An Experiment In Model Driven Architecture for e-Enterprise Systems

R. P. Smith and S. Kent

January 14, 2002

Abstract

OMG's Model Driven Architecture [5] demonstrates how a system's specification model can be used within the process of creating supporting software implementations. This article documents the findings of an experiment aimed at determining the extent to which this method of software engineering can be used within the domain of e-Enterprise systems.

Contents

List of Figures

1 Introduction

This paper reports the initial results from an experiment into the utilization of OMG's Model Driven Architecture (MDA) [5] as methodology for the implementation of e-enterprise systems.

1.1 Model Driven Architecture

The main tenet of MDA is to abstract away from particular implementation technologies (platforms) by modelling systems in a platform independent way and automating the process of developing implementations on particular platforms from those models. It is intended that a Platform Independent Model (PIM) is realized through the use of a modelling language such as UML [6] and exists to document a technology independent architecture for a specific computing process at a high level of abstraction. Since the PIM is platform independent no specific implementation technology is specified. Mappings from these PIMs to Platform Specific Models (PSMs) are documented where a specific PSM models the architecture required for software deployment within a specific implementation technology.

The advantages of following the MDA approach fall into four main categories, described by Jishnu Mukerji [4]: -

Firstly, MDA increases the scope for application portability through the reuse of the system design. This reduces the cost of development and the complexity of the application management processes.

The use of multiple PSM implementing a system specified within a shared PIM ensures the realization of identical business functions over differing implementation technologies. This is known as cross platform interoperability.

Productivity also increases when using familiar concepts which can be aided through the use of tools build upon a particular PIM to PSM relationship. MDA creates the possibility for automated services aiding the development and implementation of software systems given detailed PIM to PSM transformations.

Finally, MDA eases the re-targeting of existing application code to new technologies if a related PSM exists stemming from the original system PIM.

1.2 Application Domain

Figure 1 depicts an overview of the application domain targeted by the experiment.

A two tier architecture exists. Those function which are accessible to the system users are initiated through client machines remote from the server side processing. Application software developed within this domain requires extra functionality to overcome the complexities arising from a this two tier programming architecture. For example, the diagram illustrates the system application logic resident on both the client and server sides of the interaction, thereby creating the possibility for a distributed implementation architecture. This requires some shared knowledge between the distributed implementations to fa-

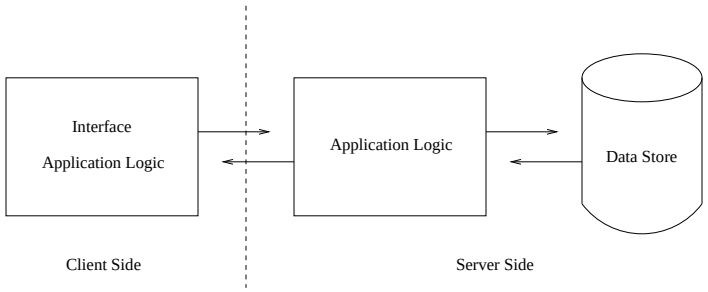

Figure 1: The Application Domain

cilitate the breakdown of application functionality and communication between the two.

System security issues create further complexities when working within a distributed programming model that differ from those attributed to traditional isolated single user computing environment. For example, the most common piece of extra functionality employed within existing e-enterprise systems is a method of client identification since the server side system has the potential to be invoked by many different clients at any one time. The identification process is used to validate an incoming client's right to perform the requested actions with respect to the current state attributed to the calling client. This is of highest importance in order to maintain secure, reliable and trusted services within a networked environment.

It is important to note however that while the extra functionality is required to support a distributed programming model, particular e-enterprise system implementations have similarities within their implementation architectures as a result. This fact can then be exploited within the development of supporting e-enterprise system PIMs since similar application structures and common processes will apply.

1.3 Experiment Overview

The experiment is concerned with the specification and implementation of a test application. The application chosen for development was that of an online auction system requiring the following functionality: -

- The registration of new system users,
- The posting of lots for auction by registered users,
- Bidding on lot items by registered users.
- Viewing lots currently for auction.

This type of application was chosen because it encompasses many of the features common to existing e-enterprise systems. For example, a user validation

process is required to ensure secure access to system functions specified as being only available to registered system users, and data repository functionality is needed to provide a persistent data store of information.

To conform to the MDA approach, the experiment was initiated through the development of a test system PIM. It is important to note that while this model is platform independent in the respect that no implementation technology constraints are specified within the model structure, it is domain specific because it contains support for e-enterprise system services modelling server side representations of remote client interactions. This PIM is described within section 2.

A PSM consisting of the architecture required for the implementing of the test system using a specific set of technologies was created in parallel to the PIM. By implementing the two models concurrently, the PIM architecture could be used within the retaliation of the PSM to create two complementing models with inherent similarities. These similarities could be exploited to facilitate the extraction of PIM to PSM mappings. The PSM is described within section 3.

Section 4 documents the mapping from PIM to PSM, describing the standardizations that were required to be adopted in order to facilitate the realization of generic transformations.

1.4 Goals

Figure 2 illustrates an overview of the experiment structure in which the top and bottom entities represent the PIM and PSM respectively.

