preparation and accounting firm that has more than 15 locations throughout the Greater Toronto Area. TOAST stands for Taxwide Operations And Security Tool. TOAST is a software system under development at Taxwide Inc. and is largely used for streamlining the Instant Tax Refund business process. Figure 6.1 shows a high level architectural diagram of the TOAST system.

The two major server side software systems in TOAST are Toast Web Services and ToastAdmin. At the heart of the TOAST system is the Toast Database. This database system is based on MySQL. The Toast Web Services provide a secure interface to the back end database system. The Web Services [62] are implemented using J2EE technologies. ToastAdmin is a web application that is used for the administration of the system. This web application is based on J2EE technologies and the Struts framework.

On the client side, a light weight browser application is used to access ToastAdmin. ToastClient is the application used by employees to carry on various day to day operations at Taxwide Inc. This is a desktop application based on the Java Swing GUI framework that communicates with the rest of the system through Toast Web Services. Swing is distributed as a part of Java SE Software Development Kit [18].

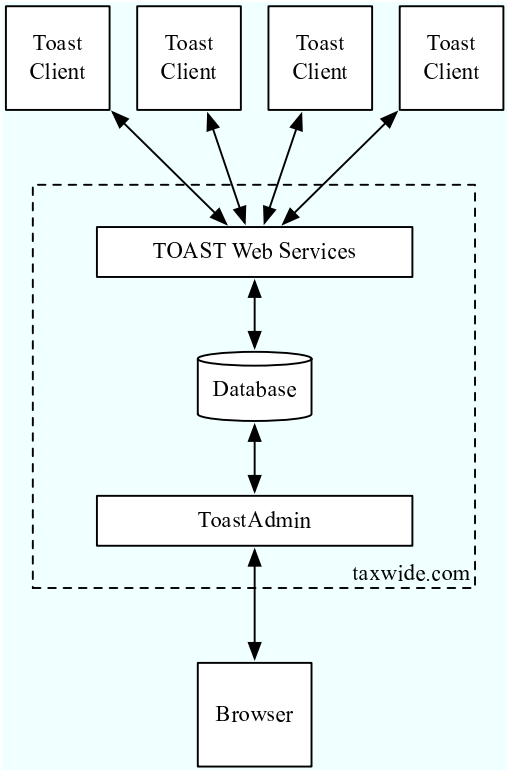

Figure 6.1: Architecture of the Taxwide Operations And Security Tool (TOAST)

