Bachelor Thesis

Thanh Minh Tu Tran

A Web-based Front-end in J2EE Environment for a Programming Logic E-Learning Application

Fakultät Technik und Informatik
Department Informations- und Elektrotechnik

Faculty of Engineering and Computer Science
Department of Information and Electrical Engineering

Thanh Minh Tu Tran

A Web-based Front-end in J2EE Environment for a Programming Logic E-Learning Application

Bachelor Thesis based on the examination and study regulations for
the Bachelor of Engineering degree programme
Information Engineering
at the Department of Information and Electrical Engineering
of the Faculty of Engineering and Computer Science
of the University of Applied Sciences Hamburg

Supervising examiner : Prof. Dr. Thomas C. Schmidt
Second examiner : Prof. Dr.rer.nat Hans-Jürgen Hotop
Day of delivery 30th May 2007

Thanh Minh Tu Tran

Title of the Bachelor Thesis

A Web-based Front-end in J2EE Environment for a Programming Logic E-Learning Application

Keywords

E-learning, Client, Server, Web application, J2EE, MVC, Design Patterns, Prolog

Abstract

E-learning has recently gained much consideration in university research projects. Beside the traditional study method that students go to the lectures and listen to professors or lecturers, E-learning offers a very flexible time and location independent way of learning for the participants. Following the idea "learning by doing", a very common learning method is to solve exercises and receive instructions during the exercise solving process.

This report will show us how such an E-learning Application can be built up from the system design to the system realization using the J2EE technology for the front-end and the Prolog language for the back-end.

Thanh Minh Tu Tran

Thema der Bachelorarbeit

Ein Web Front-End für eine J2EE-basierende Prolog E-Learning Anwendung

Stichworte

E-learning, Client, Server, Web Anwendung, J2EE, MVC, Entwurfsmuster, Prolog

Kurzzusammenfassung

E-Learning ist in den letzten Jahren zum Gegenstand von intensiven universitären Forschungsprojekten geworden. In Ergänzung zu dem traditionellen, vorlesungsorientierten Lehrmethoden bietet E-Learning einen sehr flexiblen, orts- und zeittransparenten Weg des Lehrens und Lernens. Der Idee des Lernens durch interaktive Eigenerarbeitung der Inhalte folgend, liegt eine verbreitete Lernmethode in der aufgabenbasierten, eigenständigen Problemlösung.

Diese Arbeit stellt eine exemplarische Lösung für diesen Methodenansatz vor, indem sie ein System konzipiert und realisiert, welches ein interaktives Prolog Back-End über J2EE Technologien in einem Web-basierten Front-End für den allgemeinen E-Learning Einsatz zugänglich macht.

Table of Contents

Chapter 1 – Introduction

Nowadays, E-learning is not a new word to the internet community, especially to students. There are different subjects with different type of E-learning's methods available through the internet, computer-network, and software for PC or television. For a real-time interactive E-learning system using the internet web-based application, one of the most commonly used technologies which can be realized is the client server architecture.

For a normal client server web application, the client sends a request for a resource located on the server, and the server application responds the required resource to the client. But in this E-learning client server application, there is a high interactivity between the client and the server application. The user does not only send a normal request, but he sends the request together with his proposal solution for an exercise. And there is not only one standard solution but there are different proposal solutions of different user for one exercise. The server application has to diagnose all these different solutions and send an appropriate response for each different proposal solution. And from the responses, each user will have to decide what they want to do next differently. This means that the client and the server application have an understanding between them, they can "communicate", "talk" and "understand" each other like human being in which the client is like a student and the server application is like a tutor. One can say that the server application is an artificial intelligent application. This is what high interactivity means for and it makes the E-learning client sever application be different from other normal request response client server application.

Going back to the INCOM E-learning Application and following the idea "learning by doing", a user is requested to solve an exercise by submitting his solution. He will receive step by step instructions whenever he or she has given any invalid solution. There is not only one solution for an exercise, but it can also have many solutions, and that the system needs to allow different inputs and diagnose these inputs. The system interacts with users by receiving the input values, and then making query to the backend, the backend will then return instructions to the front-end. The user will then base on these instructions and continue to give better solutions until he or she can finish the exercise successfully.

This report is written about the author's work which was apart of the INCOM project in the Natural Languages Systems, Department of Informatics, University of Hamburg.

1.1) The Problem

The INCOM [1] E-learning system has two parts the front-end and the back-end. The back-end is programmed under the Prolog [2] language. It contains a logical database [3] of exercises and their constraint-based logics for error diagnosis of each exercise which the students will need to solve. The front-end was developed using Prolog and HTML[4]. That means the entire programming language for this system was Prolog, and the Web Server was SWI-Prolog [5]. The system was released for testing before running into use. But problems occurred when using SWI-Prolog to develop a web-based system:

- Prolog is a declarative language, not a procedural language. This results in complicated codes to implement procedural algorithm.
- The web-server provided by SWI-Prolog is not stable, not matured for production systems in compare to Apache Tomcat server.
- For GUI[6], JSP is better and clearer than a mix of Prolog and HTML code.

From the above problems, the need was realized to develop the front-end in a stable way for multi concurrent accesses and usability. The J2EE platform using Apache Tomcat Servlet technology together with the Java API JPL package[7] was used to program the new front-end.

[1] INCOM is the name of the E-learning Application Project.
It stands for Inputkorrektur durch Constraints und Markups
[2] Prolog is the name of the Programming Logic language
[3] logical database is different from the well known relational database
[4] HTML stands for Hyper Text Transfer Protocol
[5] SWI-Prolog URL – http://www.swi-prolog.org
[6] GUI stands for Graphical User Interface
[7] The Java API JPL package is an open source and released by the SWI-Prolog. The INCOM Application only uses the "using Prolog from Java JPL" package

1.2) The Prolog E-learning Application Overview

"The INCOM system aims at supporting novice programmers in the Prolog Programming language in solving simple standard programming tasks (exercises) by providing help in the event of an error made by the user. " (Nguyen-Thinh Le, 2004, P.1).

The application interacts with the users through a website interface. Conceptually, the application is divided into these functional units:

- Front-end: The website is the interface of the application to the students. The students learn Prolog by solving different Prolog Exercises on the website. While solving an exercise, the student is asked to give input for his or her solution, and then he or she is asked to evaluate the provided solution. If the solution is correct, he or she will move to the next step and continue to provide and evaluate solution until he or she finish this exercise successfully. If the solution is not correct, the system will then response some helpful instruction to show where the errors are. The students will use these instructions to improve his solution and evaluate it again.

- Back-end: This back-end consists of a database of many constraints for different exercises in order to diagnose any different student's solutions and return different helpful instructions for different kind of errors. When the student submits (evaluates) his or her solution, these values are checked and structured in the front-end and then are sent to the back-end. The back-end diagnoses these values and return instructions to the front-end to the student.

1.3) Objective & the scope of this report

The objective of this work was to implement the front-end for the INCOM system under the J2EE platform. The front-end is the web-based interface application. It would have to handle all input from the users, refine the input, send queries to the back-end, handle the response from the back-end, refine the response and then send to the web interface to the users. This was the main task of the author of this report, and this report focuses mainly on the J2EE front-end. The Prolog back-end technology is beyond the scope of this report, but general information about the back-end will be provided for the general understanding of the functioning of this INCOM system.

1.4) The structure of this report

This report is divided into seven chapters. After this introduction chapter, chapter 2 follows.

Chapter 2 concerns about the technologies needed to build up this application. An introduction to the use of the Java API JPL package which is used to interact between Java and SWI Prolog server is introduced in this chapter. Since the Model-View-Controller (MVC) Architecture is the referred architecture for building web application under J2EE platform, in this chapter 2 we will have a look at the MVC architecture and the patterns which can be used inside each layer of the MVC.

In chapter 3, we will discuss the functional requirements for the INCOM Application. This includes the user requirements which are mapped to the base functions of the application. Besides, the prospect flexibility issue will also be discussed.

In chapter 4, the system design of the INCOM Application is discussed in detail. The author of this report will also explain the decisions why to choose this implementation but not another implementation, comparing different methods and patterns which are used in the system design.

Chapter 5 will discuss the process to build up the application based on the architecture in the system design in chapter 4. How to set up a working environment? How to realize the design to a running application using different J2EE technologies at hand?

Chapter 6 mentions about the quality assurance process after the designing phase and developing phase have been completed. This includes different testing methods on the INCOM Application to guarantee that this application is stable, reliable and usability.

Chapter 7 is the conclusion of this report. There is a summary of the main technique, the main design architecture.

At the end of the report, you can find an Appendix, a list of all figures used in this report, a list of the abbreviations used in this report, the glossary list, the index and the CD with the source code and documentations of this work.

Chapter 2 – Used Technologies

2.1) J2EE at a glance

"Java Platform, Enterprise Edition (Java EE) is the industry standard for developing portable, robust, scalable and secure server-side Java applications. Building on the solid foundation of the Java Platform, Standard Edition (Java SE), Java EE provides web services, component model, management, and communications APIs that make it the industry standard for implementing enterprise-class service-oriented architecture (SOA) and next-generation web applications.

The SDKs contain the Sun Java System Application Server (Sun's supported distribution of GlassFish) and provide support for Java EE 5 specifications. The new Java Application Platform SDK features additional runtimes such as Open ESB with JBI and BPEL, Portlet Container, Dynamic Faces, jMaki (Ajax), Phobos, jRuby, Rome, Atom, REST, WADL, Blogapps, and Sun Java System Access Manager." (Sun Microsystems Inc., J2EE At a Glance)

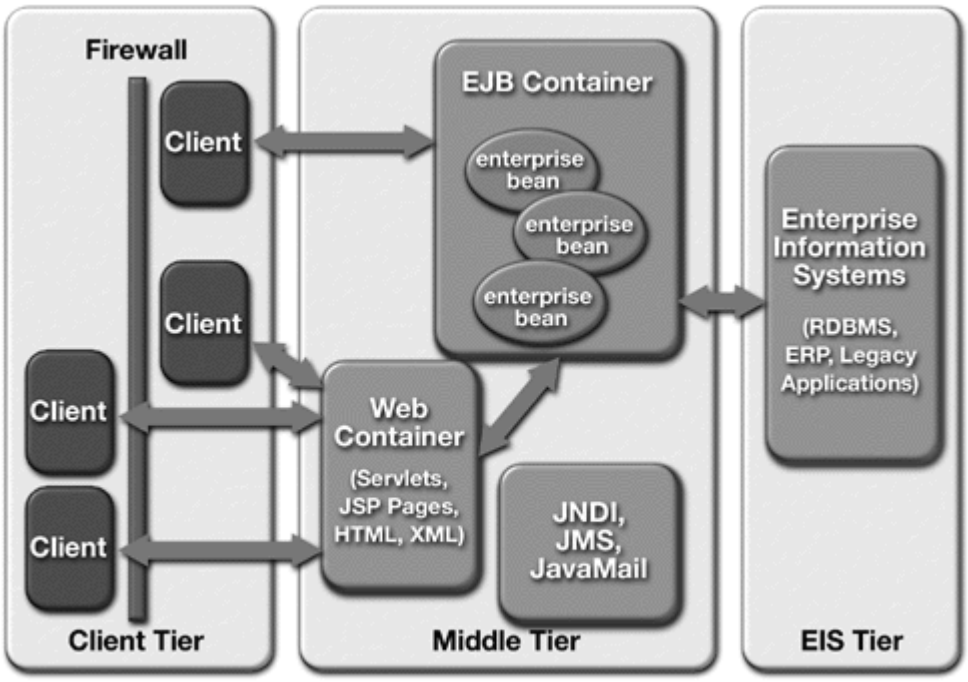


Figure 1: **J2EE Environment** [8]

[8] (Inderjeet Singh, et al., 2002, J2EE platform overview)

2.2) Apache Struts

The INCOM Application use Apache Struts as the Front Controller for all requests to the server. This Front Controller is a design pattern of this application and is mentioned in section 4.4 Design Patterns

Apache Struts is a free open-source framework for creating Java web applications.
Web applications differ from conventional websites in that web applications can create a dynamic response. Many websites deliver only static pages. A web application can interact with databases and business logic engines to customize a response.
Web applications based on JavaServer Pages sometimes commingle database code, page design code, and control flow code. In practice, we find that unless these concerns are separated, larger applications become difficult to maintain.
One way to separate concerns in a software application is to use the Model-View-Controller (MVC) architecture. The Model represents the business or database code, the View represents the page design code, and the Controller represents the navigational code. The Struts framework is designed to help developers create web applications that utilize the MVC architecture.
The framework provides three key components:

- A "request" handler provided by the application developer that is mapped to a standard URI.
- A "response" handler that transfers control to another resource which completes the response.
- A tag library that helps developers to create interactive form-based applications with server pages.

(Apache Software Foundation, Struts)

2.3) Apache Tomcat

The INCOM Application is run on the Apache Tomcat server.

Apache Tomcat is the Servlet container that is used in the official Reference Implementation for the Java Servlet and JavaServer Pages technologies. The Java Servlet and JavaServer Pages specifications are developed by Sun under the Java Community Process.

Apache Tomcat is developed in an open and participatory environment and released under the Apache Software License. Apache Tomcat is intended to be a collaboration of the best-of-breed developers from around the world. We invite you to participate in this open development project.

Apache Tomcat powers numerous large-scale, mission-critical web applications across a diverse range of industries and organizations.

(Apache Software Foundation, Apache Tomcat)

2.4) Embedded Prolog

This INCOM Application Front-end uses the JPL API to communicate with the SWI-Prolog server in the back-end. JPL is the Java Interface to Prolog which is released together with the SWI Prolog server. By using JPL, one can send queries to and receive responds from SWI Prolog server. This JPL API has high and low level interfaces. The INCOM Application uses only the high level interface.

General Description

JPL is a set of Java classes and C functions providing an interface between Java and Prolog. JPL uses the Java Native Interface (JNI) to connect to a Prolog engine through the Prolog Foreign Language Interface (FLI), which is more or less in the process of being standardized in various implementations of Prolog. JPL is not a pure Java implementation of Prolog; it makes extensive use of native implementations of Prolog on supported platforms. The current version of JPL only works with SWI-Prolog.

Currently, JPL only supports the embedding of a Prolog engine within the Java VM. Future versions may support the embedding of a Java VM within Prolog, so that, for example, one could take advantage of the rich class structure of the Java environment from within Prolog.

JPL is designed in two layers, a low-level interface to the Prolog FLI and a high-level Java interface for the Java programmer who is not concerned with the details of the Prolog FLI. The low-level interface is provided for C programmers who may wish to port their C implementations which use the FLI to Java with minimal fuss.

(SWI-Prolog, A Java Interface to Prolog)

Example

To print all of Aristotle's pupils, i.e., all the bindings of **X** which satisfy *teaches(aristotle,X)*, one could write:

```
//declare a regular variable, X is a any pupil in this context

Variable X = new Variable();

//repair a query which will be sent to the SWI Prolog server to
//find all pupils who belong to teacher Aristotle

Query q = new Query( "teaches", new Term[]{new
Atom("aristotle"),X});

//iterate through each pupils in class of teacher Aristotle

while ( q.hasMoreElements() ) {
```

```
  Hashtable binding = (Hashtable) q.nextElement();
  Term t = (Term) binding.get( X);
  System.out.println(t);
}
```

(SWI Prolog, The High-level Interface)

More information and example about the use of Java – Prolog Invocation used in the INCOM Application are introduced in the Appendix at the end of this report.

2.5) MVC [9] architecture for a client server web application

"MVC consists of three kinds of objects. The Model is the application object, the view is its screen presentation, and the Controller defines the way the user interface reacts to user input." (Gamma et al, 1995, P. 4)

"The Model-View-Controller architecture is a widely-used architectural approach for interactive applications" (Inderjeet Singh et al, 2002). The benefit of this architecture is that the MVC divides the functionalities of the application into different object components. These functionalities are clear defined in most application. To see how the MVC architecture can be applied in a multitier Web-based enterprise application, let us see the functionalities of such an application and what the MVC can offer.

- "A model represents business data and business logic or operations that govern access and modification of this business data" (Inderjeet Singh et al, 2002). In a Web-based application, there are business logic and business objects used in an application. And the model layer of the MVC fits this requirement well.
- A view is used as an interface to the users. Since the model will change its content during the process of the application, "the view will renders the content of the model" (Inderjeet Singh et al, 2002). It is true for a Web-based application. The web browser displays the content to the user that means it acts as a view to the user.
- A controller is responsible for all control logic of an application. All users' actions go through the controller and it will decide what to do to react upon the user's actions. In a Web-based application, the user will send a request to the server. This can be seen as an action. A request goes through the controller. The controller will decide which resource to use to process the request, and which view should be responded for this request. All these logic control of the application are done in this controller layer.

[9] MVC stands for Model View Controller.

The advantages of MVC and why to use it?

When the application is divided into model, view and controller layers, it "reduces code duplication and makes applications easier to maintain" (Inderjeet Singh et al, 2002). The development process can be divided into different working groups; each group works on each layer without waiting for the result of other groups. The development period can be shortening. Normally, the view is changed more often then the business rules and especially true for Web-based application, separating between view and model helps changing the view without changing the model.

According to our discussion about the MVC architecture, we can say that applying this architecture into a Web-based enterprise application has great value.

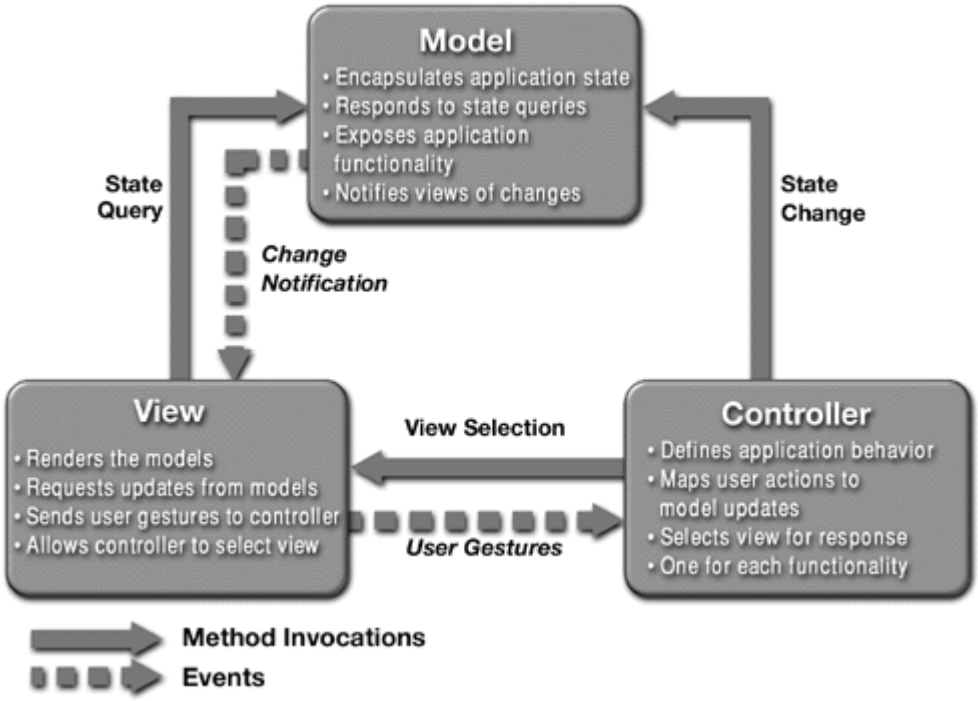


Figure 2: **The Model-View-Controller Architecture** [10]

[10] (Inderjeet Singh, et al., 2002)

2.6) Design Patterns

"A design pattern systematically names, motivates, and explains a general design that addresses a recurring design problem in object-oriented systems. It describes the problem, the solution, when to apply the solution, and its consequences. It also gives implementation hints and examples. The solution is a general arrangement of objects and classes that solve the problem. The solution is customized and implemented to solve the problem in a particular context." (Gamma et al, 1995, P. 360)

Are design patterns important to design an application?