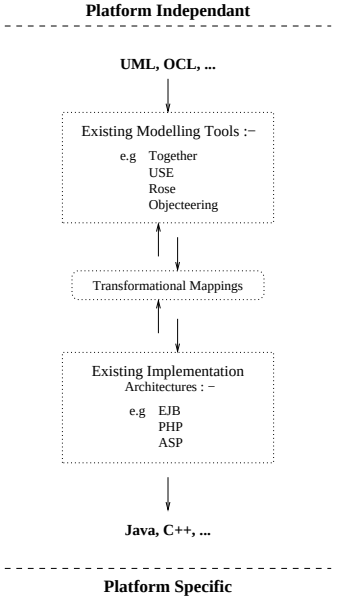

Figure 2: Domain Structure

The transformations appearing in the center of the diagram represents the experiment objective. As well as creating workable models for the platform specific and a platform independent e-enterprise system architecture, the experiment is aimed at an investigation into the extent to which transformational support between the two models can be realized through the utilization of elements held within the PIM. Therefore, the experiment results will consist of a documented set of PIM to PSM transformations with indications to where extra information is required to be present within the PIM specification to facilitate their use. Dependant on the ability to automate the process, the transformational information could become the building block onto which implementations of supporting MDA tool sets for e-enterprise system generation can be built.

Some existing research exists addressing the issues surrounding code generation from UML specifications by people such as Arief and Speirs [1] and Valdeón, Morillo and Bonilla [3]. However, the translational systems proposed within these documents currently work at a high level of abstraction, providing simulations of UML models. Within this document, we aim to present a framework through which complete executable e-enterprise applications can be created from a supporting PIM specification using the MDA.

2 Platform Independent Model

2.1 Modelling Language

The Platform Independent Model (PIM) has been developed using a subset of the Unified Modelling Language (UML) [6]. This subset is the subset supported by the USE tool [7].

2.2 The Test System Model

The elements of UML used are illustrated within the test system model [10]. Consider figure 3, a diagrammatical representation of the PIM created for the test system, named `JavaAuction`.

It is important to note that this PIM contains those elements specifically required to support the auction system application, as well as those elements required to support its implementation within a distributed environment. The separation between these two elements is described in section 2.3.

Within this model, five classes of object are present. The `Lot`, `Bid` and `User` classes represent the information required to remain persistent, thereby constituting the application's data repository. The `System` class is used as a container, with associations to both the persistent data classes and a further class named `WebSession`. A `WebSession` object in the system representation for a client making contact with the server. It contains seven operations that make up the system functionality from the clients perspective. This object can be thought of as the gateway through which users remote to the server side system interact with the server. OCL pre and post conditions are added to the

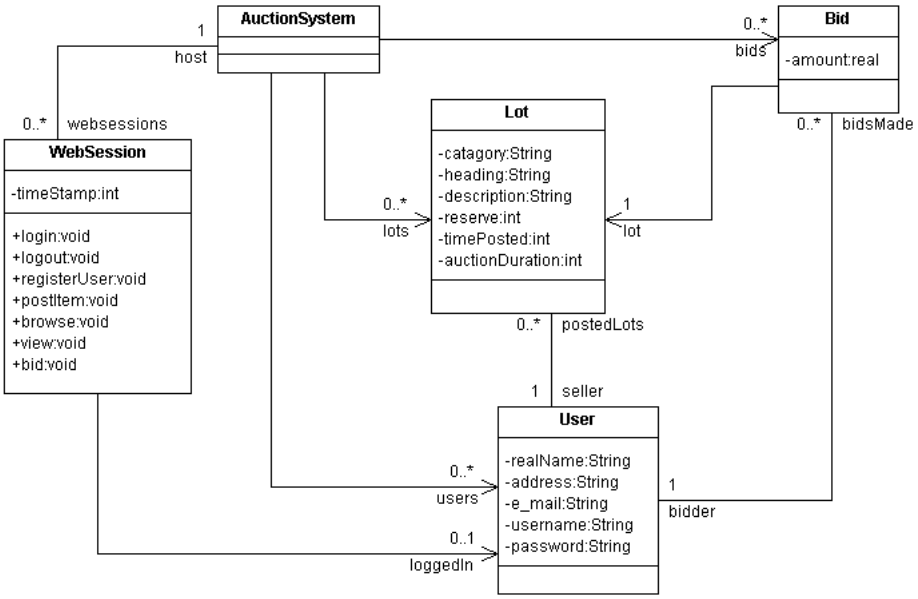

Figure 3: JavaAuction PIM

`WebSession` operation specifications, referencing the information held within the model to dictate any conditions on the operation's execution, specify any constraints on the parameters that are passed, and describe the resultant model state.

All clients are allocated a `WebSession` whether they are currently logged in or out. A client retains the same `WebSession` for the duration of its interaction with the server, which may involve many separate messages sent between the two parties. This is the method used by the majority of web based applications.