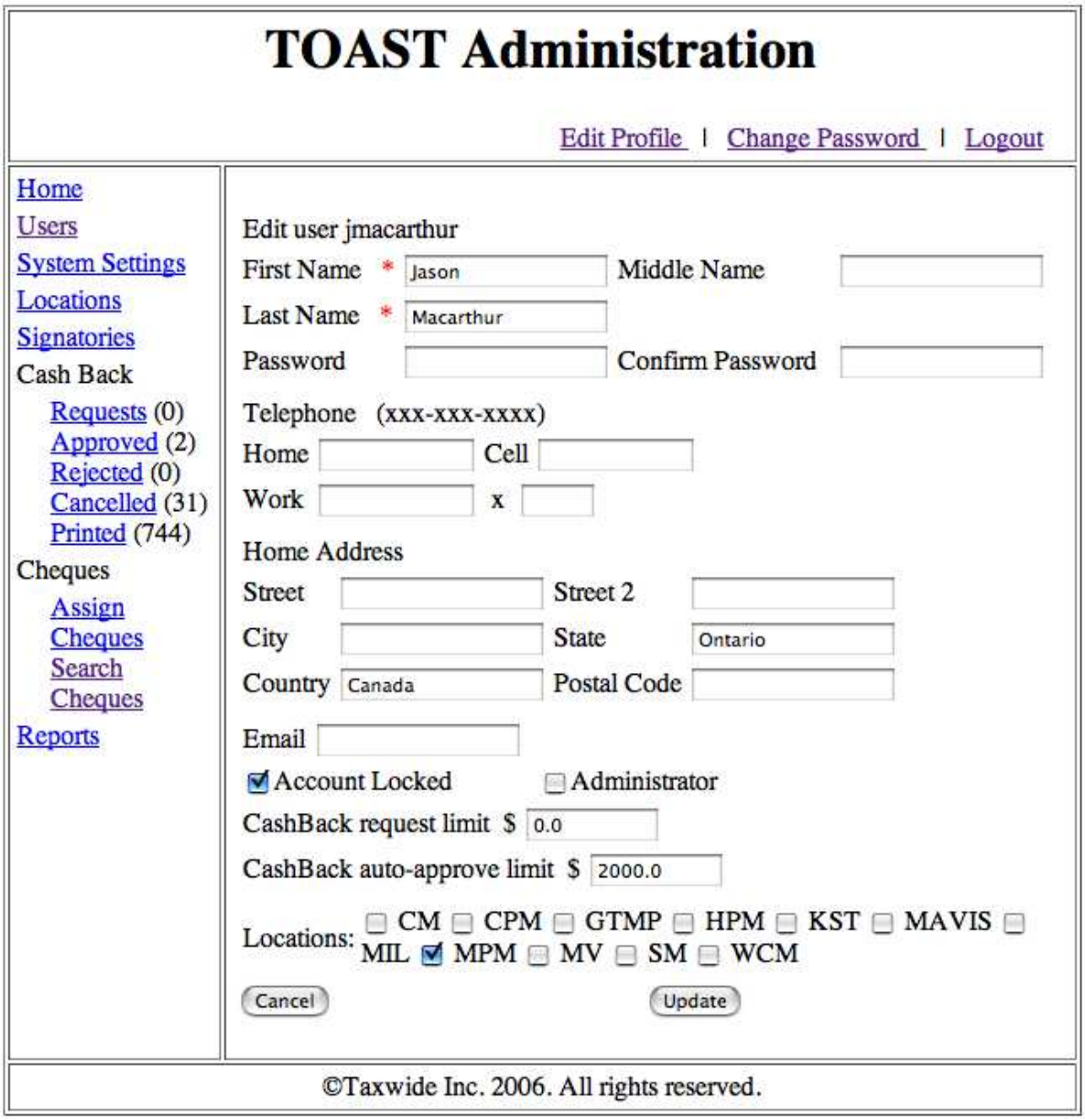

Figure 6.2: ToastAdmin Screenshot

The component of interest in this system is ToastAdmin[1]. Figure 6.2 shows a screenshot of the administration application. As a part of this work, ToastAdmin was migrated to the JSF framework using the complete migration strategy discussed in Section 3.1. Controller migration was carried out using the Eclipse plug-in discussed in Section 4.4. View migration was carried out manually using the techniques discussed in Chapter 5. This chapter discusses the migration of ToastAdmin in further details.

6.1 Controller Migration

Section 4.4 discussed the prototype FSML that was implemented as a part of this work. The prototype was used to carry out the controller component migration to JSF in a semi-automated fashion in this case study. The configuration files were migrated manually after the automated migration of the Java code of the Controller component.

The size of the Struts version of the ToastAdmin application is 19,566 lines of Java code. The controller component contributes 4,697 lines of code while the model component contributes 14,869 lines of code. The JSF version of the ToastAdmin application has a total of 18,398 lines of Java code. The controller component contributes 3,529 lines of code, while the model component contributes 14,869 lines of code. The size of the model component is unchanged, while the controller component is smaller in size for the JSF version. However, this difference in lines of code is minor and insignificant. The lines of code were calculated using the metrics plugin, which is a part of the Eclipse platform. The plugin provides for lines of code while ignoring comments and blank lines.

[1] A permission was obtained from Taxwide Inc. to use the application for this case study.

Table 6.1: Form Beans (in Struts) to Managed Beans (in JSF) Mappings

Form Bean, Actions	**Managed Bean**
userForm /AddUser	User
loginForm /Login	Login
userList /ShowUsers	UserList
locationList /ShowLocations	LocationList
editUserForm /ShowEditUserPage /UpdateUser /ShowDeleteUserPage /DeleteUser	EditUser
locForm /AddLocation /ShowEditLocationPage /EditLocation	Location
editLocForm /ShowDeleteLocationPage /DeleteLocation	EditLocation

Continued on next page

Table 6.1 – continued from previous page

Form Bean, Actions	Managed Bean
signatoryForm /ShowAddSignatory /AddSignatory	Signatory
signatoryListForm /ShowSignatories	SignatoryList
assignChequesForm /AssignCheques	AssignCheques
chequeListForm /SearchCheques	ChequeList
cashbackListForm /ShowCashbackList	CashbackList
setCashbackStatusForm /ShowSetCashbackStatus /SetCashbackStatus	CashbackStatus
editSignatoryForm /UpdateSignatory /ShowEditSignatory	EditSignatory
overviewForm /ShowMainPage	Overview
sysSettingsForm /ShowSysSettings /UpdateSystemSettings	SysSettings