Design patterns are the expert designer's day to day experiences. "Design patterns provide a common vocabulary for designers to use to communicate, document, and explore design alternatives. Design patterns make a system seem less complex by letting you talk about it at a higher level of abstraction than that of a design notation or a programming language" (Gamma et al, 1995, P. 360)

For this reason, together with the MVC architecture, the INCOM Application uses a number of different design patterns which are mentioned in the section 4.4

Chapter 3 – Functional requirements

The INCOM system is an E-learning application for learning Prolog language purpose. It has its interface in the form of a website to interact with users. It has the following requirements:

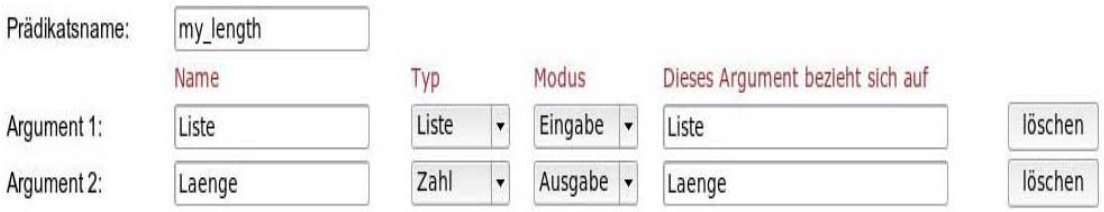
3.1) User Requirements

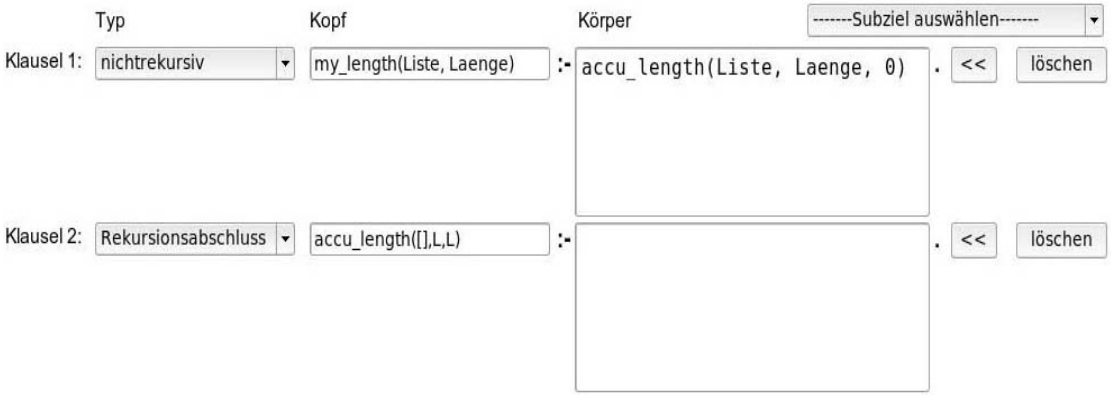
Requirement 1: Application administration
Summary: This requirement describes about the administration actions of the user. The user can register an account, log in, log out, update account, send a message to the administrator, and request a new password.
Condition: no
Description: Student Bill wants to use this Prolog E-learning application, he needs an account. First, he has to register an account. With this account he can log in and use the application. After using, he can log out. He can change his password or request a new password when he forgets it. He can also send email to the administrator. Bill can ask the application to remember his account so that in the next use, he is logged in automatically. The application will ask any un-authorized user to login before requesting the protected pages. After logging in, this user will be forwarded to his previous requested page automatically so that he does not have to the request to the protected page again.

Requirement 2: Selecting an exercise and solving this exercise
Summary: Bill selects an exercise and solves this exercise.
Condition: Bill has logged in successfully and the session of Bill on the application is still active (not yet expired)
Description: At first, a user will select an exercise. From starting selecting to finishing solving this exercise successfully, one has to go through two or three phases: Predicate Declaration, Predicate Definition, Accu Predicate Declaration (this phase is optional)

Remark: one should remember that in Prolog language, for one exercise, not only one but many different possible solutions can be accepted.

In order to solve an exercise, these are the requirements 2.1 - 2.2 - 2.3 - 2.4

Requirement 2.1: Predicate Declaration (Phase 1)
Summary: Bill declares a Predicate, Bill has to declare a number of Predicate Arguments and evaluate his solution.
Condition: Bill is logged in and has selected an exercise
Description: In this phase, Bill is asked to declare his predicate based on the description of the exercise. For this, he has to give his predicate a "name", and declare a number of "arguments" for this predicate. An argument has a name, a type (number, list, atom or any_type), a mode (input, output or both) and a meaning. Then he will submit his solution of this predicate declaration phase. If his solution is correct, then he will move to Phase 2. If it is not correct, the system will return all the errors one by one on his solution. Each error has location information (where on his incorrect solution this error occurs) and the error explanation. The user will use this hints to change his solution and submit it again until the solution for Phase 1 is correct. Below is the figure which shows the predicate declaration phase. 

Requirement 2.2: Predicate Definition (Phase 2)
Summary: Bill defines the declared Predicate, Bill has to declare a number of Predicate Clauses and evaluate his solution.
Condition: Bill has successfully declared a Predicate for the exercise in phase 1
Description: After successfully declaring a predicate, now Bill has to define it. To define a predicate, one needs to declare a number of "clauses". A clause has a type (recursive or non-recursive), a header and a body. At first when a new empty clause is declared, the default value of its header is the result in the predicate declaration phase (phase 1). There is a list of default sub goals which can be selected for the body of a clause. These default values are normally not correct for the solution, normally the user will need to change these values. After declaring these clauses, Bill will then submit his definition solution. And again, if the solution is correct, he will finish the exercise successfully. Otherwise, errors will return and as mentioned in Phase 1, Bill will need to change his solution and submit it again. In this phase 2, besides submitting the solution action, Bill can also select "Accu Predicate Declaration" action (this action is optional). When Bill selects this action, he will then move to phase 3. And Bill can go back to the Predicate Declaration and Accu Predicate Declaration (if it was declared) pages for updating. Below is the figure which shows the predicate definition phase. 

Requirement 2.3: Accu Predicate Declaration (Phase 3)
Summary: In phase 2, Bill decided to declare an Accu predicate.
Condition: Bill has successfully declared the Predicate
Description: Accu predicate declaration is used to declare a new predicate. Accu predicate is one specific type of helper predicates which are used in the main predicate. This Accu Declaration has the same idea, structure as of phase 1(Predicate Declaration). The user will declare a name, arguments (for this Accu Predicate) and submit his solution. And again, it the solution is correct, he will move back to phase 2 to finish this predicate definition. If it is not correct, errors returned, he will change the solution and submit it again. And Bill can go back to the Predicate Declaration page for updating. Below is the figure which shows the predicate definition phase. 

Requirement 2.4: Updating student's solution in different phases
Summary: Bill has the possibility to update his solution in the Predicate Declaration and Accu Predicate Declaration phases
Condition: Bill can only update the solution of the phases which were correctly declared before.
Description: After declaring a Predicate or Accu Predicate, Bill can go back to either the Predicate Declaration or Accu Predicate Declaration phase to update a new solution for it. Because the result of the Accu Predicate Declaration phase depends on the result of the Predicate Declaration phase, there are two constrains as follows: • If Bill has declared both the Predicate and Accu Predicate and he decides to update the Predicate Declaration, the Accu Predicate will be lost. He has to declare a new Accu Predicate again if he may want. • If Bill has declared only the Predicate Declaration and wants to update it or if Bill wants to update only the Accu Predicate Declaration, he can do it in a normal way.

Requirement 3: Writing to User Log file
Summary: All the interactions between a user and the server are written to a log file of each user
Condition: during the exercise solving process
Description: During the exercise solving process, whenever Bill sends any action and solution to the server as well as any returned values which the server responds to Bill, these actions and values are written to Bill's log file on the server. The purpose of doing this is that the Back-end Administrator wants to see the learning process of students. He also wants to see different possible solutions of students for each exercise that the students may submit. This will help the Back-end Administrator to improve the performance of the Back-end in the future.

Information about the content of a log file is introduced in the Appendix at the end of this report.

3.2) Base functions

According to the user requirements, these requirements are mapped into the base functions of the application. The following diagram may be useful to make the picture clear.

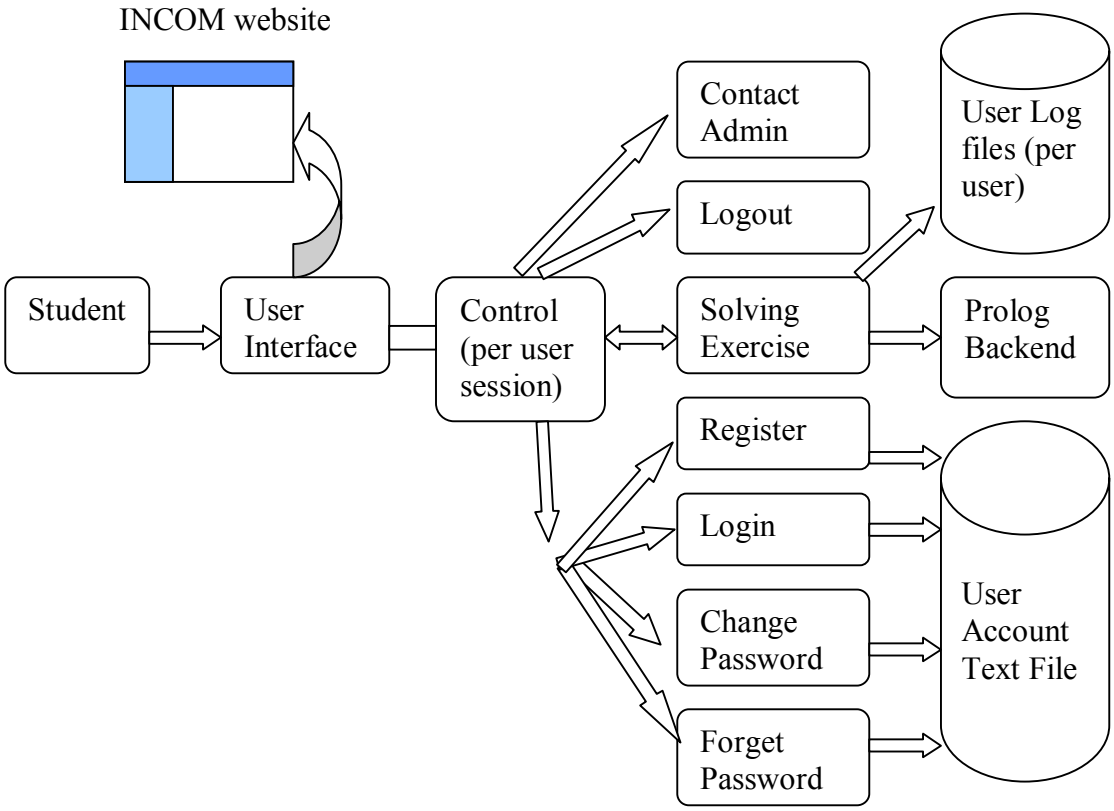


Figure 3: The base function diagram of the INCOM Application

A user interacts with the application through a website interface. All requests from the user will go through the front-end controller. From there, the controller will interpret for which resource the user has requested.

As we can see, there are a number of actions which the user can request to the system.

1) Send a message to the administrator using a contact form on the web interface
2) Register an account to use the application
3) Log in and log out the application
4) Change password or request new password
5) Select an exercise
6) Solve the selected exercise

In last action on the 6th line, during the exercise solving process, there are sub actions:

- Declare a predicate
    - Add a new argument
    - Remove an argument
    - Submit the declaration solution
    - Show error explanation
    - Move to next error
    - Move to previous error
- Declare an Accu predicate
    - Having the same actions as when declare a predicate
- Define the declared predicate
    - Add a new clause
    - Remove a clause
    - Submit the definition solution
    - Show error explanation
    - Move to next error
    - Move to previous error

In the diagram, the back-end is the SWI-Prolog server which serves to receive requests and sends responses to the front-end. The sending and receiving methods of the front-end to and from the back-end are done using the API JPL package which is available together with the SWI-Prolog server[11]. Since this back-end was not my task to develop it, so that I would like to leave the introduction to the back-end here.

The others two text file systems (User log files and user account text file) are implemented using normal text file.

The user account text file stores all the user account information [12]

For each user, there is a user log file. All the user actions, request sending information and received response information from the back-end are written into his log file. This serves for statistic purpose and for improving the system behavior later on.

[11] The interaction process between the front-end and back-end using API JPL package is shown in the Appendix at the end of this report

[12] For this INCOM Application, an account has only a username and a password

3.3) Prospects / Flexibility

The main objective of the Prospects/Flexibility issue for this Application is the question how the INCOM Application will be easily extendable in the future. To answer this question, several techniques in the design of the INCOM Application are in consideration.

Firstly, the design of the application has to be clearly separated between different layers of the MVC architecture. This helps different developers to work on different layers concurrently. And it also helps to update the view layer easily without changing the other layer's implementations.

Secondly, there are a number of design patterns which are used this application. When they are applied in a correct way into this INCOM Application, it ensures the application to be flexible in the future for updating or reuse of the components. Design patterns of this application are discussed in section 4.4

In addition, in the view layer, the page author can use the Cascading Style Sheet technique to help the design of the page layout uniformly.

Figure 4 is the use case diagram which shows the potential actors and their actions: "Use case analysis identifies the actors in a system and the operations they may perform." (Inderjeet Singh, et al., 1995)

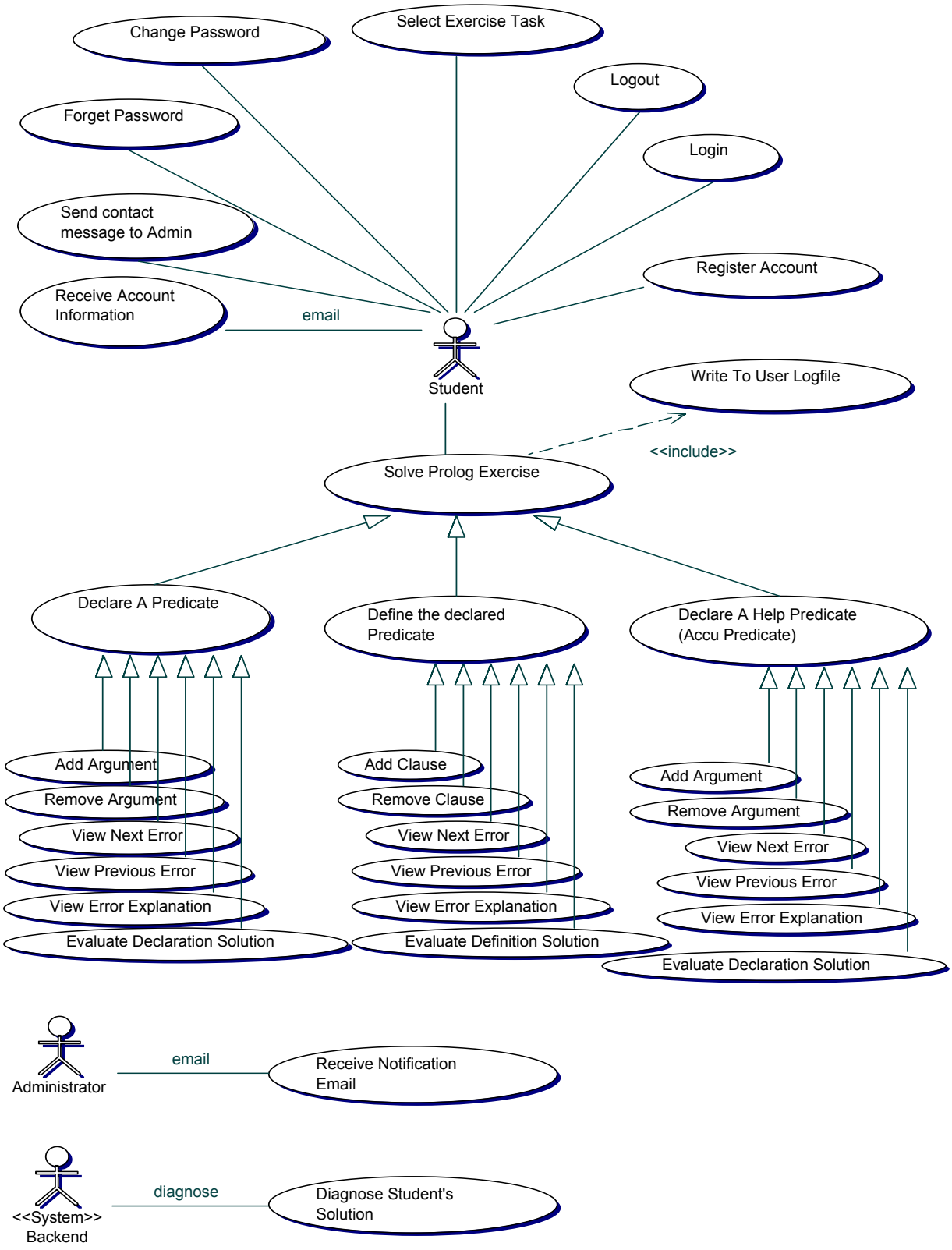


Figure 4: Use case diagram of the INCOM Application

Chapter 4 – System Design

4.1) Choosing application web tier

In the J2EE platform, application may have the following tiers:

- Client tier [13]
- Web tier
- EJB tier
- Information tier [14]

The client tier of this INCOM Application is only a browser. The information tier is the Prolog back-end.

This application front-end uses Web-tier and EJB-tier. In the MVC architecture of this application, Web-tier is used as the view layer and the controller. The EJB-tier is the model layer. Details about the design of these layers will be discussed in the Application Architecture.

The front-end and back-end are located on the same server (locally). Distributed architecture is not in consideration.

After making it clear about the functional requirements of the application, and having an understanding of the Model-View-Controller architecture, the next step is to design the application architecture.

4.2) Application architecture

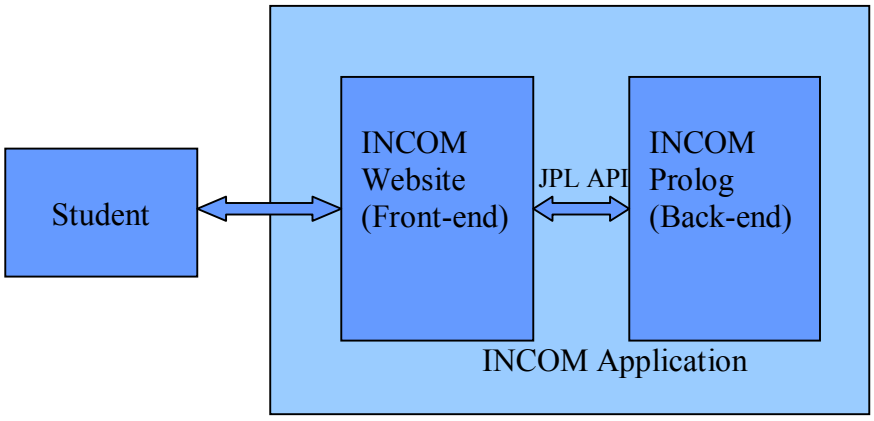


Figure 5: High-level view of the application

[13] Client Tier: software run on client PC, applet, etc.
[14] Information Tier: e.g. database system

First of all, one has to define the objects and the components which the application needs and decide to which layer they belong. There are some criterions which help to design the application objects.

- Objects must be reusable
- A clear distinction between different component functionalities
- Separate between stable code [15] from code that changes frequently [16]
- For large enterprise application project, it is required that the development process is carried out by different groups including a template team, enterprise business team, programmers, etc. in parallel. But this was not the case in my development process, since I managed all the jobs to develop the front-end

It is best practice to follow a good coding convention recommendation during the development process

The template of the website interface is as follow

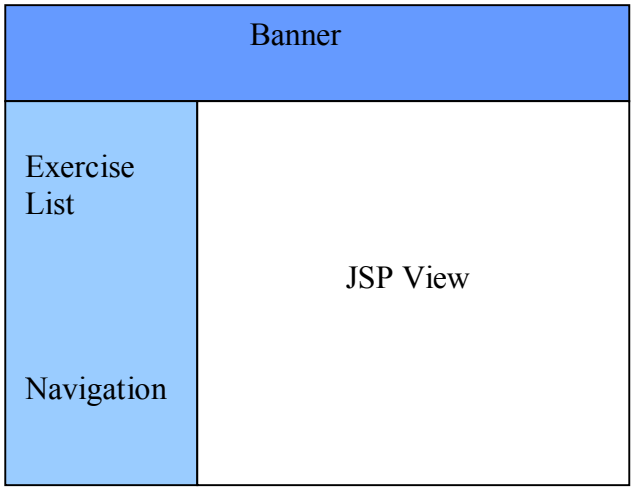


Figure 6: INCOM front-end template

Once the functional requirements are partitioned into actions of requests and responds between users and application which was already mentioned in the section 3.2 Base Functions, the next step is to identify the business logic, presentation logic, control flow logic and model them as objects.