For the test system application, some notion of client state is required, where a client can be either logged in or logged out at any particular time. Therefore, `WebSession` objects have the ability to be associated with a single `User` object that stores information specific to an individual system user. Each client registered with the system will therefore have a `User` object resident within the model containing their specific details, including information on how the client identifies itself to the system when performing a login action. When a client is logged in, the `WebSession` has two data repositories at its deposal, those items related to the particular client's `User` object, and the system as a whole. This separation can be seen within the `WebSession`'s operation specification. For Example, the `registerUser` operation within the `WebSession` class is specified within the `WebSession` class as follows: -

```
registerUser(realName :  String, address :  String,
             e_mail :  String, username :  String,
             password :  String, repassword:  String)
```

The operation has the following pre and post condition specified: -

```
pre PRE_Has_No_Logged_In_User:  not(self.loggedin.isDefined())
pre PRE_Parameters_Defined:  realName.isDefined()
    and address.isDefined() and email.isDefined()
    and username.isDefined() and password.isDefined()
    and repassword.isDefined()
pre PRE_Password_Fields_Match:  password = repassword
pre PRE_Username_Is_Unique:  (self.host.users.select(u |
    u.username = username)-> size = 0
post POST_New_User_In_System:  (self.host.users.select(u |
    u.username = username and u.realName = realName
    and u.email = email and u.password = password))->
    size = 1
```

These expressions interrogate both information which is global to the model as a whole, and the information related specifically to the client's `User` object this `WebSession` is the representation for. For example, global information is obtained by traversing the model via the `host System` object: `self.host. ...` as in `POST_New_User_In_System`, and the local client information via the `WebSession`'s related `loggedIn User` object: `self.loggedIn. ...` as in `pre PRE_Has_No_Logged_In_User`

A USE tool [7] model listing for the test system can be found at [9]

2.3 Modelling Pattern

The test system example illustrates the need for a particular pattern of modelling at the PIM level to facilitate the mapping process to the PSM. By structuring all PIMs in the same way, generic transformations can be developed between elements resident within the PIMs and the implementation structures required within the PSMs to support the specification. This pattern is illustrated within figure 4.

Four classes of object are defined: -

System As seen in the test system PIM, all objects resident within the model are required to be associated with the `System` object. This therefore acts as a container for all the data stored. From the `System`'s perspective, all information

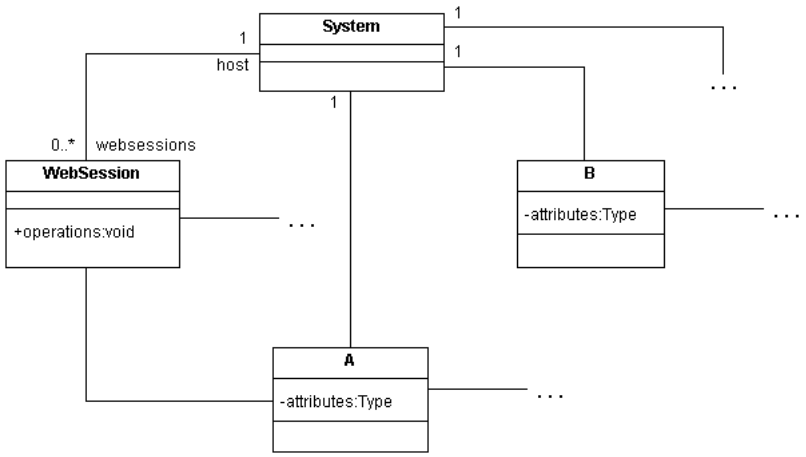

Figure 4: PIM Pattern

within the model appears as collections of persistent objects, grouped together within their object classes.

WebSession As described before, the `WebSession` class exists as the server side representation of a remote client interaction. An object of this class is allocated to a client independent of any state that may be associated with it. For example, in the test system PIM, both logged in and logged out clients have a related `WebSession` object. It is the information and object references attached to this object that determines client state.
System operations are specified within this class representing those functions that can be performed by the system, initiated through a client interaction.
From the diagram, it can be seen that the `WebSession` class has access to the `System`'s object collections by traversing their shared association to its related `host`, and access to any other classes to which it is related by traversing their shared associations directly.

A Class `A` represents those classes which are directly related to the `WebSession` class. Individual client representations can be tailored through the use of attached `A` object references.

B Other classes can exist within the model which are not associated directly to the `WebSession` class. However, objects of this type can be related to other classes within the model, some of which may have the ability to be related directly.

A specialization of this modelling pattern is where a two state client system is required thereby allowing a client to be either logged in or logged out at

any one time. The `WebSession` class specification must include operations that are used to support the client initiated transition of state. Consider figure 5, illustrating how the general modelling pattern is adapted to support the new pattern. As we have seen before, the test system uses this approach.