Continued on next page

Table 6.1 – continued from previous page

Form Bean, Actions	**Managed Bean**
newRequestForm /RequestCashback	Request
limitedCashbackListForm /ShowLimitedCashbacks	LimitedCashback
ReportForm /GenerateReport	Report

Table 6.1 shows the Actions and Form Beans of Struts version of ToastAdmin and the equivalent Managed Beans of the JSF version. The table does not list the Actions and the equivalent Managed Beans for those Actions that do not have an associated Form Bean. All such Actions have an equivalent Managed Bean.

The names of the Managed Beans are given manually. These names are entered when the configuration files are migrated. Each Managed Bean was declared in the configuration file, and the table shows the names that were given to these Managed Beans. The results of the migration were encouraging overall except for some minor issues that were discovered. These issues are discussed in the next section along with some discussion of the solutions to these issues.

6.1.1 Issues

Action Class Hierarchy

In ToastAdmin, the Action classes had a richer class hierarchy than just all the Actions extending the abstract base Action class provided by the Struts framework. For example

in ToastAdmin we have a class BaseAction which is extended by each of the other action classes. This BaseAction provides a method *hasAccess* that checks to see if the associated *HttpSession* has the required privileges. Figure 6.3 shows this scenario.

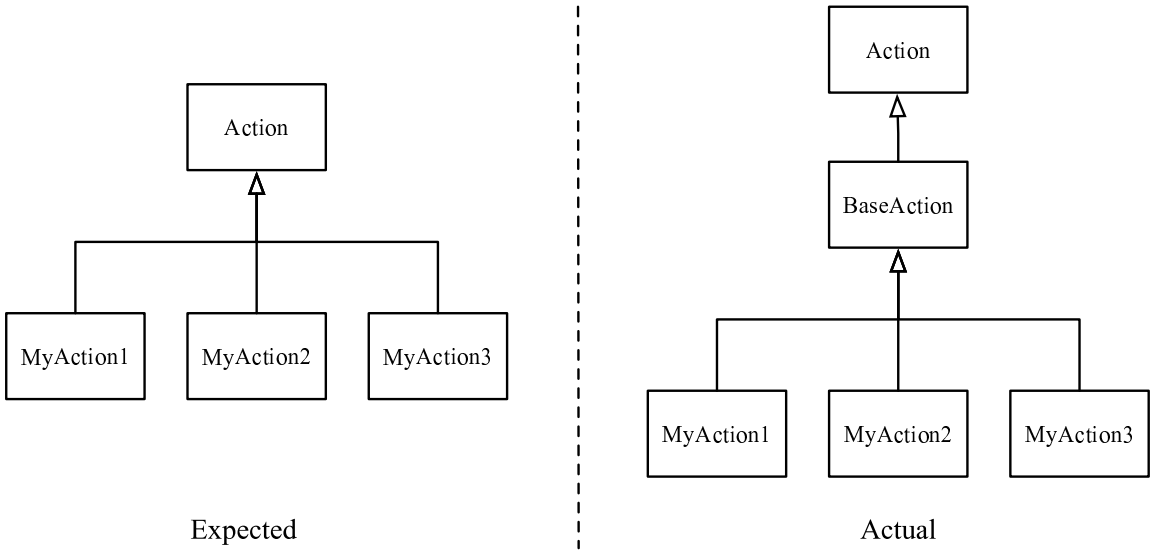

Figure 6.3: Expected and Actual Action Class Hierarchy in ToastAdmin

During migration, the Actions actually become methods of the Managed Beans. Moreover the Form class hierarchy has already been implemented as a Managed Bean class hierarchy in the migrated code and Java does not support multiple inheritence. Such an Action class hierarchy poses some serious automation challenges. Using the FSML we did capture the framework specific abstractions. But our approach does not capture the details of application-specific code. Hence, such an extensive application specific class hierarchy for Actions is not handled by our approach.

It is possible to solve this problem by capturing the Action class hierarchy in the FSML and then flatting out the class hierarchy in the migration process. This would mean code repetition, but would get us to a working Controller component. The code can later be

optimized and restructured to have a proper class hierarchy.