The following sections introduce the general MVC structure design of this application and then we will go further into internal design of each layer of the MVC to decide which objects each layer needs following some of the recommended core J2EE patterns.

[15] Stable code in this context means the business rules of the application
[16] Code that changes frequently is normally the presentation view

4.3) Overview of the MVC Architecture of the INCOM Application

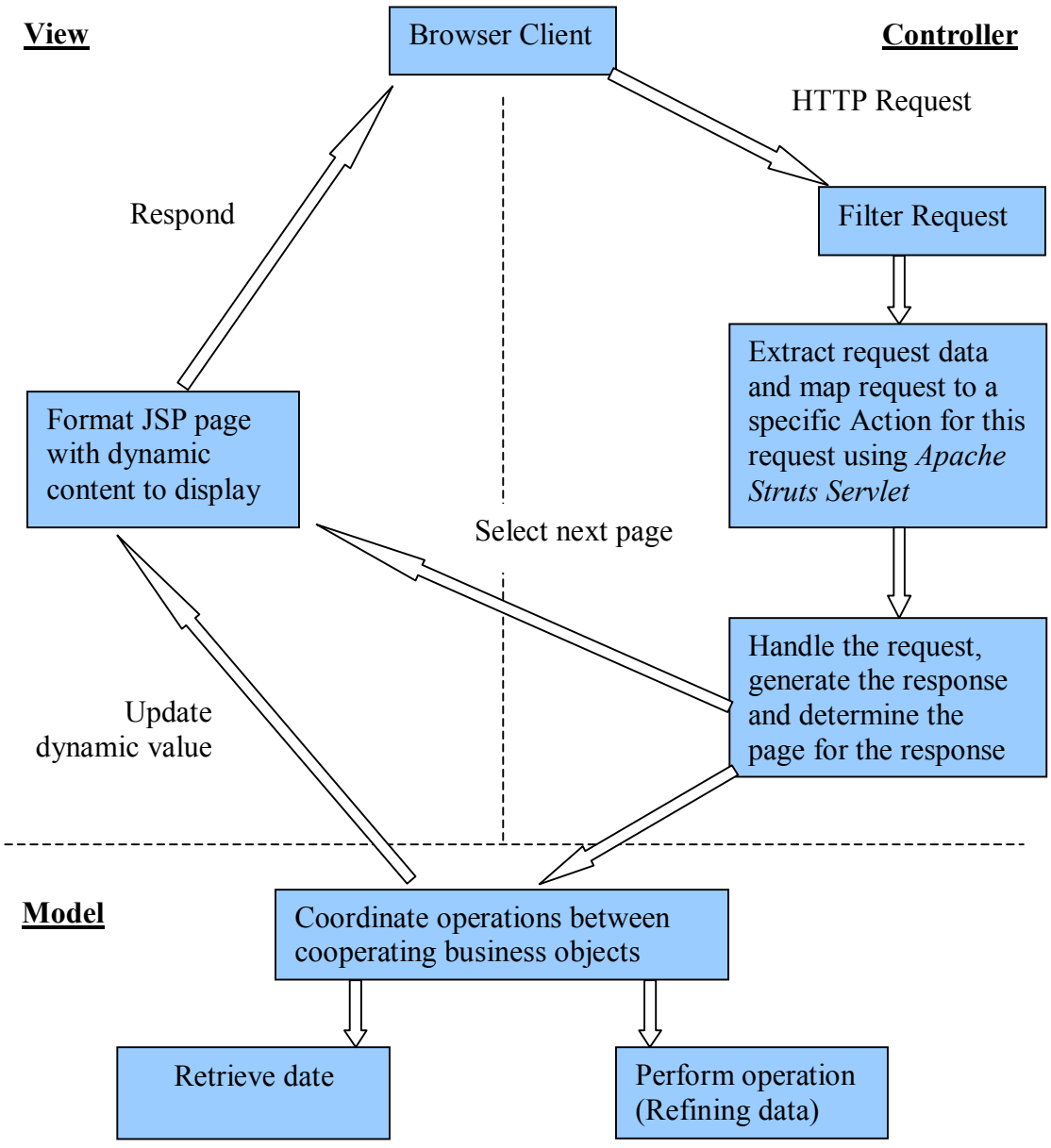


Figure 7: Model-View-Controller Architecture of the INCOM Application

A client sends a HTTP request to the controller. The controller receives the request, handles the request with corresponding control logic for this specific request, and decides the next JSP page to respond to the client.

The model has different objects to store different data type and objects which have business rules to process the data. When receiving request data from a client, data is checked for invalid input value, refined and sent to the Prolog back-end. As well, the response data from the back-end is refined and sent to the client.

The view has different JSP pages, their dynamic values are the values stored in the value objects. And of cause the state and value of these value objects can be updated.

From the functional requirements and the prospect/flexibility requirements which were mentioned in chapter 3, the following design patterns are used in the three layers of the MVC architecture for the INCOM Application.

4.4) Design patterns used in the INCOM Application

- **Intercepting filter:** The INCOM Application uses this filter for the pre-processing. It provides authorization service in the application and is implemented in the Servlet controller. This pattern is used in the Control layer of the INCOM Application
- **Front controller: "**This pattern provides a centralized controller for managing requests. A front controller receives all incoming client requests, forwards each request to an appropriate request handler, and selects a JSP page to respond to the client." (Inderjeet Singh, et al., 1995) The INCOM Application uses the open source Apache Struts as the front controller for all incoming requests. This pattern is used in the Control layer of the INCOM Application
- **Command pattern:** This pattern encapsulates each function of the application into an action class. Each command instance is responsible for each request, and performs appropriate processes for the response of the request. This pattern is used in the Control layer of the INCOM Application
- **View helper:** The INCOM Application uses view helper pattern in term of the custom handler tag and value bean object. "It encapsulates the presentation logic of a view" (Inderjeet Singh, et al., 1995). This helps to keep the view simple so that a page author can work on the view without any programming knowledge. This pattern is used in the view layer.
- **Session façade:** Since the client uses many different methods from different business objects, it will become more complex for the developer to work on different objects. The session façade provides a simpler interface for the client to access the components since all accesses to different components are centralized through the session façade object. This pattern is used in the Model layer of the INCOM Application

- **Value List Handler:** A single Value List Handler object is used to retrieve or process a set of related data. It helps to simplify the access and process of a set of related data objects. This pattern is used in the Model layer of the INCOM Application. The value list handler objects are encapsulated under the session façade object.

4.5) Detail implementation of the INCOM MVC architecture.

4.5.1) The View Layer

The J2EE platform technology for handling user view in this INCOM Application is JSP pages. In general, this application uses JSP for presentation of the dynamic contents and logic for presenting the view.

There are some considerations in the view layer

- View components focus on presentation
- View has only presentation logic
- No embedded scriptlets in JSP pages
- Make use of the view helper pattern
- Friendly user interface for the view

It is decided that the controller layer is responsible for the control logic, and the java bean objects in the model layer manage the business rules. So that the view layer will focus only on the presentation logic. Besides, the custom tag handler classes will be used to encapsulate the complex presentation logic.

In order to have no embedded scriptlets for the presentation logic in the JSP pages, one has to make use of the Standard Tag Library, Express Language, Custom Handler Tag and JavaBeans for retrieving data.

The figure 8 below shows how the view helper pattern is integrated into the design of the view layer.

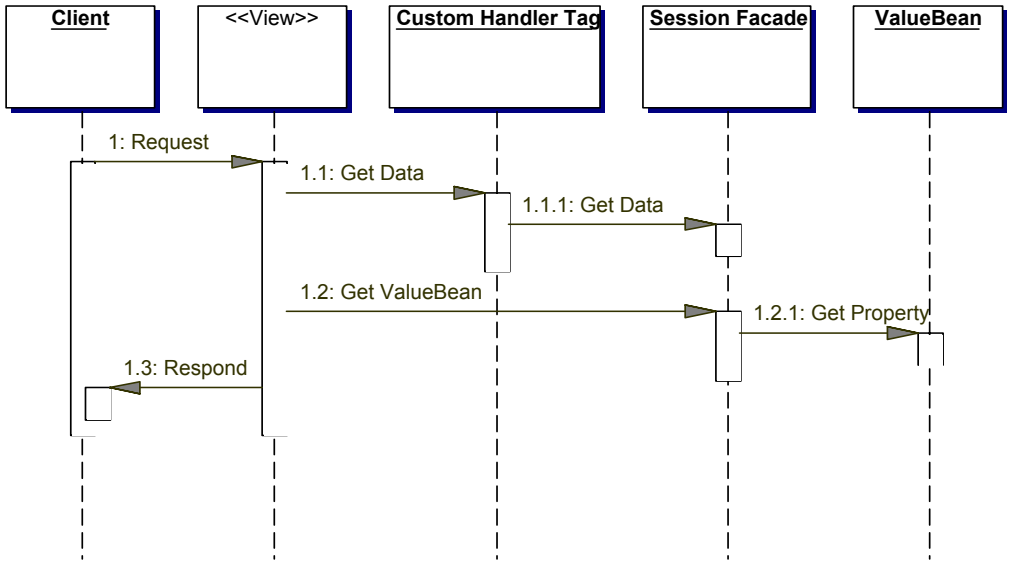


Figure 8: View Helper Pattern Sequence Diagram

The page layout:

"Applications usually strive for a common look and feel, and keeping page layouts similar within an application is important to establishing such a look and feel" (Inderjeet Singh, et al., 2002).

All the pages in the INCOM Application have the same layout style. The page template is a frame. It has a banner at the top, a navigation menu on the left, and the central place to load all the JSP pages. This satisfies the look and feel layout.

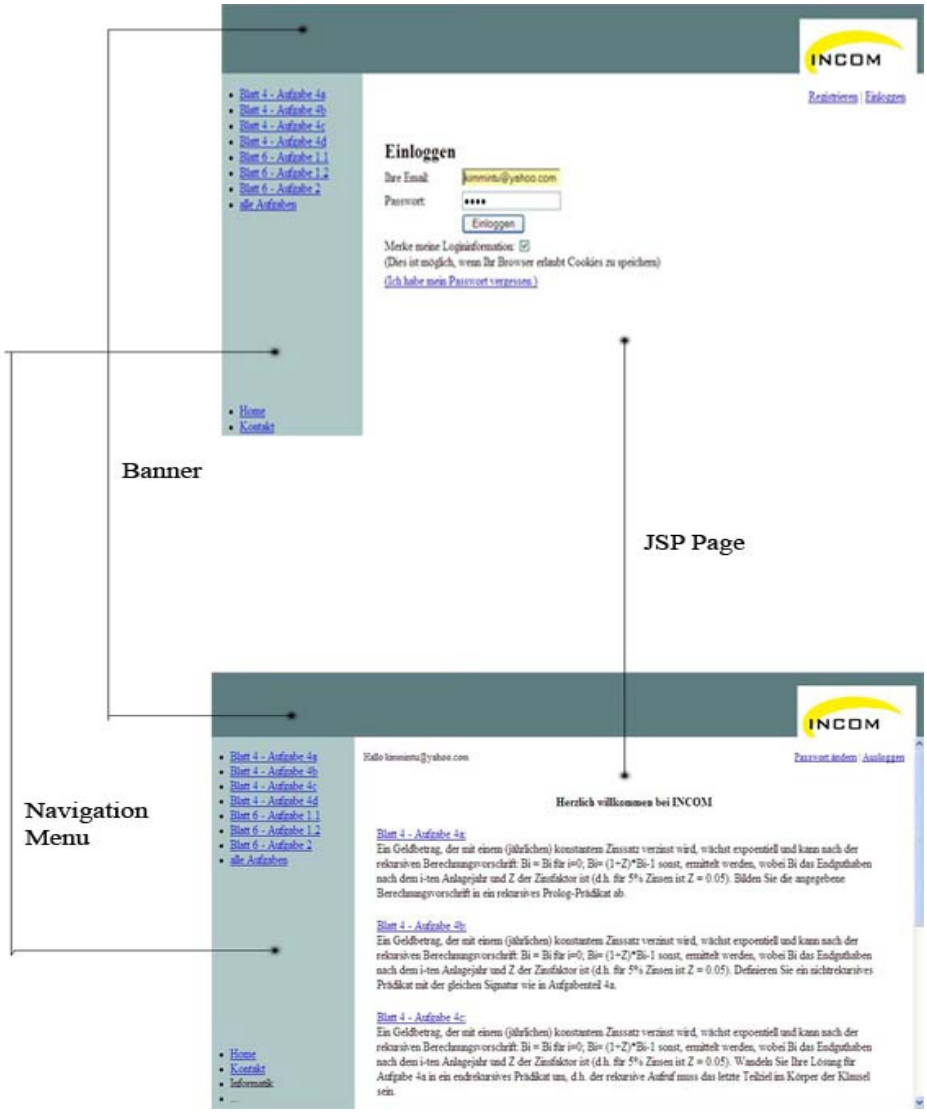


Figure 9: The template of the INCOM Application web-based interface

Conclusion on the design of View layer

With this design, the View layer is totally separated from others layer. This enforces the separation of the developer roles. The Web page author can concentrate on the visual design skill without any programming effort. The layout of the web page can be changed easily. Different developers can work independently. And the development period can be shorten.

4.5.2) The Model Layer

The model layer encapsulates the data objects and business logic of the application. "Enterprise beans and the EJB tier are the recommended J2EE technology for implementing these business objects" (Inderjeet Singh, et al., 2002). And the INCOM Application uses enterprise beans to implement its business logic.

The EJB classes of the INCOM Application were designed with the following consideration:

- They can be reusable
- Separate value beans from utility beans.
- Use a session façade pattern [17] to simplify the view accessing.
- Use value list handler pattern [18] to group objects of the same class to simplify and centralize the access to these objects.

And the model layer is realized as the class diagram in the next figure.

[17] Façade provides a unified interface to a set of interfaces in a subsystem. Façade defines a higher-level interface that makes the subsystem easier to use. – (Gamma et al, 1995, P. 185)

[18] Value List Handler – Use a Value List Handler to control the search, cache the results, and provide the results to the client in a result set whose size and traversal meets the client's requirements. – (Sun Microsystems Inc. 2002, Core J2EE Patterns)

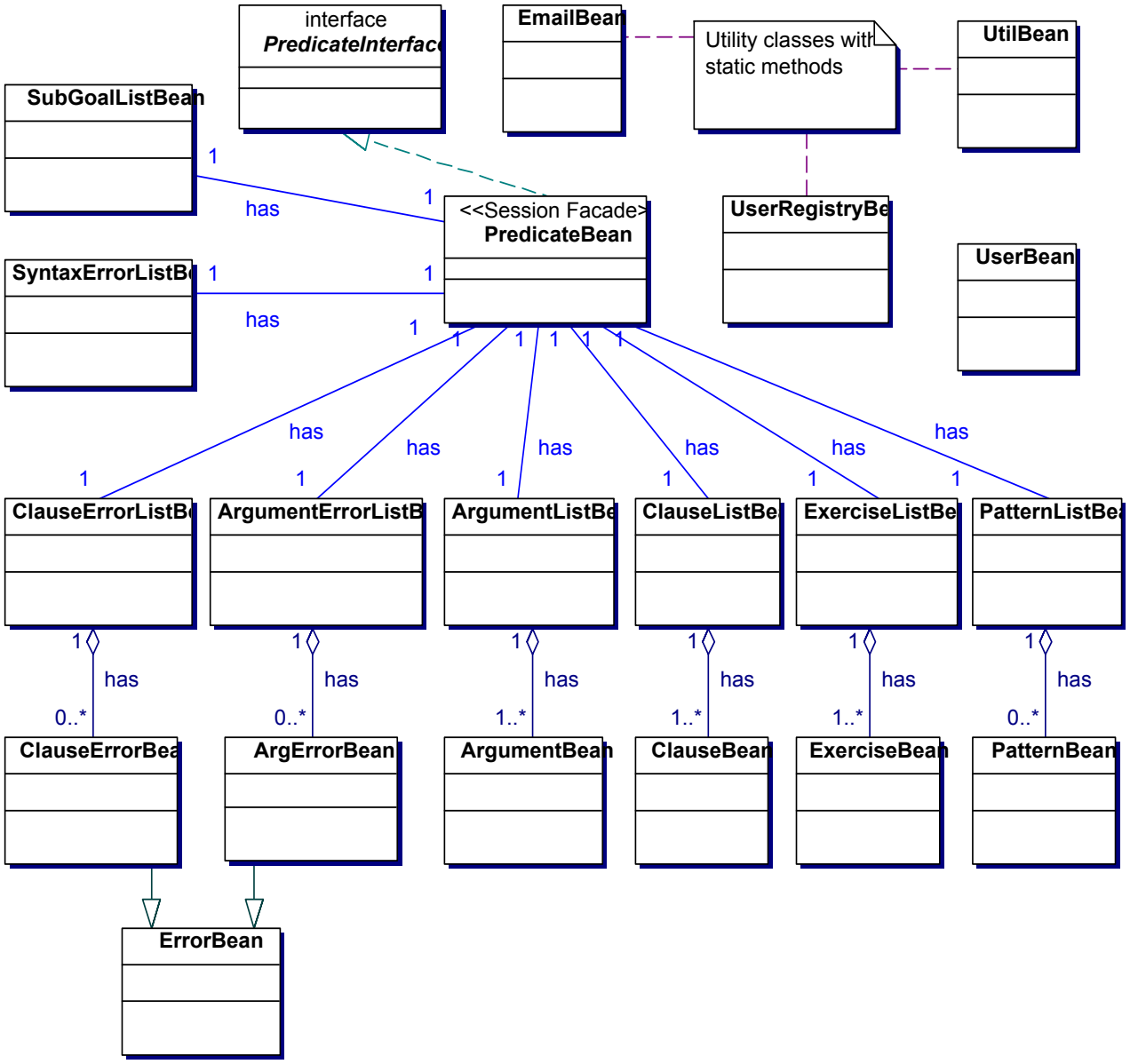


Figure 10: Model layer class diagram of the INCOM Application

Explanation of the design choice for this model layer

- Session Façade PredicateBean**:** Since the INCOM Application has many different business objects and business logics, it results into a difficulty to manage the cooperation between these objects. To have a solution for this problem, the session façade PredicateBean and command pattern [19] are in used. The PredicateBean acts as a single interface to the model layer in other layers. In the PredicateBean, there are business logics which are made up from different cooperating classes. The PredicateBean is also used as a single interface to all business value objects in the application.

- Interface PredicateInterface: all constants are declared in this interface. It is a good practice to declare all constants in one place so that the programmer can refer to and check the constant values easily during the development process.

- Utility classes with static methods UtilBean, EmailBean[20], UserRegistryBean: all the reusable methods are in the UtilBean class. EmailBean and UserRegisterBean classes have methods for sending email and methods to work with user's account registration.

- UserBean: only signed in users are allowed to use the application. A UserBean object will be created for each logged in user. The UserBean has a username, a password attributes and methods for writing log information to this user's log file.

- Pure value business objects ArgumentBean, ClauseBean, ArgErrorBean, ClauseErrorBean, ExerciseBean, PatternBean: these are value bean objects which are used to store the information on the front-end of the INCOM Application. As can be seen, the ErrorBean is the super class of the ArgErrorBean and ClauseErrorBean since these two sub classes have common attributes.

- Utility bean objects SubGoalListBean, SyntaxErrorListBean, ArgumentListBean, ClauseListBean, ArgErrorListBean, ClauseErrorListBean, ExerciseListBean, PatternListBean: These classes realize the Value List Handler pattern. These are objects which help to simplify the increasing value bean objects of each related data set and methods dealing with these set objects. For example, since the user will have to declare a number of arguments in the Predicate Declaration phase, and there are methods for processing these arguments. There are methods for adding a new argument, removing an argument, asking how many arguments are declared etc. The ArgumentListBean will handle all these requirements.

The figure blow shows the accessing strategy using the session façade pattern.