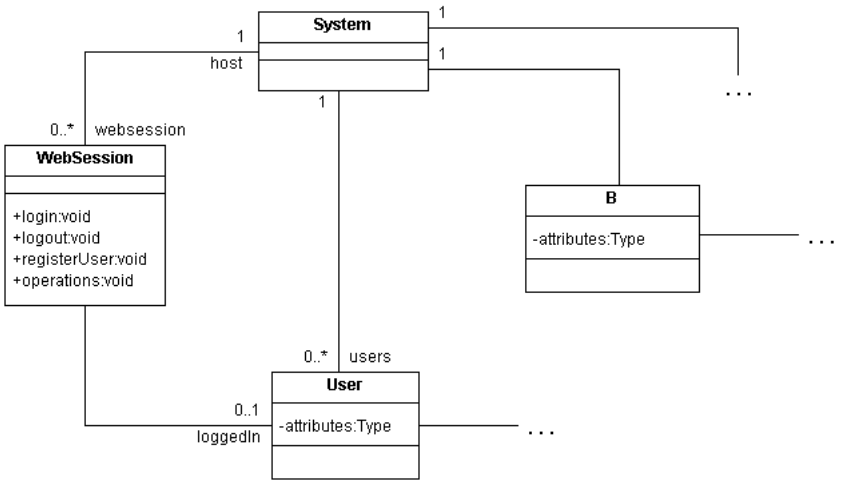

Figure 5: Logged In/Out PIM Pattern

Three operations must be included in the `WebSession` class specification: `login`, `logout` and `registerUser`, where `registerUser` provides a method of inputting client specific information into the server side data repository. The specific parameters the `registerUser` operation takes are application dependant but must include the data which will be interrogated by the `login` operation. For the `registerUser` operation within the test system model, it was deemed relevant to include a real name, address and e-mail, as well as a unique username and password for client identification.

A new class is named `User` is introduced that is used by the `registerUser` operation to create client specific `User` objects. These objects contain individual data stores relating to client details, populated by the parameters passed to the `registerUser` operation. The `User` objects can also be referenced by any other objects within the system that are specifically related to the client for which the `User` object is the system side representation. For example, within the test system, a `Lot` object holds a reference to the particular `User` object who's client was responsible for posting it.

The `WebSession` class now shares an association with the `User` class. When a client logs in, its `WebSession` stores a reference to the clients related `User` object. Therefore, from the WebSession's perspective, information within the system is organized into two different categories: That which is specific to the particular calling client, through the interrogation of any related `User` object, and that which is global system to the system as a whole, referenced through

the host `System`.

Note that this is a simplification of the actual mechanism required within specific PSMs. For example, the `WebSession` class within the test system PSM accesses the global data repository by contacting an EJB server with the appropriate JNDI lookup name for the resource it requires.

3 Platform Specific Model

3.1 Implementation Architectures

Existing application architectures can be used to provide support for the extra functionality required from two tiered client / server application. This includes a method of facilitating the client / server communication, the realization of actual server side application logic and a method of providing a persistent data repository.

All of the architectures used for the development of the test system were Java based, chosen to maximize the potential target hardware platform base, using a web server to facilitate network transport. These architectures include JSP and Java Servlets, with an Enterprise Java Bean Server providing database functionality.

3.2 The Test System Implementation

The specific use to which the existing application architectures are put can be illustrated by considering the implementation of the test system [11]. Consider figure 6, which depicts an overview of the test system architecture.

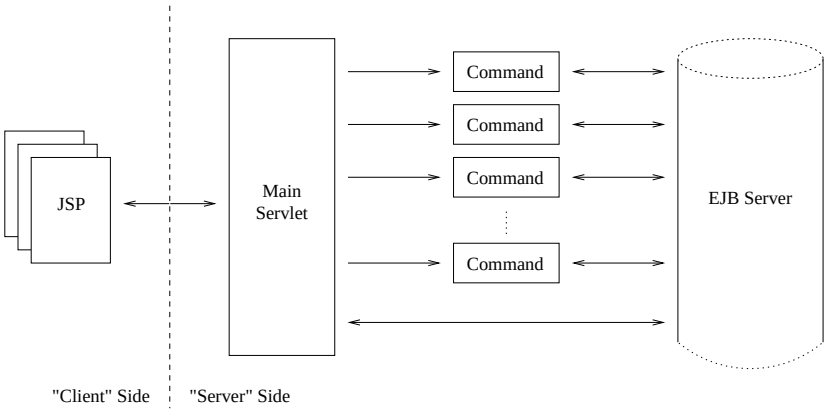

Figure 6: Test System Structure

The system is split into two sections referred to as the *Client* and *Server* sides. The output generated from a generic set of JSPs exists on the client side, acting as a thin client providing access to the application logic resident on the

server. The server side system consists of a single Java Servlet which, on inspection of incoming client requests, delegates any action required. This is achieved through the utilization of individual command class specializations containing supporting functionality for the operations specified within the `JavaAuction` PIM. An EJB server acts as the system's data repository maintaining a persistent data store utilized by both the command specializations and main servlet. These elements will now be discussed in detail, with respect to their individual functionality and allocated responsibilities.

3.2.1 Contacting the Server

Both access to system functionality and access to the different sections of the website are controlled by the server side system. Therefore, when a client first accesses the server using a web browser, the URL of the servlet is opened. In this case:-

```
http://<SERVER_NAME>/auction/servlet/JavaAuction
```

On receiving a request of this type, with no related attributes being passed, the servlet forwards the calling client to the application front page. Complete control is given to the server side application logic in order to give the client side HTML code generated from the JSPs true thin client functionality. One advantage of this approach is in the simplification of the transformation definitions from the model to implementation, using a centralized target area for the system's application logic.