Data Validation

A Form Bean in Struts has a *validate* method that validates user input data before it invokes Actions on that data. This framework abstraction was not captured in the prototype FSML. Every Action has an attribute *validate* that can be set to true or false. This can easily be captured in the FSML, and the Analyze Code and Migrate Code stages can be modified to call the validate method in the execute method of an action before invoking the rest of the code. For this case study, we implemented this validation manually achieved this.

There were 31 Actions and 29 Form Bean in the Struts-based application that were migrated to 19 Managed Beans in the JSF-based application. This migration was not fully automated because of the issues discussed earlier. The manual steps taken to migrate to a working controller based on JSF are as follows:

- A library component was created that provided the common functionality that Actions inherited from their super classes. This task was simple, as there was only one method from the super class that was in all the Action subclasses. This method was moved to a library class as a static member. Code in each action was modified to call the new library method instead of superclass method. This task took approximately one person day.

- A JSF configuration file was written to configure the Managed Beans and migrate the navigation rules. This task took approximately one person week. The effort involved in this migration task was discussed in Chapter 3 in detail.

- Data validation was migrated to either a separate library or to the corresponding

Managed Beans. For simple validation rules, where the JSF core framework validation was sufficient, validation was moved to the view component. Otherwise, validation was moved to the library and appropriate library calls were inserted. This talk took approximately one person week.

Hence, the controller migration took approximately two person weeks in total.

6.2 View Migration

The controller component was migrated before the view component. Hence, while the view component was being migrated, a working Controller component was available to perform page by page testing of the web application.

Tiles

As discussed earlier Tiles is an independent templating technology that is an addition to Struts but works well with JSF. It is highly recommended to use Tiles as the templating technology for JSF-based applications. In this case study, the Struts-based application extensively used Tiles. We used it as our templating technology for the JSF application as well. Figure 6.4 shows the layout of the ToastAdmin application. It has a HEADER at the top. There is a NAVIGATION menu and the MAIN content section beside each other. There is a FOOTER at the end of the layout. Each set of JSP pages that fills all the four sections of the layout make up one web page or one Tile. The Tiles configuration file is needs no modification for JSF. But as discussed in Chapter 5, we need to create a new JSP page for each tile, where each page simply includes the Tile definition in it.

The total number of new JSP pages created, one for each tile, in ToastAdmin is 24. These JSP pages are the vertices of the navigation graph in JSF. This migration was