[19] Command Pattern is discussed in the control layer section
[20] The source code of the EmailBean.java class is taken from (Bruce W. Perry, 2004)

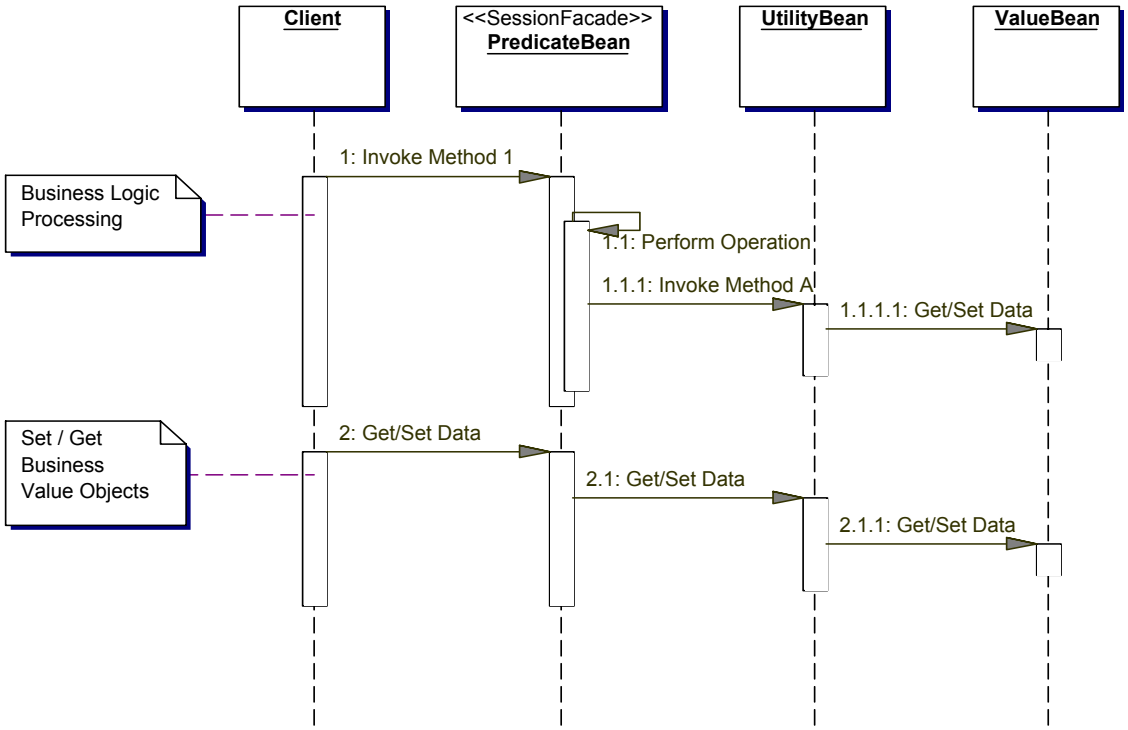


Figure 11: Session Façade sequence diagram

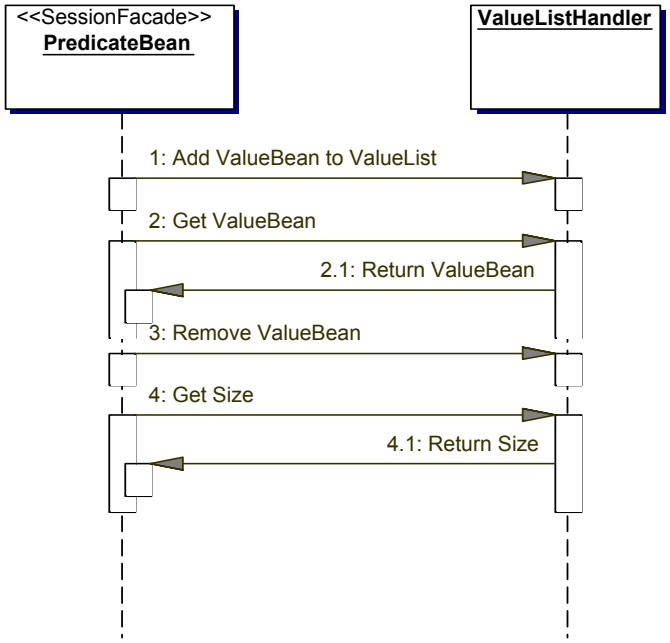


Figure 12: Value List Handler Sequence Diagram

Conclusion on the design of the Model layer

There is a clear distinction between the value beans and the utility bean to promote the reuse ability of different business components.

The data access strategy to the user account and user log files are centralized in the UserBean and UserRegistryBean classes.

The most interesting thing in this model layer is the methods to refine request data, send the request to the back-end, receive the response from the back-end, refine response data and respond to the client's view using the API JPL package. This technique will be more explored in the Appendix at the end of this report.

For a large application, separating different tasks in the model layer can help to divide the development process to different developer roles in this model layer.

4.5.3) The Controller layer

The controller layer of the MVC is used for the control logic of the application. The control logic works with both the view layer and the model layer. For the view layer, the controller is responsible for selecting the next view. For the model layer, the controller will use the business value objects and executes the business logic processes.

The controller layer belongs to the Web-Tier. It acts as a single interface for all incoming HTTP requests from the clients. Basically, a front-controller receives a request; it maps this request to a specific action class to handle this request. The action class will need to execute the business logic processing, to update the business value objects and to decide the next page to return to the client. This front-controller is realized in the Servlet technology. If the controller Servlet is developed, it has to full fill these basic requirements:

- "All the requests have to be passed to the front-controller Servlet
- The Servlet must be able to distinguish requests for different type of processing" (Hans Bergsten, 2003, P. 388)

Besides, other features one may want to support for the prospect flexibility issue

- "A strategy for extending the application to support new types of processing requests in a flexible manner
- A mechanism for changing the page flow of the application without modifying code" (Hans Bergsten, 2003, P. 388)

In the INCOM Application, Apache Struts Servlet is used as the front-controller since its Servlet satisfies all the above requirements. It receives all requests from clients such as register an account, change new password, select an exercise, evaluate a solution etc. And for each request like this, the Apache Struts will map the request to a corresponding action class such as RegisterAccountAction class. This class will receive the value posted from the client; execute a business logic method to validate the correctness of the submitted value. If the validation is passed, it will then create a new account for the client and redirect the user to another page.

The sequence diagram below shows how this front-controller works

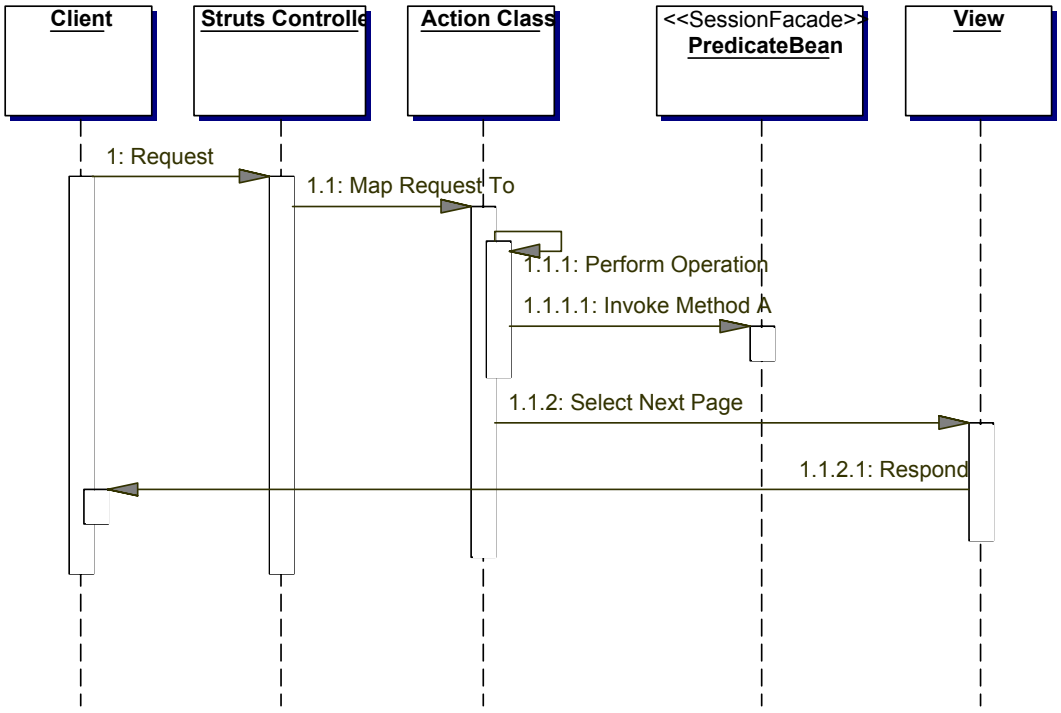


Figure 13: Action class and Apache Struts Controller sequence diagram

Based on the functional requirements of the INCOM Application in chapter 3, the following are the implemented Action classes for different requests.

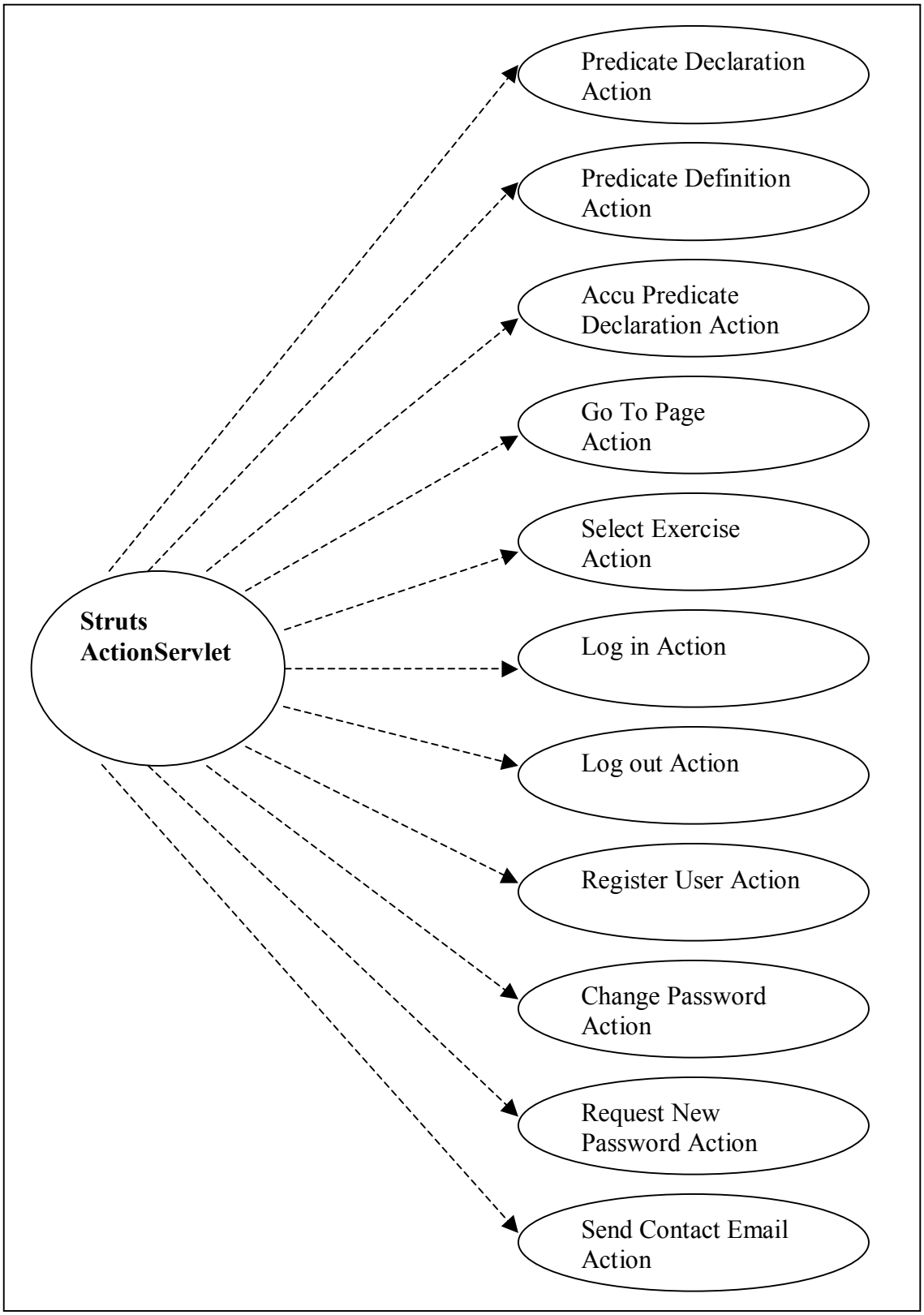


Figure 14: Controller Servlet and action classes

The first three actions are used in the three different phases during the solving exercise process. They are Predicate Declaration, Predicate Definition and Accu Predicate Declaration. Indeed, for each of the three phases, there are a number of actions on it. For example in the Predicate Declaration phase, there are actions:

- Add A New Argument
- Delete An Argument
- Evaluate The Declaration Solution
- View The Error Explanation
- Move To Error
- Move To Previous Error

All these different actions are handled in one DeclarationAction class. The DeclarationAction class will parse the submitted request value and decide which action was made to have appropriate control processing. It is true for the other two action classes DefinitionAction and AccuDeclarationAction.

The names of the action classes in this INCOM Application have already told us what they do. The GoToPageAction class is used when the user want to go back to the previous JSP page during the exercise solving process for solution updating purpose.

The decision of the Action classes to select the next view to respond to the client is made according to the following page flow diagram:

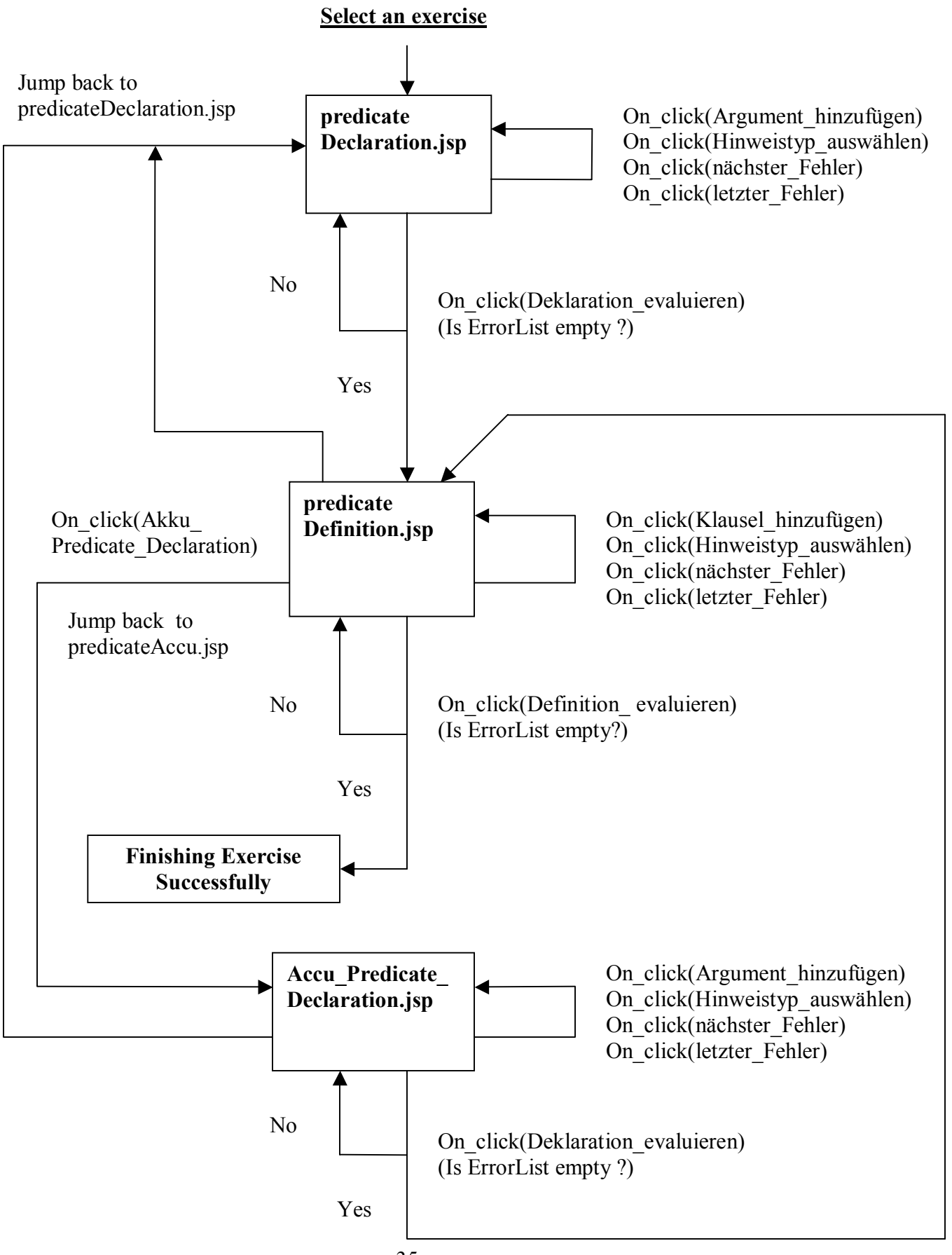

Select an exercise
Jump back to predicateDeclaration.jsp
predicate Declaration.jsp
On_click(Argument_hinzufügen)
On_click(Hinweistyp_auswählen)
On_click(nächster_Fehler)
On_click(letzter_Fehler)
No
On_click(Deklaration_evaluieren)
(Is ErrorList empty ?)
Yes
predicate Definition.jsp
On_click(Akku_ Predicate_Declaration)
On_click(Klausel_hinzufügen)
On_click(Hinweistyp_auswählen)
On_click(nächster_Fehler)
On_click(letzter_Fehler)
Jump back to predicateAccu.jsp
No
On_click(Definition_ evaluieren)
(Is ErrorList empty?)
Yes
Finishing Exercise Successfully
Accu_Predicate_ Declaration.jsp
On_click(Argument_hinzufügen)
On_click(Hinweistyp_auswählen)
On_click(nächster_Fehler)
On_click(letzter_Fehler)
No
On_click(Deklaration_evaluieren)
(Is ErrorList empty ?)
Yes

Figure 15: JSP page flow diagram

"A front controller is a good design approach because it provides a single point of contact for all application requests. It interprets requests, executes business logic, and handles security, error handling, and view selection. Centralizing application control provides a natural point for implementing application-wide services and reduces code redundancy." (Inderjeet Singh, et al., 2002)

As mentioned in the Session Façade Pattern, implementing the Controller with the Command Pattern using Action classes not only simplify the Session Façade PredicateBean but also can make use of the open source Apache Struts Servlet Controller which helps to save a lot of time for developing such a Servlet. "It also keeps the controller implementation cleaner by encapsulating event- and request-handling tasks into smaller objects" (Inderjeet Singh, et al., 2002)

In Servlet 2.3, Filter and Listener are introduced and they are used in the INCOM Application.

Intercepting Filter

"A filter is a component that can intercept a request targeted for a Servlet, JSP page, or static page, as well as the response before it's sent to the client. This makes it easy to centralize tasks that apply to all requests, such as access control, logging, and charging for the content or the services offered by the application." (Hans Bersgten, 2003, P. 374)

The controller layer of the INCOM Application also uses the intercepting filter pattern. All incoming requests have to go through the intercepting filter for user validation in order to use the application and to check if the current session of a user's browser has already been expired.