3.2.2 JSP Organization

The different sections of the website are generated on the client side by using groups of generic JSPs. One group of pages appears per system operation specified within the PIM. The allocation and content of these groups is dependant on the type of operation the group relates to. Two types of system operation appear, each requiring different pages within the supporting JSP group. For the test system, a group is referred to as the operation's `Page Set`.

The following sections describe each operation type with the corresponding contents of the `Page Set` required to support it: -

Action / Response An `Action / Response` operation is one which takes a set of parameters and updates the server side data repository to return an outcome of either success or failure. The JSP pages required to support an operation of this type are an `Action Holder`, the page that contains the operation initiator HTML form element, and the `Response Page`, indicating the operation's success. Therefore, an `Action / Response Page Set` consists of these two pages.

Request / Reply A `Request / Reply` operation takes a set of parameters and returns a formatted reply page made up of matching data extracted from the server side data repository. For this operation type, the JSP pages required are a `Request Holder`, again containing the operation initiator HTML form and a header and footer JSP file to be included in the generated reply. Therefore, a `Request Holder Page Set` consists of these three pages.

3.2.3 Facilitating Dynamic Content

Generic JSP `Page Set`s are used instead of the servlet generating entire web pages on the fly in order to cut down the amount of processing the server side system is required to perform. While dynamic content can be achieved by the servlet generating the entire client side interface, this could become complicated and time consuming. The generic pages can still provide dynamic content by utilizing a feature of the JSP technology that allows attributes to be set within the outgoing JSP request, tailoring the target page to the clients requirements. For example, the following JSP code is resident within the `browseresultsheader.jsp` file within the test system implementation: -

```
<FONT face="arial" size=5>
    <B>Browsing
    <%= (String)request.getAttribute("category") %>
    </B>
</FONT>
```

The attribute, `category`, is set by the application logic before the page is invoked, thereby tailoring this generic page to the particular clients' requirements who made the original request. For example, if the requested category was named *Graphics Card* the `Browse` operation's reply page would appear as in figure 7, (page breakdown shown for clarity)

This is used for all instances where small items of dynamic data are required. For situations where larger amounts of data is required to be displayed, such as in the reply from a `Request / Reply` operation as shown in the main body of the example in figure 7, the servlet generates the information directly.

3.2.4 Initiating System Functions

As described above, the main servlet deals with access to both the application operations specified within the system PIM and access to the different section of the site. Therefore, the server side system can perform two types of function: page linking and application command initiation. Within the test system, all application operations are initiated through a client submitting a form resident on a JSP page. This action sends a group of attributes to the server side system, one per form input field. A hidden function attribute, named `command`, is included in order to specify the exact nature of the request. For example,

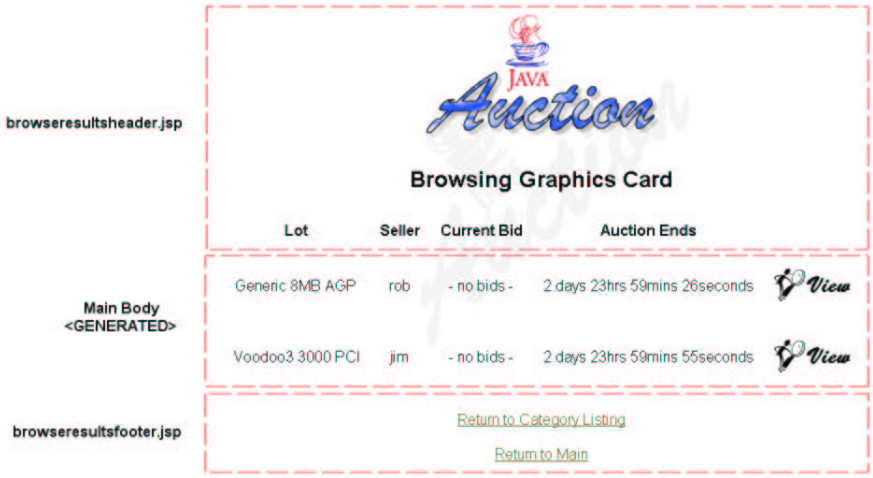

Figure 7: Browsing the Graphics Card Category

the following HTML code is used to initiate the `RegisterUser` operation, (JSP attributes and formatting omitted for clarity): -

```
<FORM action="/auction/servlet/JavaAuction" method=POST>
<input type=hidden name=command value=registeruser>
Name:</TD><TD><input type=text size=20 name=realname>
Address:</TD><TD><input type=text size=20 name=address>
e-mail:</TD><TD><input type=text size=20 name=e_mail>
Username:</TD><TD><input type=text size=20 name=username>
Password:</TD><TD><input type=password size=20 name=password>
Retype Password:  <input type=password size=20 name=repassword>
<input type=submit value="Register">
</FORM>
```

For page linking, direct attribute passing is used in place of a form. This is because only one attribute needs to be sent. Typically, a page linking function is realized with the use of a `<A>` HTML tag. For example, the following HTML code initiates a `link` function on the server, with the client asking to be transported to the `RegisterUser` section of the site, : -

```
<A href="/auction/servlet/JavaAuction?link=registeruser">
    Register User Section
</A>
```

It is important to note that no JSP is allowed to link directly to another page. All page linking requests must be made through the server side system via the utilization of a link, shown above.