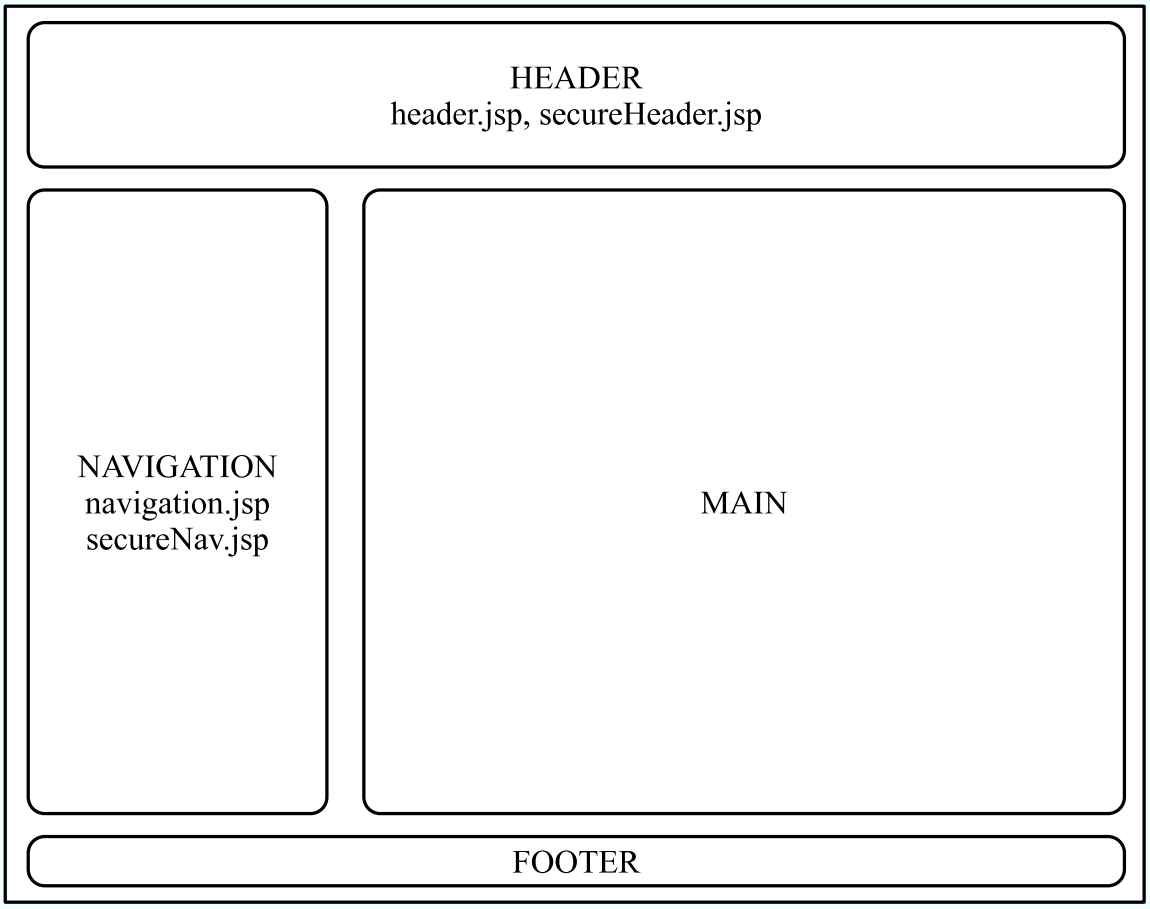

Figure 6.4: ToastAdmin Layout

carried out fully automatically by using a Java program. The time taken to write the Java program was approximately one person day. The time taken to execute the program to carry out tiles migration in ToastAdmin was negligible.

JSP Pages

After the migration of the tiles, we need to migrate each JSP page that uses them. This involves replacing the Struts tag libraries with the JSF user interface components as discussed in Chapter 5. One of the major issues encountered in this migration was the replacement of the Struts logic tags. These tags provide flexibility and control in the Struts view tier. In JSF we have more abstract user interface components that make it easier to construct pages but result in a loss of flexibility and control.

For example, in ToastAdmin we provide a list of users that can access the system. If the user is an administrator, the *Admin* text is displayed next to its name in the list of users. In Struts this is achieved using a simple logic tag to check if the user is an administrator. This is not supported in JSF because JSF does not provide logic tags. JSFs tags are the user interface components with server side state. There are two different ways of achieving the same effect in JSF.

One way is to use the *rendered* attribute of the JSF tags to display or not display any JSF component. This is a straightforward approach that works for the simple problem we just discussed. It also requires modification of the associated Managed Bean in certain cases. But this might not work for more complex uses of the logic tags. In that case, the second approach would be to use custom components. JSF supports customer user interface components. We used the Datatable user interface component to display the list of users. It is possible to extend this user interface component and add the needed functionality to the component. This would mean that code would move from the view

tier to the controller tier. This is preferred in certain cases where the web design team does not have Java skills, but might not be preferable in other scenarios where the web design teams need more flexibility and have the required skills. In ToastAdmin, we solved this problem using the *rendered* attribute, as it served our needs well.

There were a total of 33 JSP pages that were migrated to JSF pages. This took approximately 4 person weeks. A large proportion of the time was spent in modifying the data model (Managed Beans) where the JSF tags did not provide sufficient flexibility to replace Struts tags. This was particularly true where Struts logic tag library had been used. In ToastAdmin, this problem was faced in 6 JSP pages when the HTML table was used in conjunction with the Struts logic tags.

JSF and JSTL

It is possible to use JSF and JavaServer Pages Standard Tag Library (JSTL) together, but it is not possible to mix JSF and JSTL. JSTL provides many logic and iteration library tags, but the only way JSTL can access the data from a JSF controller is by using the Managed Beans. Using expression language in JSTL it is possible to directly access Managed Beans because these are just regular JavaBeans. On the other hand, it is not possible to use any JSTL tags within JSF components. That does not work because these two tag libraries are executed at different times during request processing life cycle of JSF, and do not produce the desired result. Hence, JSTL and JSF can be used together if a clear line of separation is drawn in terms of using the tag libraries.

6.3 Lessons Learned

The controller component migration should be carried out before the view migration. If the controller component is already migrated, then unit testing for each web page can

be performed during the migration process. Controller migration should be followed by configuration file migration, followed by view migration. The configuration files provide for loose coupling between the controller and the view components. Hence it is required for the view component to communicate with the controller component.

View migration should be carried out incrementally, while testing each page individually against the controller. This validates the migration process as the migration moves ahead.