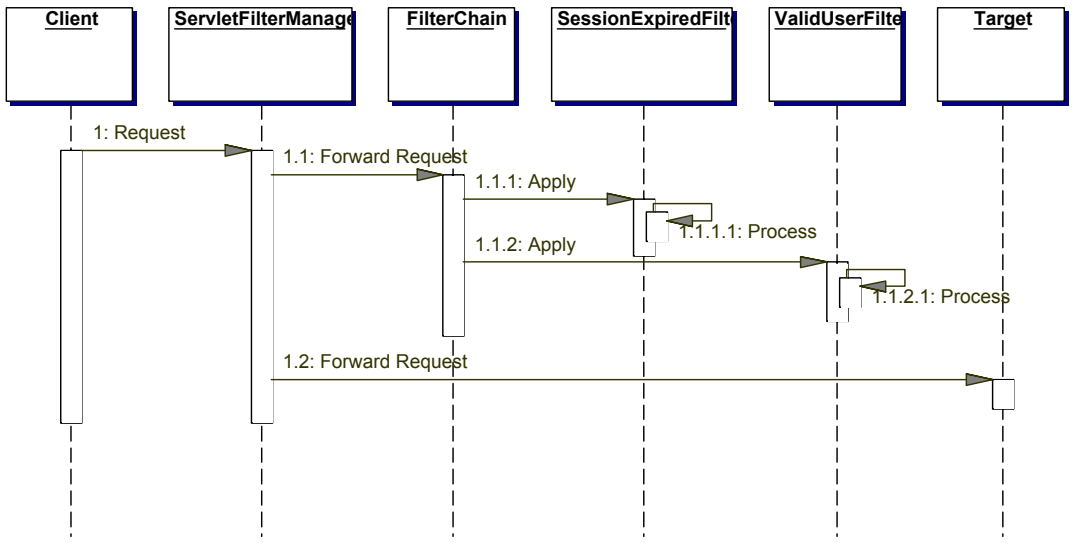


Figure 16: Intercepting Filter Sequence Diagram

Listener

"Listeners allow your application to react to certain events. A session attribute event listener also makes it possible to deal with attribute binding events for all sessions in one place, instead of placing individual listener objects in each session." (Hans Bersgten, 2003, P. 375)

In the INCOM Application, all the application scope attributes are registered in the Listener class when the application is first started. This helps to centralize all the application scope attributes in one place and makes it easy to remove or add new attributes.

4.6) The total picture of the INCOM Application Design

Let us have a look at the general picture of the MVC architecture that makes up the design of the INCOM Application. The MVC architecture is decomposed into components. And each component is realized followed a corresponding design pattern for its functionalities.

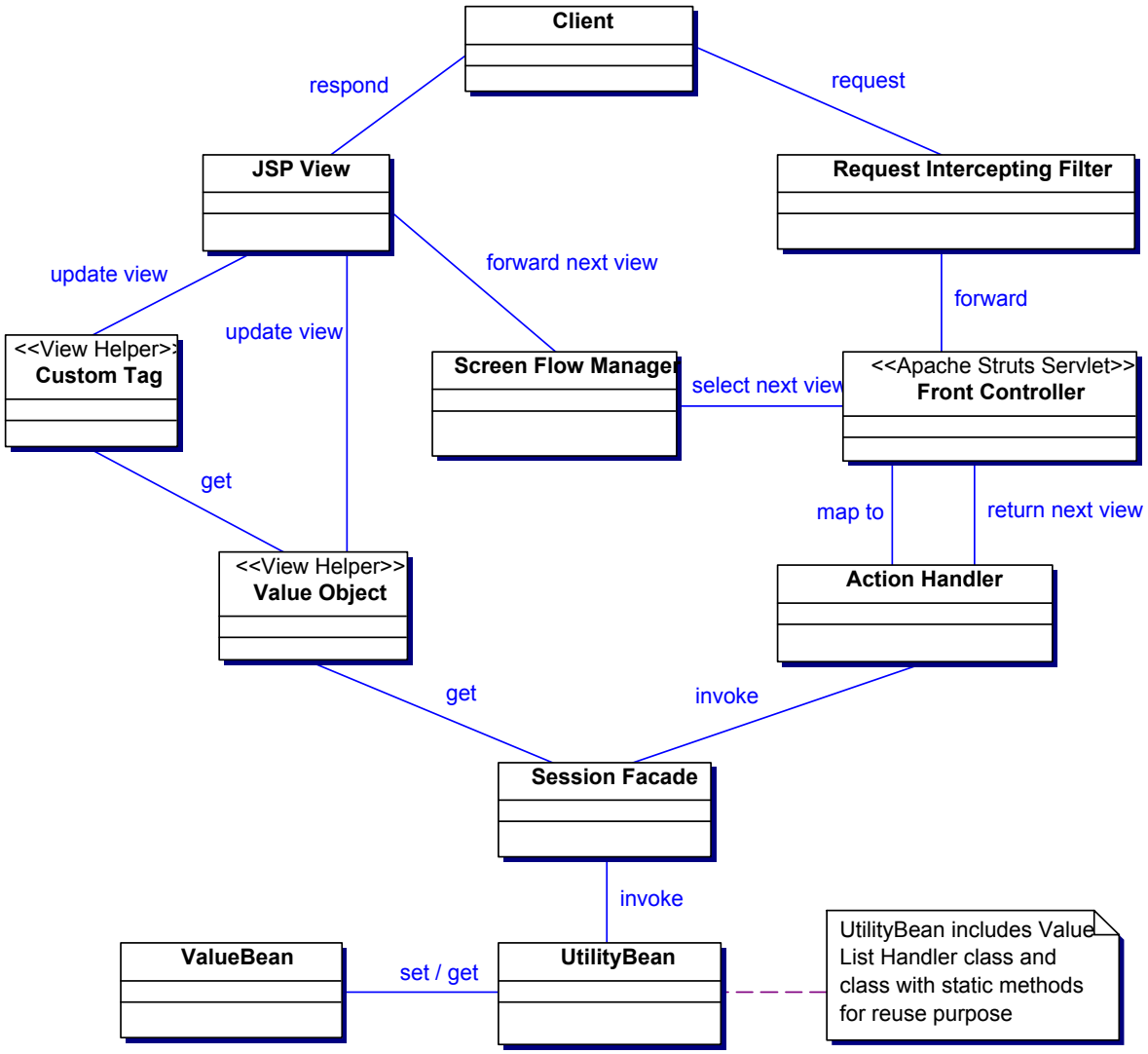


Figure 17: **INCOM Application Architecture Component Class Diagram**

At the beginning, the client interacts with the front-end web page interface of the application and then submits an HTTP request to the front-controller.

Before entering the front-controller, the Intercepting Filter will have to check if the current session of this client is expired or not. Then it checks if this is a valid user. If one of these checks is not passed, the client is forwarded to the login page. If they are passed, the request is forwarded to the front-controller. The front-controller checks what kind of request it is, and maps the request to a corresponding action handler class. The action handler object simplifies the Session Façade by determining the actions to perform.

The Action Handler object receives the request with all of its parameter values. It will perform the control logic for this particular request. During the control logic process, it calls any business rule methods which are encapsulated in the Session Façade and the Utility Bean. At the end of the control logic process, it determines the next JSP page which will be responded to the client. This selected JSP page is returned to the front-controller, the front-controller dispatches the next view to the Screen Flow Manager, a mechanism which is integrated inside the Apache Struts Servlet. The Screen Flow Manager will then respond the next view to display to the client.

The Session Façade is the front end of the model layer in this INCOM Application. All the business rules, utility beans and value list handler beans are encapsulated in this Session Façade. The value beans are encapsulated in the value list handler beans because for each value bean, there is a group of this value bean objects and there are a number of methods to process these objects in its group.

The value objects have the values needed to render on the JSP views. The View Helper uses the value objects to make up its custom handler tags. Both the value objects and custom handler tags are used in the JSP view to render dynamic content which are defined at run time of the application.

At the end, the JSP view is responded to the client. The client will decide what to do next and this process is repeated again and again until he leaves his session on the application.

Chapter 5 – System Realization

5.1) Setting up the Working Environment

- Installing the Java runtime environment (JDK 1.5)
- Installing Apache Tomcat Server
- Installing the Eclipse Java Development Tool Kit with the Web Tool Platform plug-in

When the above three requirements are ready on the machine, the next step is to run Eclipse and create a new dynamic web project.

The next figure is the Web Project file structure on the Eclipse JDK.

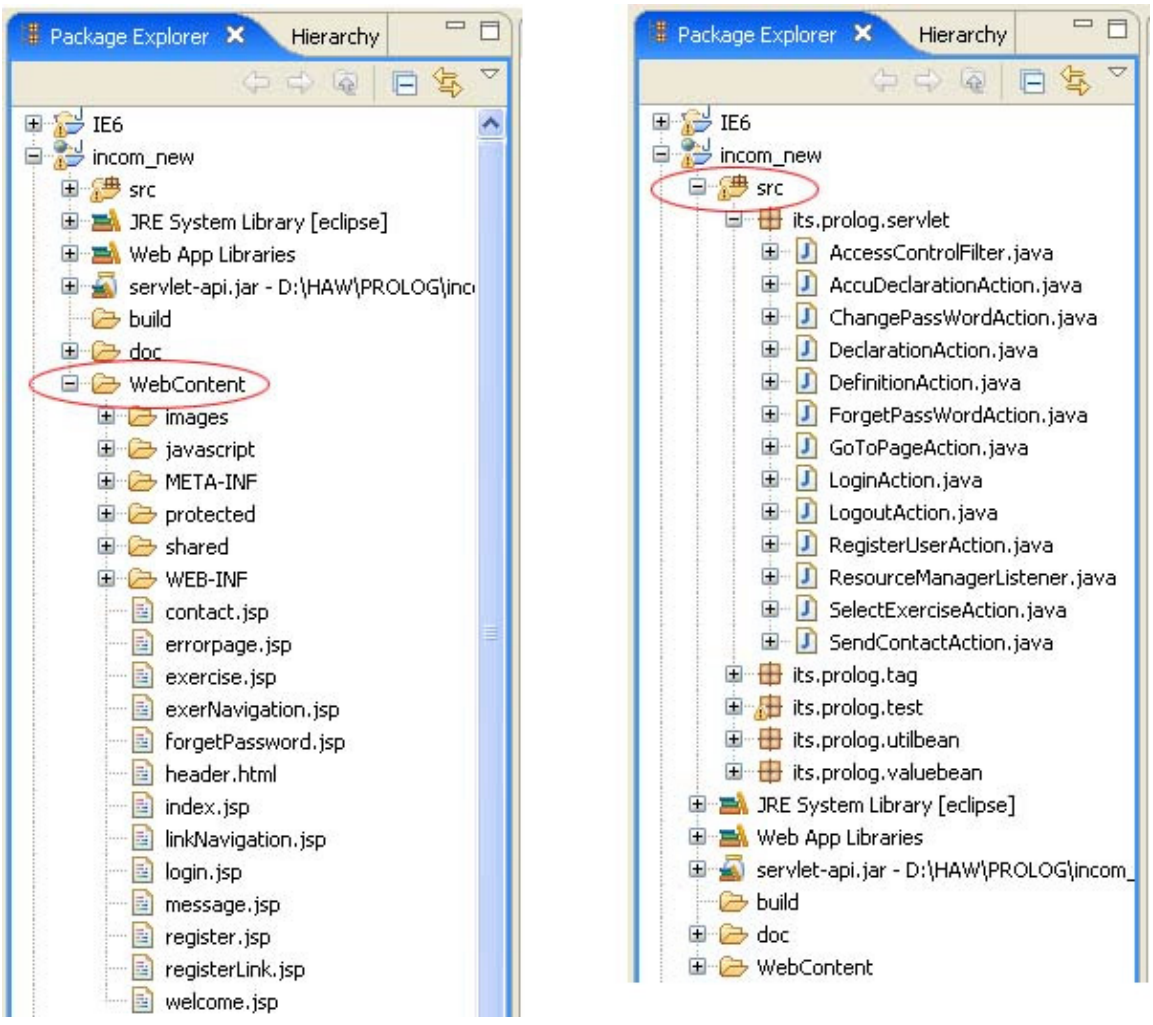


Figure 18: Eclipse Workspace for the INCOM front-end project

There are two main directories the “src” and the “WebContent” directories.

The “src” directory is where all the java packages are stored. During the development process, all java classes such as value beans, utility beans, value list handler beans, action classes, custom tag handler classes, Intercepting filter class, Listener class and JUNIT testing classes are located under this directory. These classes are grouped into different packages.

The “WebContent” directory is used for running the application on the server.
As can be seen, all the JSP files are located in this directory. Another important directory inside this WebContent is the WEB-INF directory. All the compiled java classes and java libraries with a “.jar” extension are stored under the classes and lib directories. There are sub directories in the class directory which reflect the package name in the src directory. When the application wants to search for a class method, he will check in the class directory first, if not found, he will check in the lib directory.

The WebContent/WEB-INF/tlds stores the XML files which are the configuration file of the standard and custom tag libraries. The configuration XML file – taglib.tld for the custom tag library will be discussed in section 5.2.2 when we talk about the implementation of the custom tag handler classes.

The WebContent/WEB-INF/Struts-config.xml file is the configuration file for Front-controller Apache Struts Servlet. The Struts Servlet uses this configuration file for mapping the request to a corresponding action class to process this request.

```
<action-mappings>
  <action path=”/logout” type=”its.prolog.Servlet.LogoutAction”/>
  <action path=”/register”
          type=”its.prolog.Servlet.RegisterUserAction>”
  ……..
</action-mappings>
```

<action-mapping>: the mapping pairs are declared as action sub elements inside the action-mapping element. The attribute path specifies the path to the Servlet action that will be requested. The type attribute is the correspondent Action class which is used to handle this request. The first example above is the logout request and the second example is the register a user’s account request.

Last but not least, a very important (WEB-INF/web.xml file) is the deployment descriptor file that contains all the configuration information for the INCOM Application.

```
<!--Struts Controller Servlet -->
<Servlet>
   <Servlet-name>action</Servlet-name>
   <Servlet-class>org.apache.struts.action.ActionServlet</Servlet-
class>
   <load-on-startup>1</load-on-startup>
</Servlet>

<Servlet-mapping>
   <Servlet-name>action</Servlet-name>
   <url-pattern>*.do</url-pattern>
</Servlet-mapping>
```

- `<Servlet>`: the Struts Servlet class is defined within this element. There are two sub elements. The `<Servlet-name>` is the logical name and the `<Servlet-class>` is the fully qualified class name `org.apache.Struts.action.ActionServlet`.
- `<Servlet-mapping>`: defines the extension for request mapping to this Servlet. The `<Servlet-name>` is the name of the declared Servlet name in the above sub element `<Servlet-name>` of `<Servlet>`. With this mapping the container will invoke the Struts Servlet for all requests that end with .do.

For example:

```
<form name="registration" action "register.do"> or
<a href="logout.do">Log out</a>
```

5.2) The MVC layers of this INCOM Application

5.2.1) The Model Layer Components

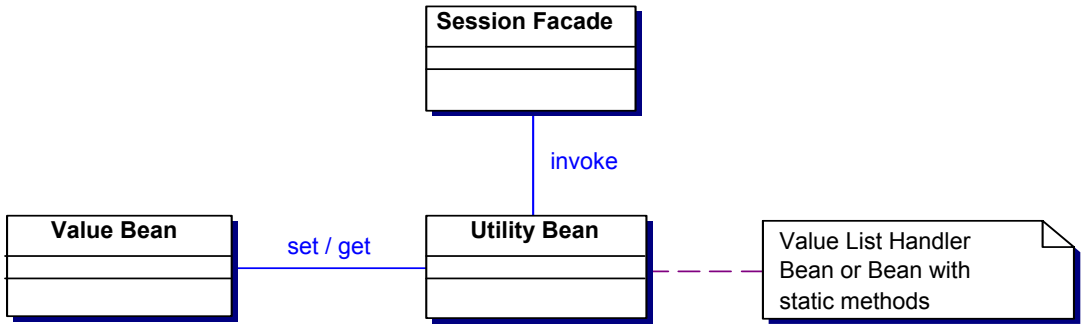


Figure 19: The model layer components

All the java bean components in this model layer implement the Serializable interface to support for persistence. Persistence in this context means that it is possible to save the state of a bean in an external format, such as a file. In addition, a bean in session scope can only be used in a distributed system application if it implements the Serializable interface. This is also an issue for the prospect flexibility and scalability of the application in the future.

To creating a Java bean class, one has to follow naming conventions.

- The bean class has no argument constructor
- The bean properties are accessible using the setter and getter methods. The setter and getter methods have to follow strictly the naming conventions for them. It is the word get or set, plus the property name with the first character of each word capitalized.

The java bean classes can be grouped into two packages, the value bean package and the utility bean package.

5.2.1.1) The value bean

The value bean contains only attributes, setter and getter methods. There is no business processing method in such a value bean. Since the value bean stores only values for an object, it benefits the reuse purpose.

Example of the `ArgumentBean.java` java bean class in the INCOM Application:

```
public class ArgumentBean implements Serializable
{
	// Properties of an argument
	private String name = "";
	private String type = "";
	private String mode = "";
	private String meaning = "";

	public void setName(String name){
		this.name = name;
	}

	public String getName(){
		return name;
	}
	……
}
```

5.2.1.2) Utility Bean

Different from the value bean, a utility bean has method for processing information or to perform some actions. We can divide the utility bean into two sub categories, the value list handler bean which follows the value list handler pattern and the bean class with static methods.

5.2.1.3) Value List Handler Bean Class

The value list handler beans have methods to process the information relating to the group of objects of the same value bean class.

An example of such a value list handler bean in the INCOM Application:

```
public class ArgumentListBean implements Serializable
{
    private ArrayList<ArgumentBean>
      arguments = new ArrayList<ArgumentBean>();

    public void setArguments(ArgumentBean argument){
      arguments.add(argument);
    }

    public ArgumentBean getArguments(int i){
      return (ArgumentBean)arguments.get(i);
    }

    public int getSize(){
      return arguments.size();
    }
    ……..
}
```

As we can see there is a list of arguments object in this class. The `ArgumentListBean` class contains the processing methods on this argument list. When a new argument is added, the `addAnEmptyArgument()` method is called. When the application wants to check the number of declared arguments, the `getSize()` method is called etc.

5.2.1.4) Bean Class with static methods

Since there are methods which are used by different bean classes because these methods process some common general tasks. These methods are declared as static and are put together in a utility bean class. These kind of utility beans help to simplify the code in other classes and they are reusable.

An example of a utility bean:

```
public class UtilBean implements Serializable
{
/**
 * This method is used to verify the inputs for the predicate
 * declaration form. These characters ; and ! are not allowed
 */
      public static String checkInputValue(String st) throws
            NullPointerException{
      if(!st.equals("")){
            st = st.replace(";", "<font color=red>;</font>");
            st = st.replace("!", "<font color=red>!</font>");
            if(st.charAt(st.length()-1) == ','){
                  st = st.substring(0, st.length()-1) + "<font
                        color=red><b>,</b></font>";
            }
      }
      return st;
    }
    ……
}
```

All the methods in the `UtilBean` class are declared as static methods. These are reusable methods and are called in different classes. The above `UtilBean` class has the `checkInputValue(String)` method which accepts a string as the input parameter . This parameter string is checked for invalid characters. If there are invalid characters in the input string, it returns a string which contains the HTML tag to highlight the invalid characters, otherwise the original input string is returned. As we can see, the task of this method is very general and can be used in other classes for this purpose.

To call a static method of the `UtilBean` class, one has to use the class name and then the method name with the specified parameters i.e.

```
String result =  UtilBean.checkInputValue(myString);
```

5.2.1.5) Multithreading Considerations

The java bean encapsulates the business logic in the application. When designing a java bean, we have to carefully consider thread safety problem for beans in session or application scope, especially, beans which share the same resource like reading or updating a text file. To avoid dirty read or update in this case, java provides mechanisms for dealing with concurrent access to resource such as synchronized blocks and thread notification methods.

The INCOM Application has one text file which contains the account information of all users. Since the account text file has to be read for validating valid users and has to be updated, the thread safety issue in this case is in consideration.