The aim of the JSP allocation and implementation method was to create a framework in which a JSP developer could be presented with a document dictating the structure and contents of all the JSP pages required to generate the client side interface, less any aesthetic properties such as page layout or graphics. From this document, all the JSPs required could be created directly. It would therefore be the responsibility of the PIM to PSM mappings to generate the contents of this document in order to facilitate the creation of the client side GUI. This is possible because all application logic is resident on the server.

3.2.5 Server Side Processing

Moving to the server side, the main servlet has its functionality split into three sections named `Session Management`, `Page Access Control` and `Command Access Control`. The access control mechanisms currently supported within the PSM simply check whether the incoming request has been initiated by a client in the correct state. This state is that which is specified by the OCL condition related to login status attached to the corresponding operation within the PIM. For applications where many privilege levels are required, more complex access control checks would be needed but these could still be implemented within the same overall structure as described within the reagent sections below.

Session Management Since all application logic is resident on the server, a method of identifying which client is accessing the server is required in order to control access to the different sections of the website and application commands. This is important because both page and command access can be specified within the PIM to be dependant on a client's login status and so the correct identification of the calling client is required.

The method used by the test system is the setting of a JSP session attribute within the clients request, in the same way used for facilitating dynamic content, described before. The attribute is set to a value unique to the clients server side `WebSession`, described below. This enables the specific server side `WebSession` object to be found as and when the related client contacts the system thereby identifying the calling client through the interrogation of this attribute.

When a client calls the server, the `Session Management` method is invoked. This method deals with the client identification process, involving the lookup of the calling clients related `WebSession` object, if one exists, or creating one if it is the first time the client has made contact since opening his web browser. After creation, the JSP session attribute relating to `WebSession` id is set in the calling clients request in order to ensure the same `WebSession` object is retrieved on its return.

This method also interrogates any function attribute attributed with the incoming communication to establish whether a `link` or `command` function is required. If the calling client it requesting a `link` function, the `Page Access`

`Control` method is called. Similarly, if a `command` function is requested, the `Command Access Control` method is called.

Page Access Control The `Page Access Control` method contains a structure that interrogates an attribute named `link` passed within the clients incoming request. Therefore, there is an entry within this structure for each section of site. One section appears per system operation, where the contents of each section contains `link` matches for the particular operation `Page Set`. For example, the following code appears within the test system for the `RegisterUser` operation: -

```
// Register User
// Action Holder
if(link.equals("registeruser")){
    if(websession.getUser() == null){
        RequestDispatcher rd =
          sc.getRequestDispatcher("/registeruser.jsp");
        rd.forward(request, response);
    else{
        RequestDispatcher rd =
          sc.getRequestDispatcher("/errorLoggedIn.html");
        rd.forward(request, response);
    }
}
// Response Page
else if(link.equals("userregistered")){
    if(websession.getUser() == null){
        RequestDispatcher rd =
          sc.getRequestDispatcher("/userregistered.jsp");
        rd.forward(request, response);
    else{
        RequestDispatcher rd =
          sc.getRequestDispatcher("/errorLoggedIn.html");
        rd.forward(request, response);
    }
}
```

Notice the interrogation of the clients related `WebSession` object, required to establish the incoming client's right to access this section of the site. Entries appear for both the `Action Holder` page and the `Response Page` indicating the operations success, (failure is discussed bellow)

Command Access Control With a similar structure architecture to the `Page Access Control` implementation, the `Command Access Control` method is called by a client wishing to execute a system operation, as specified within

the system PIM. This is achieved by passing an attribute named `command` to the servlet set to the particular operation required. This attribute is compared with a string representation of each system operation. Again, the client's associated `WebSession` is interrogated to establish the right to perform the action.

For example, the structure governing access to the `RegisterUser` operation is implemented as follows: -

```
// Register User
if(command.equals("registeruser")){
    if(websession.getUser() == null){
        RegisterUserCommand register =
        RegisterUserCommand(
          websession, request, response, this);
    }
    else{
        ServletContext sc = getServletContext();
        RequestDispatcher rd =
          sc.getRequestDispatcher("/errorLoggedIn.html");
        rd.forward(request, response);
    }
}
```

System operations are implemented by using individual `command` objects, in this case, `RegisterUserCommand`. On receiving a command operation from a client in the correct logged in state, an object of the correct command type is created. This object contains all the functionality required to perform the operation and runs when the class constructor is called, i.e. as it is created. Each command performs its functionality split into four groups: Extract the required attributes from the incoming request; perform any precondition checks; perform the action application logic; perform any post condition checks. This pattern will be discussed in the template described in section 3.3.