The Tiles plugin works with the JSF framework and is independent of the Struts framework. Tiles should be used as the preferred templating technology when migrating from the Struts framework. This minimizes the migration work involved in the view tier and hence lowers the risks. If another templating technology has to be used, then it is recommended to start by migrating to the JSF framework using Tiles. Once a working view component is available, then migrate to the selected templating technology.

Struts logic tags can be replaced by JSTL logic tags as the JSF view tier does not provide for any logic tags. But JSTL tags do not work within the scope of a JSF component in a JSP page. Hence, extending the basic JSF components to create custom component that achieve the desired logic is recommended.

The *flush* attribute of each of the *tiles:insert* tags should be set to *false*. The *true* value throws an exception. This is a minor incompatibility issue when using Tiles with JSF.

6.4 Chapter Summary

This chapter discussed the results of a migration case study performed using the process and techniques discussed in previous chapters. Code metrics, the issues related to the controller migration, the view migration challenges related to the JSF tags and the lessons learned from the case study were discussed. Chapter 7 concludes this work.

Chapter 7

Conclusion

The Struts framework has become the de-facto standard for J2EE-based web application development. The JSF framework is the new framework which has similarities to Struts, but extends it by providing a more flexible controller component. It improves on Struts by offering an event-oriented approach, an extendible user interface component framework, and improved tool support. Existing Struts based applications with a requirement for a rich user interface should consider migrating to the JSF framework. The implementation choice should be based on the richness of the user interface component library.

In Chapter 3 we discussed two strategies for migration to JSF: the Complete migration strategy and the Partial migration strategy. Section 3.2 discussed the advantages and disadvantages of the partial migration strategy. The partial migration strategy aims at an application that uses the JSF framework for the user interface components in the view component and the Struts framework for the Controller component. It requires less migration effort since the Controller component is unmodified. But the application can only use the components provided by the Struts-Faces Integration Library (SFIL) [38]. This strategy does not exploit the market place of the user interface components available for view component in JSF. Unfortunately, the partial migration strategy results in an

application dependent on two separately developing and evolving technologies. Any new features of JSF that can not be supported by Struts can not be used by the application. The declarative navigation handling in the controller component and the data conversion extension points in the View component are examples of such features. Hence, the complete migration strategy is recommended to produce a pure and native JSF application.

The research objective of this work was also to explore opportunities for automating the migration from Struts to JSF. The migration was divided into controller migration, view migration and configuration file migration. Controller migration is similar to API migration and was automated using a framework-specific modeling approach based on a special purpose Framework Specific Modeling Language (FSML). The FSML used in this work captures the common concept abstractions for the Struts and JSF frameworks. This enables reverse engineering of a Struts-based controller to a model and then forward engineering (code migration) to a JSF-based controller from the model. Framework modeling has been used for round trip engineering of framework-based applications before [64]. But this work successfully used framework specific modeling for migration purposes. This is possible only when the frameworks have similar framework concept abstractions and hence a common FSML is possible.

Configuration file migration is discussed and it is largely a manual process, except for the automatic creation of Tiles pages using a script. View migration is also a manual process and can be carried out in an incremental fashion once the controller component and the configuration files are migrated.

Future Work

An FSML was used for migration purposes in this work. This was possible only because the two frameworks involved in the migration are sufficiently related and have similar

framework abstractions that are captured by the common FSML. In framework migration scenarios, where the two frameworks do not provide similar framework abstractions, it is not possible to define a common FSML. In such migration scenarios two different FSMLs can be used for the source and the target framework. Model transformation can be used to transform the source model, reverse engineered from the source application and expressed in the source FSML, into the target model which will guide the forward engineering or the code migration and is expressed in the target FSML.

View component migration can be automated using an FSML or other program transformation tools. TXL is one such programming language specifically designed for source analysis and program transformation [47]. It uses a rule-based approach to parse and rewrite structured programs. It can be used to replace the Struts tags with equivalent JSF user interface components. As the JSF user interface component framework is extendible, different implementations of JSF provide different components. Therefore, the TXL rules would be specific to the JSF implementation under consideration. Stratego [37] is another such tool that can assist in automated migration of the view component.

In this work, we did not capture the validation methods in the Form Beans as a framework abstraction in the defined FSML. These validations were moved to the action listeners in the JSF Managed Beans. The FSML can be extended to capture this framework abstraction as well.