The `UserRegistryBean` class handles these tasks.

```
/*
 * insert a new account
 */
public static boolean register(String userName) throws
      IOException, Exception
{
      if(!EmailBean.isValidEmailAddress(userName)){
            return false; //wrong email format
      }
      else{
            synchronized(UserRegistryBean.prop){
            UserRegistryBean.inFile = new
                  FileInputStream("/conpro_user.account");
            ……
      }
}

/*
 * update the user's password
 */
public static boolean forgetPassWord(String username)
```

For the prospect/flexibility requirements, one more benefit is to use property files for the java bean classes wherever possible. A java bean uses a property file when there are frequently changed attribute values that the bean uses. If these values are stored in the property file, and when these values are changed, there is no need to compile the bean class again. This is often used when the application has to deal with different languages for displaying. The text for each language is stored in a property file, so that the bean can display different languages by reading different property files. In the INCOM Application, the `EmailBean` class uses a property file (mailDefaults.properties) for reading the SMTP server address and the email address of the administrator for sending email since these values can be changed quite often. And the content of the file is:

```
DEFAULT_SERVER=nats47
DEFAULT_TO=le@informatik.uni-hamburg.de
DEFAULT_FROM=le@informatik.uni-hamburg.de
```

5.2.2) The View Layer Components

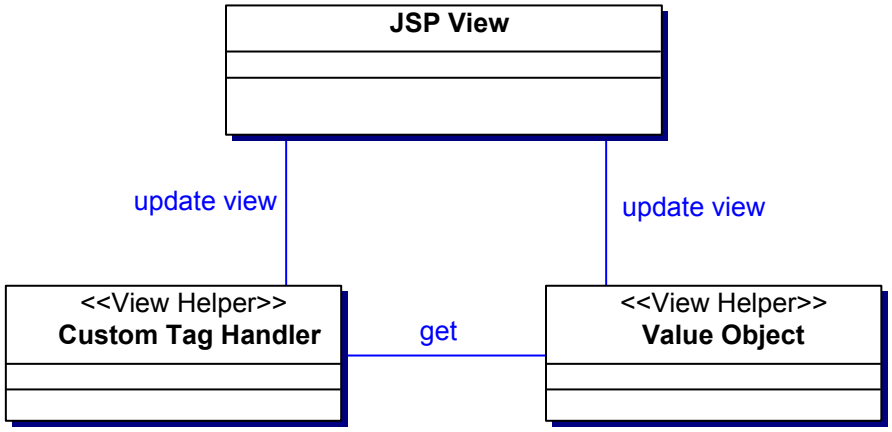


Figure 20: The View Layer Components

Besides using the JSTL[21], this view layer also uses a technique called the View Helper pattern. There are two types of View Helper, one is Value Object, and the other is Custom Tag Handler. The INCOM Application uses both types of the View Helper pattern.

5.2.2.1) JSTL Example

```
<c:choose>
      <c:when test="${!empty syntaxError}">
            There is syntax error
      </c:when>
      <c:otherwise>
            There is no syntax error
      </c:otherwise>
</c:choose>
```

The tags `<c:choose>`, `<c:when>`, `<c:otherwise>` are standard tags. The meaning of the above code is that it checks if there is syntax error, it displays the string "There is syntax error", and otherwise it displays the string "There is no syntax error". It works like an if-else statement.

[21] JSTL stands for JavaServer Pages Standard Tag Library. JSTL encapsulates as simple tags, core functionality common to many JSP application. – Sun Microsystems Inc. JSP Document – URL http://java.sun.com/products/jsp/jstl/reference/docs/index.html - 02.May.2007

5.2.2.2) Value Object

The Value Object encapsulates the intermediate model state for use by the view. In this application, all these value beans are encapsulated in the value list handler beans which are accessed under the session façade `PredicateBean` object.

In order to access a value object in a JSP page, one can use the expression language [22] as follows:

```
<input type="text" name="argName"
value="${predBean.arguments.arguments[index].name}">
```

`${predBean.arguments.arguments[index].name}` is called expression language.

The above is the HTML tag to create an input text field. The text field is filled with a default value of a name of an argument. This value argument object is located at the index position in the list of arguments named arguments. This list of arguments named arguments is encapsulated in the session façade PredicateBean object named predBean.

The expression language code in the JSP page is corresponding to the following java code:

```
predBean.getArguments().getArguments()[index].getName();
```

5.2.2.3) Custom Tag Handler

Custom tag encapsulates the presentation logic in the form of a normal HTML tags which are familiar to the page author. "Using these actions, the amount of Java code in your JSP pages can be kept to a minimum, making your application easier to debug and maintain". (Hans Bergsten, 2003, P. 421)

In the INCOM Application, custom tags are implemented using java classes. These classes are called tag handler classes. A custom tag may have attributes and its body. All the custom tags used in the application do not have body, some of them have attributes, and some do not have any attribute.

A tag handler class is in fact a bean with setter methods for its attributes or without any setter method if it has no attribute. Since JSP 2.0 version, Simple Tag Handler was introduced. This technology was used to develop the tag handler classes in this Application. To develop a simple tag handler class, one has to use a set of classes and interfaces. As an example, the following class is a tag handler class which was developed and used in the INCOM Application.

---

[22] An expression language makes it possible to easily access application data stored in JavaBeans components. Sun Microsystems Inc. 2002 – J2EE 1.4 Tutorial – URL http://java.sun.com/j2ee/1.4/docs/tutorial/doc/JSPIntro7.html - 06.May.2007

```
public class DeclarationColumnTitleTag extends SimpleTagSupport
{
      private String column;

      public void setColumn(String column) {
            this.column = column;
      }

      public void doTag() throws JspException, IOException {
            StringBuilder buff = new StringBuilder();
            ArgErrorBean currentError =
                  (ArgErrorBean)getJspContext().
                  getAttribute("currentError",
                  PageContext.SESSION_SCOPE);

            //Predicate Name header in declaration page
            if(this.column.equalsIgnoreCase("predicateName")){
                  if(currentError != null &&
                        currentError.getErrType().
                        equalsIgnoreCase("pred")){
                        buff.append("<font
                        color=\"red\">Prädikatsname:</font>");
                  }
                  else{
                        buff.append("Prädikatsname:");
                  }
            }
            else if

            ……

            //print the buffer onto the jsp page
            getJspContext().getOut().print(buff);
      }
}
```

A tag handler class has to implement the `javax.Servlet.jsp.tagext.SimpleTag`. In this class, it extends the base class `javax.Servlet.jsp.tagext.SimpleTagSupport` of the `SimpleTag` interface. It has setter methods for its column attribute and implements the `doTag()` method.

The `doTag()` method in this class will print the column title or argument row header on a JSP page with or without highlighted color depends on the current error of the solution evaluation.

One note on this class is that it accesses to the `currentError` session attribute in order to check the type of this current viewed error on the JSP page. If this `currentError` session attribute exists, the type of this error will decide whether to highlight the column attribute or not.

To access to the session attribute in this class, the following code is used:

```
//session attribute currentError is accessed using
ArgErrorBean currentError =
(ArgErrorBean)getJspContext().getAttribute("currentError",
PageContext.SESSION_SCOPE);
```

The next figure shows us the use of this custom tag. The row header “Argument 2:” and the column header ‘Dieses Argument bezieht sich auf” which are displayed by the `DeclarationColumnTitleTag` custom tag handler class in the below figure are highlighted because the error location of the current viewed error is at the position of “Dieses Argument bezieht sich auf” on “Argument 2”.

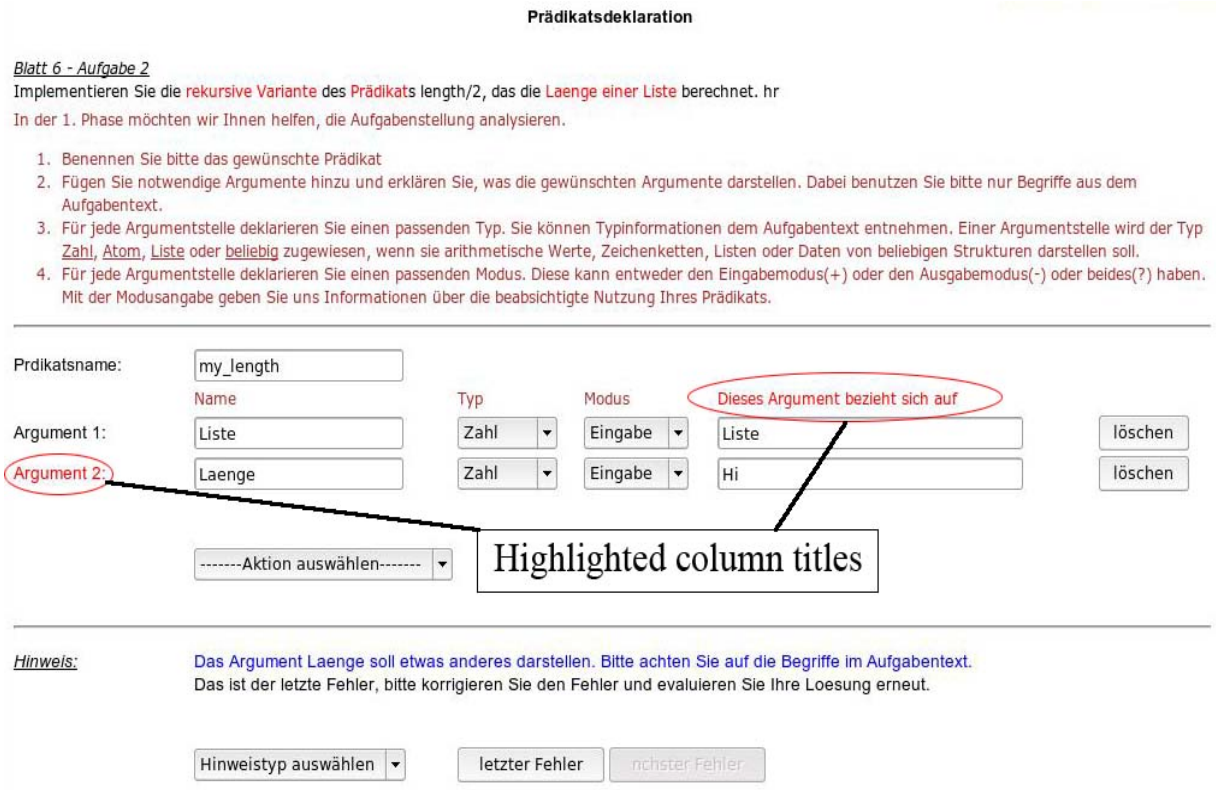


Figure 21: Custom tag components in a JSP page

To compile this tag handler class, one has to include the Servlet and JSP API in the classpath which are the Servlet-api.jar and jsp-api.jar located in the common/lib directory for Tomcat. These classes are placed in the WEB-INF/classes directory.

The next step is to declare this custom tag in a TLD [23] file.

[23] TLD stands for Tag Library Descriptor. The custom tags have to be declared in this descriptor file in order to make it available to use in the JSP pages of the application

```
<description>
    A tag library for the INCOM project
</description>
<tlib-version>3.0</tlib-version>
<short-name>incom</short-name>
<uri>its.prolog.mylib</uri>

<tag>
    <description>
      Format the color of the column titles in the declaration page
      Used in Declaration, Accu Declaration
    </description>
    <name>declarationColumnTitle</name>
    <tag-class>its.prolog.DeclarationColumnTitleTag</tag-class>
    <body-content>empty</body-content>

    <attribute>
      <name>column</name>
      <required>true</required>
      <rtexprvalue>false</rtexprvalue>
    </attribute>
</tag>
```

The TLD maps the custom action to the corresponding tag handler class which is its.prolog.DeclarationColumnTitleTag in this case and defines the column attribute of this class. The TLD file is placed in the WEB-INF/tlds directory with the name mylib.tld.

Since these custom tags are programmed by a programmer, but are used by a page author, the page author has to know what information a custom tag displays and whether this custom tag requires any attribute or a body. For this reason, an API reference for these custom tags is needed.

The API reference for all the custom tags used in this INCOM Application can be found in the customtagApi.pdf file which is available in the CD included together with this report.

5.2.3) The Controller Components

The controller uses the Servlet technology. By using the open source Apache Struts Servlet, there is no need to write any Servlet class in this INCOM Application. All the requests and responses are managed by the Struts. The components used in this layer are the Struts Servlet, Action classes used together with Struts, a resource initialization listener class, an access control intercepting filter class. These components are organized as the class diagram below:

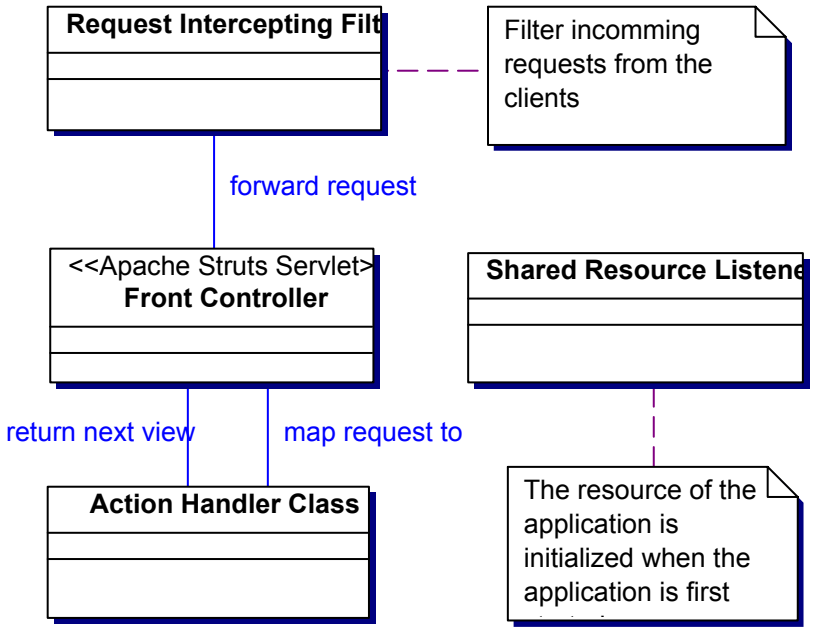


Figure 22: The Controller Layer Components

5.2.3.1) The Shared Resource Listener

Since there is one exercise list which is displayed to all user's browser, this exercise list is registered as the application scope [24] attribute. "An application lifecycle listener is a perfect tool for this type of resource initialization. This kind of listener implements the `javax.Servlet.ServletContextListener` interface, with methods called by the container when the application starts and when it shut down." (Hans Bergsten, 2003, P. 380)

[24] Application scope attribute is the global attribute which is shared by all user's sessions and is accessible throughout a Web Application

```
public class ResourceManagerListener implements
ServletContextListener
{
/*
 * When the server is started, then initialize a list
 * of exercises as application scope attribute
 */
      public void contextInitialized(ServletContextEvent sce)
      {
            ServletContext application = sce.getServletContext();
            ExerciseListBean exerciseList = null;
            application.setAttribute("exerciseList", exerciseList);
      }

/*
 * Destroy all the application resource when stop the server
 */
      public void contextDestroyed(ServletContextEvent sce)
      {
            ServletContext application = sce.getServletContext();
            application.removeAttribute("exerciseList");
      }
}
```

The `contextInitialized()` method is called when the application starts, before any requests are delivered. The object `ServletContextEvent` parameter has method for getting the `ServletContext` instance of this application. This instance is important since it provides methods for working with the application scope attributes and sharing data between Servlet, listener, filter, java bean, and JSP pages. In the above source code, the exercise list is registered as the application scope attribute in the `contextInitialized()` method and is removed in the `contextDestroyed()` method when the application stops running.

To make this listener available to the INCOM Application, it has to be declared in the application deployment descriptor (the WEB-INF/web.xml file). The following listener declaration has to be added to the web.xml file:

```
<listener>
    <listener-class>
       its.prolog.Servlet.ResourceManagerListener
    </listener-class>
</listener>
```

5.2.3.2) Access Control using a Filter

In this INCOM Application, all requests that need access control are shown in the next table:

Context-relative path	Resource
/protected/changePassword.jsp	The change password JSP page
/protected/predicateDeclaration.jsp	The predicate declaration JSP page
/protected/predicateDefinition.jsp	The predicate definition JSP page
/protected/predicateAccu.jsp	The Accu predicate declaration JSP page
/selectExercise.do	The action for selecting an exercise

Table: INCOM Application context-relative URI paths for access control filter

All the JSP pages in the /protected/* directory are accessible only for authenticated users. The /selectExercise.do is the URI that invokes the Servlet. In order to redirect all requests matching these URI paths through an access-control filter, the following xml descriptor has to be added to the application deployment descriptor (WEB-INF/web.xml file):

```
<filter>
      <filter-name>
            accessControl
      </filter-name>
      <filter-class>
            its.prolog.Servlet.AccessControlFilter
      </filter-class>
      <init-param>
            <param-name>loginPage</param-name>
            <param-value>/login.jsp</param-value>
      </init-param>
</filter>

<filter-mapping>
      <filter-name>accessControl</filter-name>
      <url-pattern>/protected/*</url-pattern>
</filter-mapping>

<filter-mapping>
      <filter-name>accessControl</filter-name>
      <url-pattern>/selectExercise.do</url-pattern>
</filter-mapping>
```

The `<filter>` and nested `<filter-name>` elements define a name and the implementation class for this filter. The `<init-param>` element defines a context-relative path to the login JSP page in the loginPage parameter which can be used in the filter class.

The <filter-mapping> tells the container to redirect all requests matching the <url-pattern> to the filter accessControl which is defined in the <filter> element. It is possible to add more filter mapping patterns to this filter.

The implementation of the filter class is as follow:

```
public class AccessControlFilter implements Filter {

    private FilterConfig config = null;

    public void doFilter(ServletRequest request, ServletResponse
            response, FilterChain chain){
        HttpServletRequest httpReq = (HttpServletRequest) request;
        HttpServletResponse httpResp = (HttpServletResponse)response;

        if (!isAuthenticated(httpReq)) {
            String forwardURI = getForwardURI(httpReq);

            // Forward to the login page and stop further processing
            ServletContext context = config.getServletContext();
            RequestDispatcher rd =
                  context.getRequestDispatcher(forwardURI);
            rd.forward(request, response);
            return;
        }
        chain.doFilter(request, response);
    }

    ……
}
```

A filter class must implement the javax.Servlet.Filter interface. This interface provides the methods init(), doFilter() and destroy() which are needed to initialize, to process and to destroy a filter. The parameter in the init() method is the reference to the FilterConfig instance so that the filter can access to the ServletContext object. The ServletContext interface defines a set of methods that a Servlet uses to communicate with the Servlet container. These methods are important to use and create shared attributes or to dispatch requests. The context-relative of the login page is initialized so that all the invalid requests are redirected to this login page. The destroy() method is used to release the resource of the filter.

The main process of the filter is done in the doFilter() method. Whenever it receives a request matching a mapping pattern for this filter, it checks if the user is authenticated by looking at the session bean object validUser. If it can not find this session object, it forwards the user to the login page.

5.2.3.3) Centralized request processing using Apache Struts Servlet

This is the main part of the controller layer components to process the control logic of the application. Luckily, most of implementation of this controller is done in the Apache Struts Servlet as mentioned in the Controller layer in the application design chapter. The left implementation for this controller is that we have to implement the action classes to handle different requests.

In order to use this Struts Servlet in this INCOM Application, first of all, it has to be declared to the application. The following descriptor is added to the (WEB-INF/web.xml file):

```
<Servlet>
      <Servlet-name>action</Servlet-name>
      <Servlet-class>
            org.apache.struts.action.ActionServlet
      </Servlet-class>
      <load-on-startup>1</load-on-startup>
</Servlet>

<Servlet-mapping>
      <Servlet-name>action</Servlet-name>
      <url-pattern>*.do</url-pattern>
</Servlet-mapping>
```

One remark in this descriptor about the value of the <url-pattern> element, the container will invoke the Struts Servlet for all requests with the ending .do.

e.g.

```
<form action="login.do"> or
<a href="protected/logout.do">Logout</a>
```

And the implementing of an Action Class

```
public class LoginAction extends Action {
    public ActionForward perform(ActionMapping mapping,
        ActionForm form, HttpServletRequest request,
        HttpServletResponse response) throws IOException,
        ServletException {

        HttpSession session = request.getSession(false);
        if (session != null) {
          session.invalidate();
        }
        ActionForward nextPage = mapping.findForward("login");
        return nextPage;
    }
}
```

All the action classes extend the Struts Action class. The main method which is first invoked in this class is the perform() method. The above action class handles the logout request.

When the logging in user click on the log out link Logout, the the LogoutAction class is performed to end the current session and redirect the user to the login page. The mechanism used to pass this request to a corresponding action class will be discussed later.

The perform() method returns the nextPage ActionForward object which contains information about the JSP page that the Screen Flow Management of the Struts will have to invoke for the response.

```
ActionForward nextPage = mapping.findForward("login");
return nextPage;
```

One remark about the use of the logical names `logic` as the parameter in the `findForward("login")` method is that the login logical name identifies the corresponding JSP pages which are declared in the configuration file of the Struts Servlet. The benefit of this is that the page flow can be controlled outside the class source code, so that the page flow can be change easily without the need to change the source code.