For both system operation types, failure is dealt with in the same way. This involved the client being returned to the original command initiator page, after first setting a JSP attribute in the page request used to indicate the nature of the error that had occurred. This attribute is used to format the page with an appropriate error message. For example, the page which initiates the `RegisterUser` operation has the following extract of code within the HTML form element: -

```
<!-- Username Field -->
<%if((String)(request.getAttribute("paramusernameblank"))
        == "true"){%>
    <TR><TD><FONT size=1 face=arial color=red>
        You must type a username</FONT></TD></TR>
<%}
if((String)(request.getAttribute("usernameexists"))
        == "true"){%>
    <TR><TD><FONT size=1 face=arial color=red>
        Username already in use</FONT></TD></TR>
<%}%>
<TR><TD align=right>Username:</TD>
<TD><input type=text size=20 name=username></TD></TR>
```

This page appears originally as in figure 8.

Figure 8: The Register User Action Holder Page

If the `username` field was left blank, the `RegisterUserCommand` would set an attribute named `paramusernameblank` to `true` before returning the client back to the original command initiator JSP. The page would then appear as shown in figure 9.

Data repository functions are provided within the test system implementation through the use of an Enterprise Java Bean (EJB) server. EJB's provide a method of creating a persistent virtual database within a Java environment. They come in two types, `Session` beans and `Entity` beans.

`Entity` beans are Java objects that remain persistent within the system, surviving crashes and system shutdowns. All objects of the same `Entity` bean

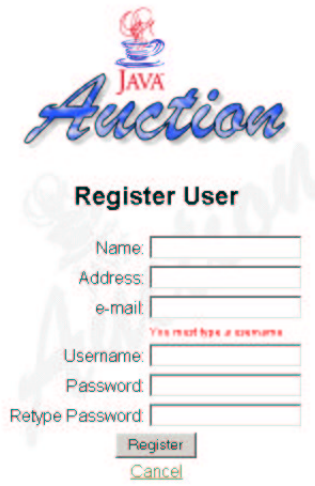

Figure 9: The Register User Action Holder Page (After Error)

class make up an entire virtual database table. An individual `Entity` bean object is a single row within that table, an attribute of which being a field within that row.

`Session` beans contain business logic that interact with the Entity bean classes, but since the use of separate command object specializations already separates the system functionality from the data storage, the test system implementation only uses Entity beans.

The EJB server deals with the complexities of providing transactional support functionality leaving the developer to simply implement the specific classes of information that are required to be stored in the form on Entity beans implementations.

Within the test system, the `User`, `Lot` and `Bid` classes are all realized within Entity EJB implementations. The `WebSession` class is also implemented as an entity bean, simply because the EJB server provides a simple mechanism for storing `WebSession` objects between client interactions. This does not mean that sessions are recoverable after system shutdown since JSP attributes are themselves reset when the web server restarts. A clients request will therefore no longer store the id of it's server side `WebSession` representation. Other mechanisms for storing `WebSession`s can be used in place of the EJB server, such as declaring a static collection of `WebSession`s in the main servlet which is shared by all instances created on the server.

Both the main servlet and command implementations access the EJB server directly. It is important to note that the associations resident within the `JavaAuction` PIM between the `System` container and the rest of the system as a whole is realized within the implementation through the use of this EJB server. Therefore, no specific `System` class is required.

A full program listing of the test system implementation can be found at [9].

3.3 Implementation Template

The use of generic transformations mapping from a PIM conforming to a standardized modelling pattern creates the need for a corresponding standardized implementation template [12]. PSMs are created by populating the standard template using information contained within the PIM. This template can be drawn from the test system implementation, becoming a skeleton implementation containing the functionality required to support programming within a two tier application environment, and a set of rules governing the organization and structure of the implementation code.

The contents of the template is as follows: -

- A skeleton Java Servlet with rules governing the implementation of the access control structures
- A standard command implementation pattern

The following sections describe the content of each of these elements

3.3.1 Skeleton Servlet

Within the test system's PSM, described in the previous section, a single Java Servlet deals with requests from calling remote client machines. This servlet has implementation split into three sections used for `Session Management`, `Page Access Control` and `Command Access Control`. These sections are implemented as individual methods within the Servlet implementation class.

Since this experiment is interested in those applications where a client's logged in status is an issue, the session management method within the skeleton servlet can become part of the generic Implementation Template.

The contents of the `Page Access Control` and `Command Access Control` structures are split into sections for each system operation specified by the PIM.

Within the `Page Access Control` structure, the specific implementation of these sections is dependant on both the clients right to perform the action in its current logged in state and the functions type, being either `Action / Response` or `Request / Reply`, as described before. This categorization by type is required in order to accommodate the different command `Page Set`'s associated with each of them. Therefore, `Action / Response` commands are accommodated with a `Page Access Control` section similar to the following Java code, where the checks made on the `WebSession`'s related `User` object governing login status are changed dependant on the particular system operation: -

```
// An Action / Response System Operation
// Action Holder
if(link.equals("myFunctionHolder")){
    // If the client is logged in...
    if(websession.getUser() != null){
        RequestDispatcher rd =
          sc.getRequestDispatcher("/myFunctionHolderPage.jsp");
        rd.forward(request, response);
    // If he is not...
    else{
        RequestDispatcher rd =
          sc.getRequestDispatcher("/myErrorPage.html");
        rd.forward(request, response);
    }
}
// Response Page
else if(link.equals("myFunctionResonse")){
    // If the client is logged in...
    if(websession.getUser() != null){
        RequestDispatcher rd =
          sc.getRequestDispatcher("/myFunctionResponse.jsp");
        rd.forward(request, response);
    // If he is not...
    else{
        RequestDispatcher rd =
          sc.getRequestDispatcher("/myErrorPage.html");
        rd.forward(request, response);
    }
}
```