Appendix A

Glossary of Terms

HTTP - HyperText Transfer Protocol

XML - eXtensible Markup Language

HTML - HyperText Markup Language

WML - Wireless Markup Language

JSP - JavaServer Pages

JSF - JavaServer Faces

J2EE - Java 2 Enterprise Edition

Java EE - Java Enterprise Edition

J2SE - Java 2 Second Edition

JVM - Java Virtual Machine

JRE - Java Runtime Environment

J2ME - Java 2 Micro Edition

EIS - Enterprise Information System

EJB - Enterprise Java Bean

JTA - Java Transaction API

IDE - Integrated Development Environment

RAD - Rapid Application Development

AJAX - Asynchronous JavaScript and XML

DHTML - Dynamic HyperText Transfer Protocol

ASP - Active Server Pages

API - Application Programming Interface

POJO - Plain Old Java Objects

JCP - Java Community Process

JSR - Java Specification Request

OOP - Object Oriented Programming

MVC - Model View Controller

DTD - Document Type Definition

JSTL - Java Standard Tag Library

SFIL - Struts Faces Integration Library

FSML - Framework Specific Modeling Language

Bibliography

[1] Apache Axis - Apache Software Foundation.
http://ws.apache.org/axis/.

[2] Applets - Sun Microsystems, Inc.
http://java.sun.com/applets.

[3] ASP.Net - The Official Microsoft ASP.Net 2.0 Site.

[4] Borland Software Corporation.
http://www.borland.com.

[5] Eclipse - An open development platform.
http://www.eclipse.org.

[6] Eclipse Modeling Framework - Eclipse Tools.
http://www.eclipse.org/emf/.

[7] Eclipse Web Tools Platform.

[8] Enterprise JavaBeans Technology, Sun Microsystems Inc.
http://java.sun.com/products/ejb.

[9] Hibernate Object Relation Mapping.
`http://www.hibernate.org/`.

[10] IBM Corporation.
`http://www.ibm.com`.

[11] IntelliJ IDEA.

[12] Inversion of Control and The Dependency Injection pattern.
`http://www.martinfowler.com/articles/injection.html`.

[13] The J2EE™1.4 Tutorial.
`http://java.sun.com/j2ee/1.4/docs/tutorial/doc/`.

[14] Java 2 Platform, Enterprise Edition - 1.4.
`http://java.sun.com/j2ee/1.4/index.jsp`.

[15] Java Platform, Enterprise Edition - 5.0.
`http://java.sun.com/javaee`.

[16] Java Platform Micro Edition.
`http://java.sun.com/javame/index.jsp`.

[17] Java Servlet Technology.
`http://java.sun.com/products/servlet/`.

[18] Java Standard Edition Overview - Sun Microsystems Inc.
`http://java.sun.com/javase`.

[19] Java Studio Creator.

[20] Java Transactions API, Sun Microsystems Inc.
`http://java.sun.com/products/jta`.

[21] JavaBeans - Sun Microsystems, Inc.
`http://java.sun.com/products/javabeans`.

[22] JavaServer Faces - Java Community Process (Java Specification Request).
`http://jcp.org/en/jsr/detail?id=127`.

[23] JavaServer Pages 2.0 Expression Language.

[24] JavaServer Pages Standard Tag Library.

[25] JavaServer Pages Technology.
`http://java.sun.com/products/jsp/`.

[26] JVM - Java Virtual Machine.
`http://www.java.com/getjava/`.

[27] MacApp: User Interface Framework for Mac OS X.
`http://developer.apple.com/tools/macapp/`.

[28] Microsoft .Net Platform.
`http://www.microsoft.com/net/default.mspx`.

[29] Model-View-Controller, Java Blueprints - J2EE Patterns.
`http://java.sun.com/blueprints/patterns/MVC-detailed.html`.

[30] MyEclipse.

[31] MyFaces - The Apache Software Foundation.
`http://myfaces.apache.org`.

[32] NetBeans IDE.

[33] Oracle ADF Faces - Oracle Corporation.
http://www.oracle.com/technology/products/jdev/htdocs/partners/addins/-exchange/jsf/index.html.

[34] Oracle Corporation.
`http://www.oracle.com`.

[35] Oracle JDeveloper.

[36] Spring Framework.
`http://www.springframework.org/`.

[37] Stratego Program Transformation Language.
`http://www.program-transformation.org/Stratego/WebHome`.

[38] Struts-Faces Integration Library, Apache Software Foundation.
`http://struts.apache.org/1.x/struts-faces/`.

[39] Struts Project Homepage - The Apache Software Foundation.
`http://struts.apache.org`.