The login logical name and the mapping information to the `LoginAction` class are declared in the (WEB-INF/Struts-config.xml)

```
<global-forwards>
      <forward name="main" path="/welcome.jsp" redirect="true" />
      <forward name="login" path="/login.jsp" redirect="true" />
</global-forwards>

<action-mappings>
      <action path="/login" type="its.prolog.Servlet.LoginAction">
            <forward name="toSelectExerciseAction"
                  path="/selectExercise.do" />
      </action>
</action-mappings>
```

The request URI to this `LoginAction` class is `/login.do`, but in the upper action mapping path, it is `/login` without the ending .do. It is because Struts Servlet has already removed the ending .do of the request URI after receiving this request.

It is clear in the `<action>` element that for this `login.do` request, it is passed to the LoginAction class. One interesting thing in this action mapping declaration is that the `<action>` element has a nested `<forward>` element. The use of this nested `<forward>` element is when the user has logged in, it will continually redirect to the `selectExercise.do` Servlet if the user has tried to select any exercise before logging in. This automatic redirection of request helps the user not to select the exercise again.

5.3) Using a common JSP error page

During the running process of the application, bug exceptions may occur. The users do not want to see these bug exception in their JSP pages. And according to the friendly user interface issue, a friendly common JSP error page is used for all runtime errors, no matter if they occur in the JSP pages, in Servlet, Action classes, Listener or java beans.

For this purpose, in the application deployment descriptor (WEB-INF/web.xml file), the following declaration is added:

```
<error-page>
      <exception-type>java.lang.Throwable</exception-type>
      <location>/errorpage.jsp</location>
</error-page>
```

The `<exception-type>` element contains the full package name and class name of the type of the exception. If any exception of this type occurs, the `/errorpage.jsp` will be responded to the user. This helps the application still to run if there is any exception occurs, and it responds a friendly error page to the user. Hopefully, the user will not get angry if any exception occurs.

Of cause, one can use any other `<error-page>` elements for different error pages for different exceptions.

When sending a HTTP request to the server, the server could also respond a HTTP error code e.g.

- 404 - The server has not found anything matching the URL given
- 500 - The server encountered an unexpected condition which prevented it from fulfilling the request.
- etc.

In this case, the `<exception-type>` element is replaced by the <error-code> element.

```
<error-page>
      <error-code>404</error-code>
      <location>/notfound.jsp</location>
</error-page>
```

The errorpage.jsp displays a friendly error message to the users.

5.4) The total picture of the INCOM Application Realization

The next two figures show the total picture of the INCOM Application Realization

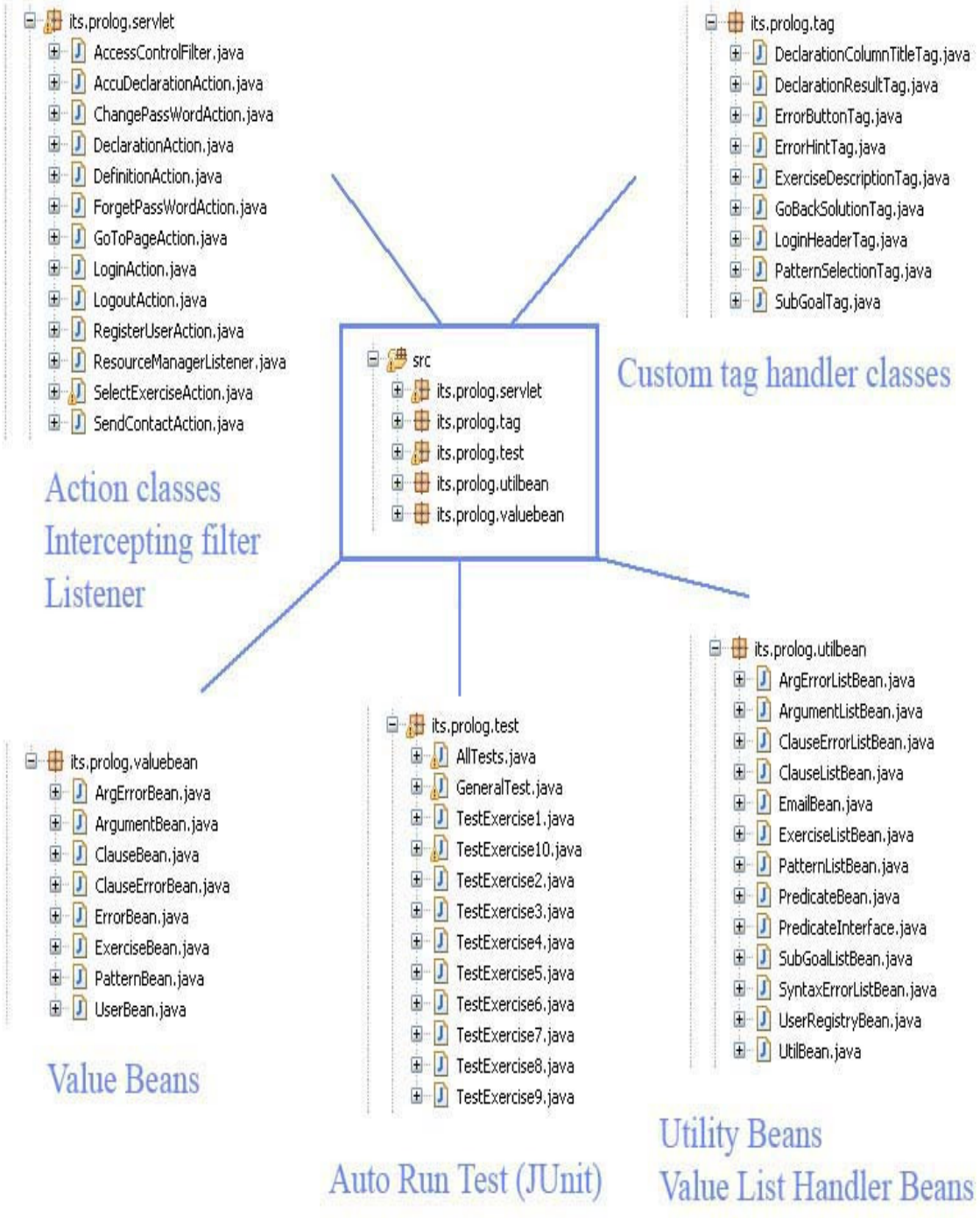


Figure 23: The Java packages and classes of the INCOM Application

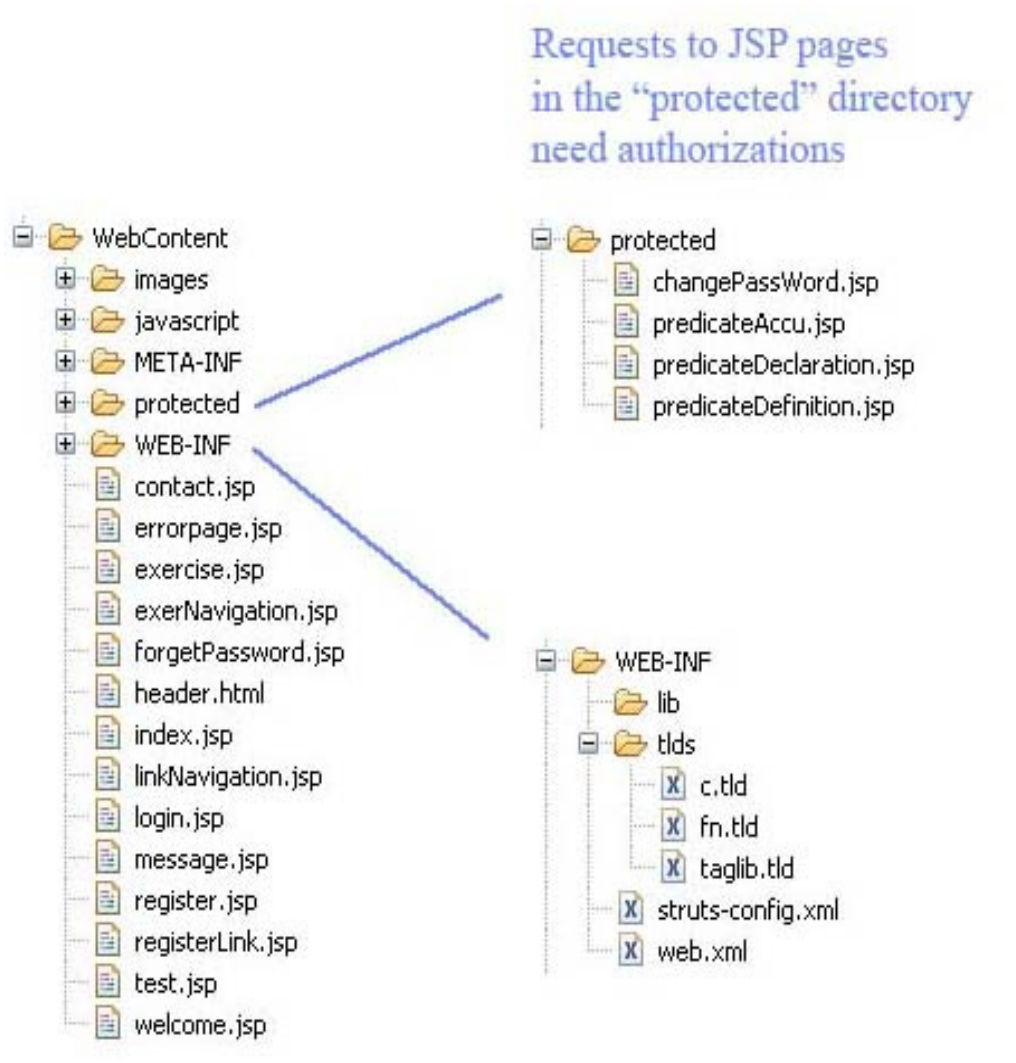


Figure 24:The JSP pages and descriptor files of the INCOM Application

Chapter 6 - Application Testing

6.1) Auto Run Test with JUnit 4.0

6.1.1) What is JUnit?

"JUnit is a simple, open source framework to write and run repeatable tests. It is an instance of the JUnit architecture for unit testing frameworks. JUnit features include:

- Assertions for testing expected results
- Test fixtures for sharing common test data
- Test runners for running tests

JUnit was originally written by Erich Gamma and Kent Beck." (JUnit.org)

6.1.2) Testing with JUnit

The testing is programmed in the front-end in java with Junit and the back-end in prolog with special constrains for different test cases. The purpose of this test is to check the correctness and stability of the front-end and also of the back-end.

In the INCOM Application, there are a number of exercises available for the learning process. Each exercise has different correct solutions. Now consider that we want to test exercise number one. Of cause, we will try to use many different solutions for this exercise. These test case solutions are not decided at runtime, but they are already defined in the back-end. For each of the test case solutions in the back-end, it has its returned value. The front-end has to use these re-defined solutions to send to the prolog back-end, and receives the returned value from the back-end. This returned value is then compared to the re-defined result of this test case solution which is defined in the back-end. These solutions and their re-defined results are known to the programmer who programs the test cases in the front-end.

If the returned value and the re-defined result match, this test case is correct. Otherwise there is error in the front-end or in the back-end, and they have to be checked for the correctness for this test case.

Implementation of a test case for exercise 1:

Test case in the back-end

The back-end provides the test case solution and its re-defined result for this test case solution. And the first test case for exercise number one is as follow.

A) Predicate declaration page:

Predicate name: zins
Pattern: no_pattern_recursiv

	Name	Type	Mode	Meaning
Argument 1	Arg1	number	+	Betrag
Argument 2	Arg2	number	+	Satz
Argument 3	Arg3	number	+	Anlagejahr
Argument 4	Arg4	number	-	Endguthaben

The result for this predicate declaration test case: no error.

B) Predicate definition page:

The list of declared clauses:

	Type	Head	Body
Clause 1	`basecase`	`zins(Betrag,_,0,Betrag)`	
Clause 2	`recursivecase`	`zins(Betrag, Satz, Zeit, End)`	`Zeit>1, NeuBetrag is Betrag*(Satz+1), NZeit is Zeit -1, zins(NeuBetrag, Satz, NZeit, End)`

The result for this predicate definition: 1 error and the error code is “s8c”

Test case in the front-end

```
public class TestExercise1 extends TestCase {
      @BeforeClass
      public static void testOneTimeSetUp()
      public void teardown()
      public void testExer1_1() throws PatternSyntaxException,
            NumberFormatException, IndexOutOfBoundsException,
            JPLException, NullPointerException, IOException {

            clauseBean[0] = new ClauseBean("basecase",
                  "zins(Betrag,_,0,Betrag)", "");
            predBean.getClauses().setClauses(clauseBean[0]);

            //invoke this method to process the evaluation
            predBean.evaluatePredicateDefinition();

            assertEquals("Exer1_1 should have 1 error:", 1,
                   predBean.getClauseErrorList().getSize());
            assertEquals("1st error should be [s8c]: ", "s8c",
                  predBean.getClauseErrorList().
                        getClauseErrorList()[0].getId());
            System.out.println("Test case 1.1 is correct");
      }
}
```

The `testOneTimeSetUp()` method is run only once at the beginning of the test case JUnit class for exercise one. The annotation `@BeforeClass` marks this method as the method which is needed to be run at first. In this method, all the attributes which are used by different test cases are initialized.

Since all the test cases for exercise 1 use the same test case for the predicate declaration page which should be correct so that it can continue to do the test in the predicate definition page. Four arguments are declared, the predicate's name and the predicate pattern are set. Now all the needed value for our test case solution is ready, it is time to evaluate this solution by calling the `evaluatePredicateDefinition()` method. And this solution expects no error by comparing the length of the error list with zero. If they are equal, then the string "predicate declaration is correct" is printed out. Otherwise, the string "Exercise 1 Declaration should have no error but found 1 or 2 or etc." is printed out.

The `testExer1_1()` method is then run after the `testOneTimeSetUp()` and the `tearDown()` method. This is the test case for the predicate definition process. The procedure is similar to the test case in the predicate declaration page. But in this test case it expects one error with the error code "s8c" for this solution. That means it has to compare the number of errors and the error codes.

The `tearDown()` method is run after each test case has finished. The purpose is to reset all the attributes for the next test case to run.

Beside `testExer1_1()`, one can add different test cases in different methods like `testExer1_2()`, `testExer1_3()` etc.

Figures 23 and 24 show the screens of the auto test on the shell command window when the auto test starts and finishes successfully.

```
tran@node8:~/conpro/frontend$ ant runtest
Buildfile: build.xml

prepare:

compile:

runtest:
     [echo] Loading libraries1: /opt/pkg/pl/lib/pl-5.6.6/lib/i686-linux/
     [java] initExerciseList()
```

Figure 25: Start of the auto test

```
     [java] ENDING**********PredicateBean.java : setPredDefinition()*************
     [java] ............Finished Testing Exer10_6 Successfully!!!.....................


     [java] ------------------------------------------------------------


     [java] Time: 135,132

     [java] OK (52 tests)


BUILD SUCCESSFUL
Total time: 2 minutes 17 seconds
tran@node8:~/conpro/frontend$
```

Figure 26: End of the auto test

The INCOM Application was tested with 52 test cases for ten different exercises. The line [java] OK (52 tests) in figure 24 shows that all the 52 test cases are passed successfully.

6.2) Usability Test

With the giving input as in figure 27, errors are responded to the client.

Figure 28 shows the correct solution for the predicate declaration of an exercise. And after submitting this solution for the predicate declaration page, the client moved to the predicate definition. In the predicate definition page, the client decided to select the Accu declaration page and he submitted a wrong solution for the Accu declaration page which is shown in figure 29.

In figure 30, it is the correct solution for the Accu declaration page. After submitting the solution in figure 30, the client moved back to the predicate definition page and he tried to submit a wrong solution for the definition page which is shown in figure 31.

At last, he submitted the correct solution in figure 32 for the predicate definition page and it means that he finished the exercise successfully.

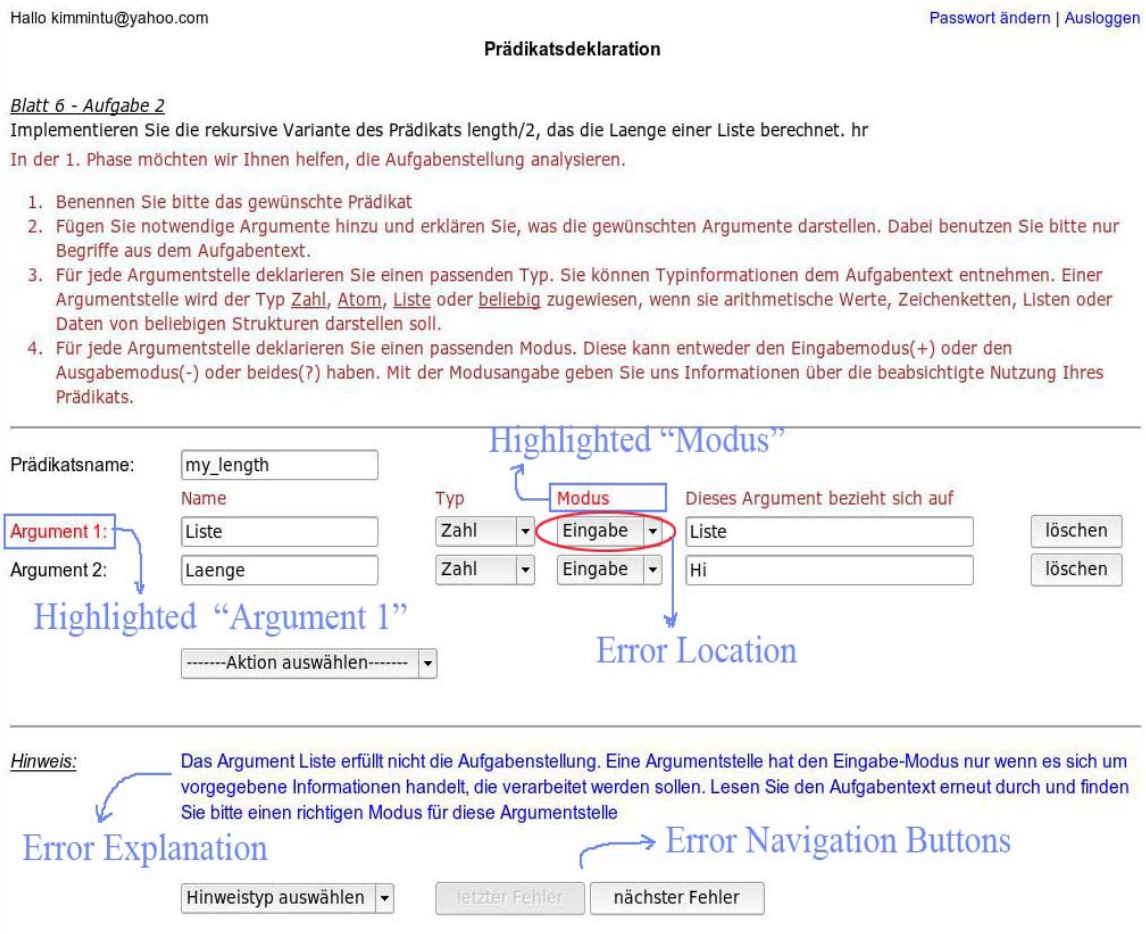


Figure 27: Wrong solution for the predicate declaration page

Figure 28: Correct solution for the predicate declaration page

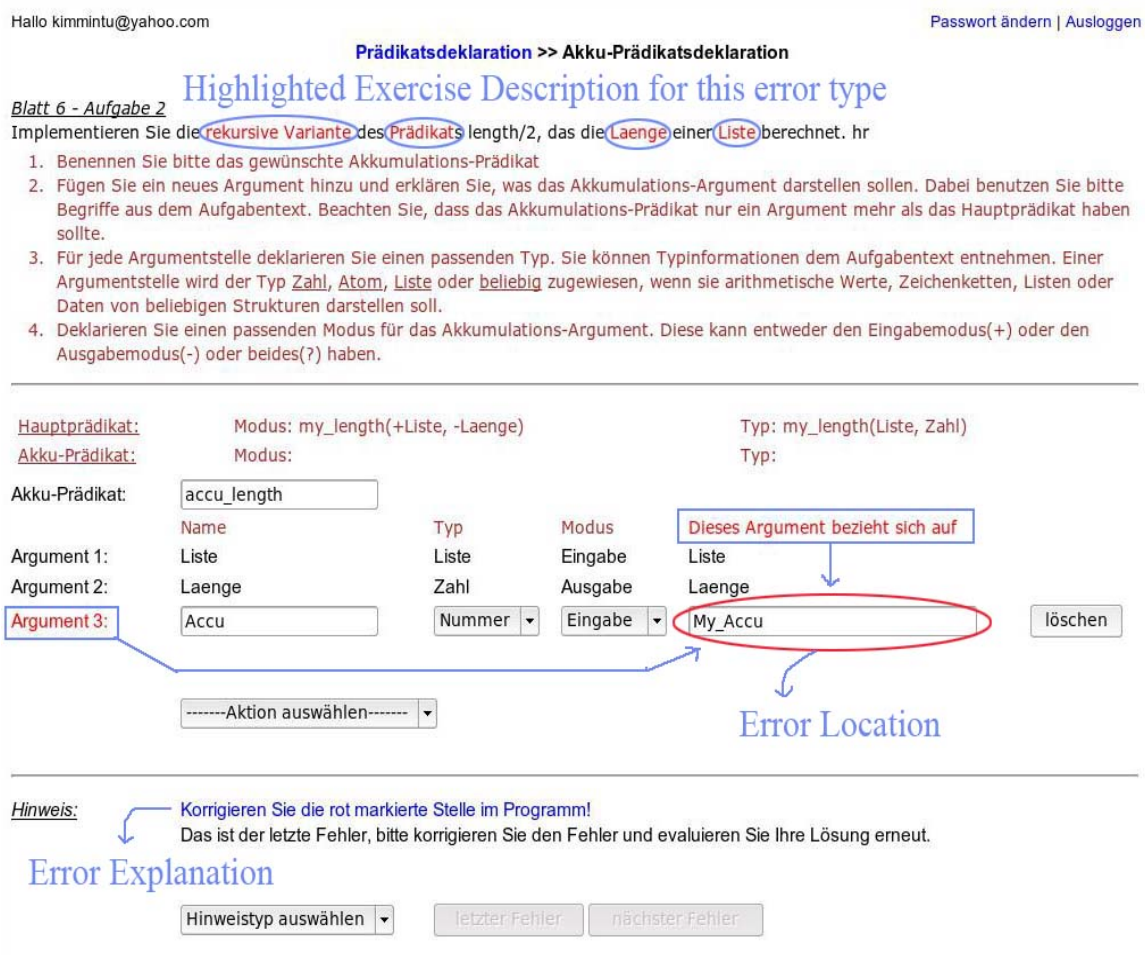


Figure 29: Wrong solution for the Accu predicate declaration page

Figure 30: Correct solution for the Accu predicate definition page

Figure 31: Wrong solution for the predicate definition page

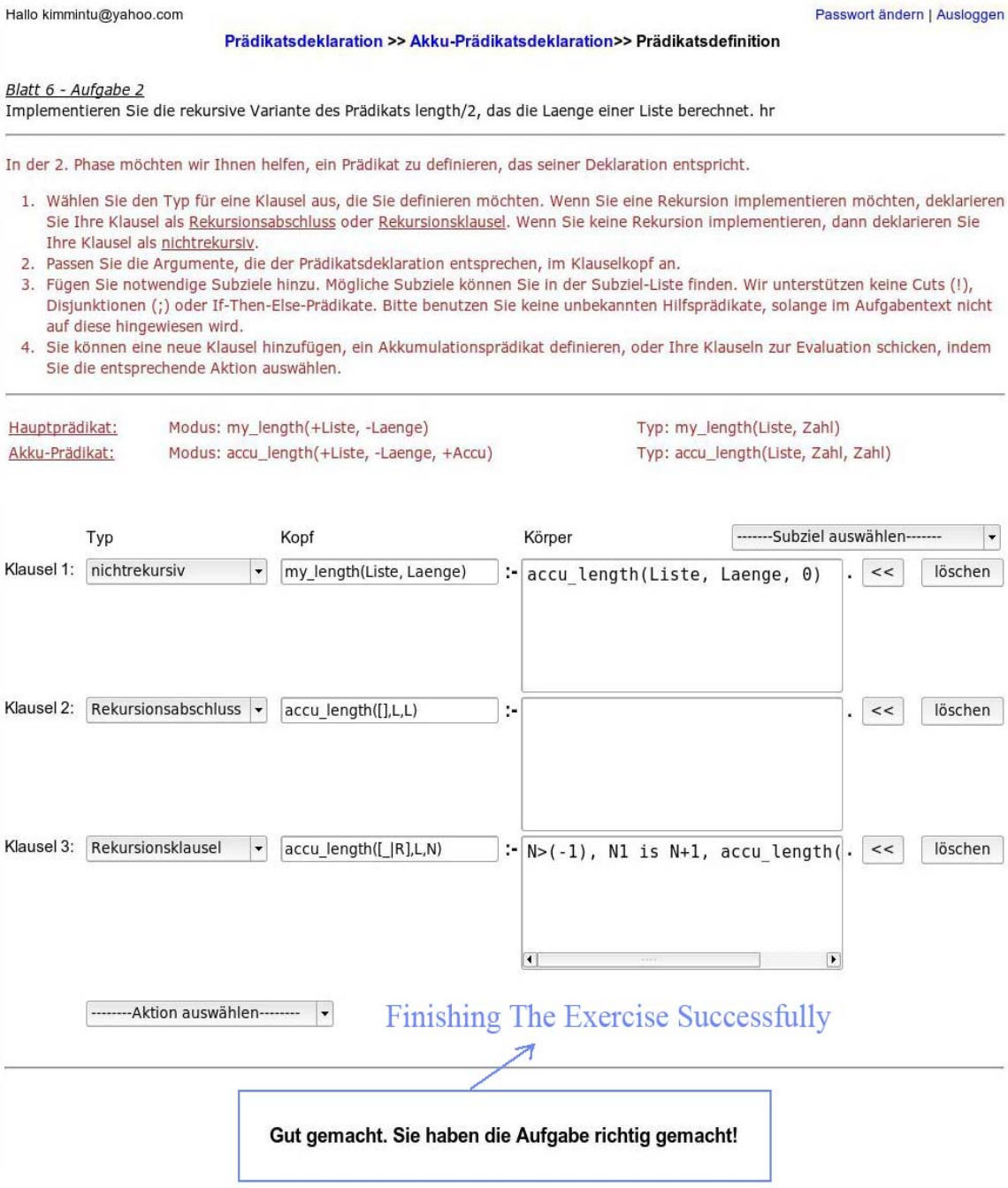


Figure 32: Correct solution for the predicate definition page

Chapter 7 - Resume

This chapter begins with a conclusion on the INCOM Application from the Know How concept to the technique implementation. And the question of how flexibility of the application for further development in the future will be discussed.

7.1) Conclusions

In order to develop an enterprise application in the J2EE platform, one needs to know the requirements and the tools one can use. J2EE is a large platform; it provides different tools for different purposes. The INCOM Application is a typical a client-sever application. And according to the blueprint guidelines web pages on the java.sun.com site, "J2EE applications that are interactive benefit from using the Model-View-Controller (MVC) architecture. MVC is particularly well-suited for interactive Web applications-applications where a Web user interacts with a Web site, with multiple iterations of screen page displays and multiple round-trips of requesting and displaying data." (Inderjeet Singh, et al., 2002).

The MVC model was decided to use since this architecture is supported by a number of different design patterns which are recommended on the Core J2EE Patterns[25] web pages on the java.sun.com site. Further more, there are very good open source software to support this architecture, especially is the Apache Struts Servlet used in the Controller layer.

This report described how the MVC architecture applies to the INCOM Application in the J2EE platform. It showed how the recommended design patterns for a J2EE application are used to decompose the application into smaller components.

The functionality of the application is partitioned into modules[26]; each module is partitioned into component objects by applying the appropriate design pattern for it. This helps to re-use the source code, to separate stable code from frequently changed code, and to maintain and extend the application more easily.

[25] The Core J2EE Patterns homepage can be found at http://java.sun.com/blueprints/corej2eepatterns/index.html

[26] Modules here are the three layers of the MVC architecture: Model layer, View layer and Controller layer

7.2) Prospect

The INCOM Application was designed with the issue of prospect flexibility in mind at the beginning. It would be easy to have any further development. More functionality could be added on without any changing to the current state of the application. This means that the design and source code implementation need to be well structured and well documented so that other developers can understand and have further development on the application latter on. In order to do this, the "reuse" and "dividing application into smaller components" issues are in consideration. When the application is divided into smaller components, it is easy to change one component without any effect on other components. The application development can be assigned to different group and each group can work on their components. It helps to solve the debug problem in a big application and saves a lot of time for the development process. This can be realized by using the MVC architecture and the core J2EE design patterns for this INCOM Application. Once the application has the "reuse" components, the coding process can be shortening, the structure of the source code becomes clearer to developers. This coding style is helpful for a new developer to read and understand the application quickly. This can be realized by using the CSS technique for the template of a web page, or by using the Utility classes which have general common methods.

Nowadays, there are many E-learning applications using the client-server model. Besides the stability issue and other issues, the friendly user interface issue is also important for the successful of an application. For this purpose, the dynamic animation using MX Flash technology would improve the front-end of the INCOM Application. For the dynamic content, the MX Flash needs to use the XML technology.

This future implementation would not be a problem since the presentation module of this application is separated from the model and controller modules. Further on, there is no control logic in the view module and the presentation logic of the view is kept simple and encapsulated using the custom tag handler classes. This helps to have any further development on the application with a minimized possibility in re-designing as well as code changing.

Figure Illustration Index

References

[Nguyen-Thinh Le, 2004] Nguyen-Thinh Le: INCOM.pdf: Evalution of prototype Prolog Database query – URL http://nats-www.informatik.uni-hamburg.de/pub/INCOM/Dokumentation/INCOM.pdf - 06.May.2007

[Publications of Nguyen-Thinh Le] Nguyen-Thinh Le: https://nats-www.informatik.uni-hamburg.de/view/User/ThinhsPublications - 28.May.2007

[Sun Microsystems Inc.] Sun Microsystems Inc.: Java EE at a Glance – URL http://java.sun.com/javaee/ - 06.May.2007

[Inderjeet Singh, et al, 2002] Inderjeet Singh, Beth Stearns, Mark Johnson, and the Enterprise Team: Designing Enterprise Applications with the J2EE™ Platform, Second Edition, Sun Microsystems Inc. – URL http://java.sun.com/blueprints/guidelines/designing_enterprise_applications_2e/DEA2eTOC.html - 06.May.2007

[Apache Software Foundation, Struts] Apache Software Foundation, Struts – URL http://struts.apache.org/ - 06.May.2007

[Apache Software Foundation, Apache Tomcat] Apache Software Foundation, Apache Tomcat – URL http://tomcat.apache.org/ - 06.May.2007

[SWI-Prolog] SWI-Prolog JPL - A Java Interface to Prolog – URL http://www.swi-prolog.org/packages/jpl/java_api/index.html

[Gamma et al, 1995] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides: Design Patterns – Elements of Reusable Object-Oriented Software, Addison-Wesley, 1995 – ISBN 0201633612

[Sun Microsystems Inc. 2002] Sun Microsystems Inc. 2002: Core J2EE Design Patterns – URL http://java.sun.com/blueprints/corej2eepatterns/ - 06.May.2007

[Hans Bergsten, 2003] Hans Bersgten: JavaServer Pages, Third Edition, O'Reilly, 2003 – ISBN 0-596-00563-6

[JUnit.org] JUnit.org – URL http://www.junit.org/index.htm - 06.May.2007

[Bruce W. Perry, 2004] Bruce W. Perry: Java Servlet & JSP Cookbook, First Edition, O'Reilly, 2004 – ISBN 0-596-00572-5

Appendix

Java – Prolog Invocation

Below is an example about the interaction between Java front-end and Prolog back-end. This is an example when the user submits a solution for the predicate declaration. When the Java front-end receives the solution, it will refine the data as the data structure of the String queryString in the source code below.

```
String queryString = ”diagnose_declaration(6, [(sp, 'zinseszins',
[('arg1', '+', 'number', 'Betrag'), ('arg2', '+', 'number', 'Satz'),
('arg3', '+', 'number', 'Zeit'), ('arg4', '-', 'number', 'End')])],
DecMap, ErrorList)”
```

The last two parameters DecMap and ErrorList are the ouput arguments. The type of a parameter which is output or input is provided to the front-end developer by the back-end developer. That means, after making the query queryString to the Prolog back-end, the front-end can retrieve the returned string of DecMap and ErrorList.

```
Query q = new Query(queryString);
HashTable table = q.oneSolution();

String decMap = table.get("DecMap").toString();
String errorList = table.get("ErrorList").toString();
```

The errorList contains all the error information of the submitted solution. There are may be many errors in this errorList, or one error or no error.

```
System.out.println(“errorList: ” + errorList);
```

If there is no error in the solution that means it is a correct solution. And the output of this System.out is the string “[]”.

If there are errors, the output string may look like:

```
'.'(error(penalty(0.1), poslist('.'(pred, '.'(zinseszins, []))),
hint('Die Deklaration für das Prädikat zinseszins hat weniger Argumente
als erforderlich.')), '.'(error(penalty(0.1), poslist('.'(pred,
'.'(zinseszins, []))), hint('Die Deklaration für das Prädikat
zinseszins hat weniger Argumente als erforderlich.')),
'.'(error(penalty(0.1), poslist('.'(pred, '.'(zinseszins, []))),
hint('Die Deklaration für das Prädikat zinseszins hat weniger Argumente
als erforderlich.')), '.'(error(penalty(0.1), poslist('.'(pred,
'.'(zinseszins, []))), hint('Die Deklaration für das Prädikat
zinseszins hat weniger Argumente als erforderlich.')),
'.'(error(penalty(0.3), poslist('.'(text, '.'(3, '.'('.'('Bonus',
'.'('Bonus-Zins', [])), [])))), hint('Das Argument arg3 hat nicht die
richtige Bedeutung. Bitte achten Sie auf die Information im
Aufgabentext.')), [])))))))
```

In this case, the Java front-end has to parse this list of errors, refine each error to a corresponding error object. For example, the first error is:

```
'.'(error(penalty(0.1), poslist('.'(pred, '.'(zinseszins, []))),
hint('Die Deklaration für das Prädikat zinseszins hat weniger Argumente
als erforderlich.'))
```

The information of this error string is retrieved by using some of the methods of the API JPL package.

```
Term errorTerm = jpl.Util.textToTerm(errString);
Term[] errorArray = errorTerm.toTermArray();

String id = errorString.arg(1).arg(1).toString().trim();
double  penalty = Double.parseDouble
                  (errorString.arg(2).arg(1).toString().trim());
String errType = errorString.arg(3).arg(1).
                  toTermArray()[0].toString().trim();
String argNum = errorString.arg(3).arg(1).
                  toTermArray()[1].toString().trim();
String explanation = ((Atom)errorString.arg(4).arg(1)).name();
```

At this step, all the information of this first error are available, so that one can creat an ArgErrorBean object for this error type.

```
ArgErrorBean argError = new ArgErrorBean(id, penalty, errType, argNum,
explanation);
```

And now the error string returned from the Prolog back-end is mapped into error objects which the front-end can continue to use for different purpose.

User Log File Data

In the functional requirements of chapter 3, there is a requirement that all the request information and response information between a user and the server application are written into a text file for each user. And the data in a log file may look like the below:

Fri Nov 24 17:57:10 CET 2006
ACTION: Select exercise Blatt 4 - Aufgabe 4a
Fri Nov 24 17:57:10 CET 2006
INFO: Ein Geldbetrag, der mit einem (jährlichen) konstantem Zinssatz verzinst wird, wächst expoentiell und kann nach der rekursiven Berechnungsvorschrift: Bi = Bi für i=0; Bi= (1+Z)*Bi-1 sonst, ermittelt werden, wobei Bi das Guthaben nach dem i-ten Anlagejahr und Z der Zinsfaktor ist (d.h. für 5% Zinsen ist Z = 0.05). Bilden Sie die angegebene Berechnungsvorschrift in ein rekursives Prolog-Prädikat mit dem Prädikatsschema: zins(+Anlagebetrag, +Zinsfaktor, +Anlagedauer, -Endguthaben) ab.
Fri Nov 24 17:57:19 CET 2006
ACTION: Add new Argument
Fri Nov 24 17:58:06 CET 2006
ACTION: Evaluation
Fri Nov 24 17:58:06 CET 2006
.....

Abbreviations

API	Application Programming Interface
EJB	Enterprise Java Beans
GUI	Graphical User Interface
HTTP	Hyper Text Transfer Protocal
INCOM	Inputkorrektur durch Constraints und Markups
J2EE	Java 2 Enterprise Edition
JSP	Java Server Pages
JSTL	Javaserver pages Standard Tag Library
MVC	Model View Controller
TLD	Tag Library Descriptor
XML	Extensible Markup Language

The accompanying CD

In the CD, you can find the source code for the front-end of the INCOM Application and the WAR file of the application web module. Besides, you can find the INCOM Application Custom Actions and API References PDF file together with the Java API HTML Document of the application and the Bachelor Report in PDF format.

Glossary[27]

access control	The methods by which interactions with resources are limited to collections of users or programs for the purpose of enforcing integrity, confidentiality, or availability constraints.
Applet	A component that typically executes in a Web browser, but can execute in a variety of other applications or devices that support the applet programming model.
business logic	The code that implements the functionality of an application. In the Enterprise JavaBeans model, this logic is implemented by the methods of an enterprise bean.
deployment	The process whereby software is installed into an operational environment.
deployment descriptor	An XML file provided with each module and application that describes how they should be deployed. The deployment descriptor directs a deployment tool to deploy a module or application with specific container options and describes specific configuration requirements that a Deployer must resolve.
distributed application	An application made up of distinct components running in separate runtime environments, usually on different platforms connected via a network. Typical distributed applications are two-tier (client-server), three-tier (client-middleware-server), and multitier (client-multiple middleware-multiple servers).
enterprise bean	A component that implements a business task or business entity and resides in an EJB container; either an entity bean or a session bean.
HTTP	Hypertext Transfer Protocol. The Internet protocol used to fetch hypertext objects from remote hosts. HTTP messages consist of requests from client to server and responses from server to client.
JSP page	A text-based document using fixed template data and JSP elements that describes how to process a request to create a response.
JSP tag library	A collection of custom tags identifying custom actions described via a tag library descriptor and Java classes

[27] All the explanations of these terms in this Glossary are taken from (Inderjeet Singh, et al., 1995)

resource manager	Provides access to a set of shared resources. A resource manager participates in transactions that are externally controlled and coordinated by a transaction manager. A resource manager is typically in a different address space or on a different machine from the clients that access it. Note: An enterprise information system is referred to as resource manager when it is mentioned in the context of resource and transaction management.
Servlet	A Java program that extends the functionality of a Web server, generating dynamic content and interacting with Web clients using a request-response paradigm.
Servlet container	A container that provides the network services over which requests and responses are sent, decodes requests, and formats responses. All Servlet containers must support HTTP as a protocol for requests and responses, but may also support additional request-response protocols, such as HTTPS.
Servlet mapping	Defines an association between a URL pattern and a Servlet. The mapping is used to map requests to Servlets.
session	An object used by a Servlet to track a user's interaction with a Web application across multiple HTTP requests.
session bean	An enterprise bean that is created by a client and that usually exists only for the duration of a single client-server session. A session bean performs operations, such as calculations or accessing a database, for the client. While a session bean may be transactional, it is not recoverable should a system crash occur. Session bean objects either can be stateless or can maintain conversational state across methods and transactions. If a session bean maintains state, then the EJB container manages this state if the object must be removed from memory. However, the session bean object itself must manage its own persistent data.
WAR file	A JAR archive that contains a Web module.
Web application	An application written for the Internet, including those built with Java technologies such as JavaServer Pages and Servlets, as well as those built with non-Java technologies such as CGI and Perl.
XML	eXtensible Markup Language. A markup language that allows you to define the tags (markup) needed to identify the data and text in XML documents. J2EE deployment descriptors are expressed in XML.

Index

Declaration

I declare within the meaning of section 25(4) of the Examination and Study Regulations of the International Degree Course Information Engineering that: this Bachelor Thesis has been completed by myself independently without outside help and only the defined sources and study aids were used. Sections that reflect the thoughts or works of others are made known through the definition of sources.

Hamburg, 30th May 2007
City, Date

Thanh Minh Tu Tran