[40] Strutsbox Eclipse Plugin for Struts Development.
`http://www.box.de/`.

[41] Sun Microsystems, Inc.: JavaServer Faces.
`http://jcp.org/en/jsr/detail?id=127`.

[42] Taxwide Inc.
`http://www.taxwide.com`.

[43] The Apache Software Foundation.
`http://www.apache.org`.

[44] The Java Community Process Program.
`http://jcp.org/en/home/index`.

[45] The JavaServer Faces Technology - Sun Microsystems, Inc.
`http://java.sun.com/javaee/javaserverfaces/`.

[46] Tiles Library Documentation.
`http://www2.lifl.fr/~dumoulin/tiles/`.

[47] The TXL Programming Language (Turing eXtender Language).
`http://www.txl.ca/nabouttxl.html`.

[48] WebGalileo Faces - Java Web Components.
`http://www.javawebcomponents.com/index.html`.

[49] WebWork - Web Application Framework.
`http://www.opensymphony.com/webwork/`.

[50] WebWork - Web Application Framework.
`http://tapestry.apache.org/`.

[51] World Wide Web Consortium.

[52] IEEE Standard for Software Maintenanece. IEEE Std 1219-1998, October 1998.

[53] Deepak Alur, John Crupi, and Dan Malks. *Core J2EE Patterns*. Prentise Hall PTR, 2003.

[54] Jun Chen and Steve MacDonald. Roadmapassembler: a new pattern-based j2ee development tool. In *CASCON*, pages 55–69, 2005.

[55] Jack S. Chi. Virtual frameworks for source migration. Master's thesis, School of Computer Science, University of Waterloo, 2004.

[56] Mohamed Fayad and Douglas S. Schmidt. Object-Oriented Application Frameworks. In *Communications of the ACM. Special Issue on Object-Oriented Application Frameworks*, volume 40. ACM, 1997.

[57] Erich Gamma, Richard Helm, Ralph Johnson, and John M. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[58] Raja Harinath, Jaideep Srivastava, Jim Richardson, and Mark Foresti. Experiences with an Object Oriented Framework for Distributed Control Applications. In *ACM Computing Surveys*, volume 32. ACM, March 2000.

[59] Richard Hightower. JSF for Non-believers: JSF Component Development. `http://www-128.ibm.com/developerworks/java/library/j-jsf4`.

[60] Ralph E. Johnson and Brian Foote. Designing reusable classes. In *Journal of Object-Oriented Programming 1(2)*, pages 22–35, June-July 1988.

[61] K. Kontogiannis, J. Martin, K. Wong, R. Gregory, H. Muller, and J. Mylopoulos. Code migration through transformations: An experience report. In *Proceedings of CASCON*, pages 1–13, Toronto, ON, 1998.

[62] Yves Lafon. Web Services - World Wide Web Consortium. `http://www.w3.org/2002/ws`, 2006.

[63] B. Lientz and E. Swanson. *Software Maintenance Management*. Addison-Wesley, 1980.

[64] K. Czarnecki M. Antkiewicz. Framework-Specific Modeling Languages with Round-Trip Engineering. In O. Nierstrasz, J. Whittle, D. Harel, and G. Reggio, editors, *MoDELS - Model Driven Engineering Languages and Systems*, pages 692–706, Genoa, Italy, October 2006.

[65] Andrew J. Malton. The software migration barbell. In *First ASERC Workshop on Software Architecture*, August 2001.

[66] H.A. Muller and J. Martin. C to Java Migration Experiences. In *Proceeding of the Sixth European Conference on Software Maintenance and Reengineering*, pages 143–153, March 2002.

[67] David Lorge Parnas. Software Aging. In *Proceedings of the 16th International Conference on Software Engineering ICSE '94*, pages 279–287, May 1994.

[68] Savitha Srinivasan and John Vergo. Object Oriented Reuse: Experience in Developing a Framework for Speech Recognition Applications. In *Proceedings of the 20th International Conference on Software Engineering*, pages 322–330, Kyoto, Japan, 1998.

[69] A. A. Terekhov and C. Verhoef. The Realities of Language Conversions. In *IEEE Software*, volume 17, pages 111–124, November 2000.

[70] Andre Weinand, Erich Gamma, and Rudolf Marty. ET++-an Object Oriented Application Framework in C++. In *In Proceedings of the Conference on Object-oriented Programming Systems, Languages and Applications*, pages 46–57, San Diego, CA, USA, 1988.