`Request / Reply` commands will therefore require the following section implementation to support their three page `Page Set`: -

```
// A Request / Reply System Operation
// Request Holder
if(link.equals("myFunctionHolder")){
    // If the client is logged in...
    if(websession.getUser() == null){
        RequestDispatcher rd =
          sc.getRequestDispatcher("/myFunctionHolderPage.jsp");
        rd.forward(request, response);
    // If he is not...
    else{
        RequestDispatcher rd =
          sc.getRequestDispatcher("/myErrorPage.html");
        rd.forward(request, response);
    }
}
// Reply Header
else if(link.equals("myFunctionReplyHeader")){
    // If the client is logged in...
    if(websession.getUser() == null){
        RequestDispatcher rd =
          sc.getRequestDispatcher("/myFunctionReplyHeader.jsp");
        rd.forward(request, response);
    // If he is not...
    else{
        RequestDispatcher rd =
          sc.getRequestDispatcher("/myErrorPage.html");
        rd.forward(request, response);
    }
}
// Reply Footer
else if(link.equals("myFunctionReplyFooter")){
    // If the client is logged in...
    if(websession.getUser() == null){
        RequestDispatcher rd =
          sc.getRequestDispatcher("/myFunctionReplyFooter.jsp");
        rd.forward(request, response);
    // If he is not...
    else{
        RequestDispatcher rd =
          sc.getRequestDispatcher("/myErrorPage.html");
        rd.forward(request, response);
    }
}
```

Within the `Command Access Control` structure architecture, each section entry follows the same basic pattern because all commands are initiated in the

same way whether they are `Action / Response` or `Request / Reply`. However, the implementation of each entry is still dependant on state that clients are allowed to be in in order to perform the related system command. For example, the following section entry would be required for a command which is only available clients currently logged in: -

```
// Action Holder
else if(command.equals("myCommand")){
    // If the client is logged in...
    if(websession.getUser() != null){
        MyCommand myCommand =
          new MyCommand(websession, request, response, this);
    }
    // If he is not...
    else{
        RequestDispatcher rd =
          sc.getRequestDispatcher("/myErrorPage.html");
        rd.forward(request, response);
    }
}
```

It becomes evident that implementing these sections as lists of `if` statements generates the need for large amounts of supporting code for each system operation. For example, a single `Request / Reply` operation requires a `Page Access Control` section amounting to over thirty lines without taking into account any code comments. Experiments into breaking these structures down into smaller more manageable object oriented solutions were found to complicate both the implementation of the servlet and the identification of mappings from the PIM. However, the use of this standard template does enforce a strict pattern to the structure of the implementation and so an automated code generation process may be the best route to create the PSM utilizing the mappings documented within section 4.

3.3.2 Command Pattern

The test system implementation has each system operation implemented within a separate command class. To enforce a standardized implementation approach for all command classes, the implementation template provides a `Command` super class which each system command class must extend.

Consider the following extract from the `Command` super class constructor: -

```
public Command(WebSession websession,
        String failPage,
        HttpServletRequest request,
        HttpServletResponse response,
        HttpServlet caller){

    try{
        // Get Page Attribute(s)
        getAttributes(websession, request, response);

        // Check Pre Condition(s)
        checkPreConditions(websession, request, response);

        // Execute Command
        execute(websession, request, response);

        // Check Post Condition(s)
        checkPostConditions(websession, request, response);

    }
    catch(PreConditionException preE){
        // Forward to Fail Page
        forwardTo(failPage, request, response, caller);
    catch(PostConditionException postE)
    catch(Exception anyotherE){
        anyotherE.printStackTrace(System.out);
    }
}
```

The four methods called within this extract are all declared abstract. To create a particular system command specialization simply involves the implementation of these four methods within the extending class.

The super class also provides helper functions dealing with EJB lookup and text formatting to aid the developers code implementation. These are: -

forward(...) Used to forward the calling client onto another web page

include(...) Used to include a page within a generated response

getHomeInterface(...) Returns a reference to the EJB related to a specified JNDI path.

outputText(...) Outputs a given String text parameter to a given `PrintWriter` object, converting all line breaks into explicit `<BR>` HTML tags.

Pre and Post condition failure is dealt with through the raising of pre defined exceptions, `PreConditionException` and `PostConditionException`. When an exception is thrown, the corresponding handler catches the exception and forwards the client onto the commands failure page. As described before, the failure page within the test system command implementations is the original JSP command initiator. The command's `checkPreConditions(...)` method therefore sets any JSP attributes required to format the page, indicating the error to the client as seen before, before throwing the exception.

A full implementation template code listing can be found at [9]

4 Platform Independent to Platform Specific Mapping

In order to enable the generation of implementation code from a higher level UML design specification, a documented set of identifiable mappings are required between the information that occurs within the model and the corresponding data within the generated system.

Below is a table that lists the elements that appear within the test system's PIM and the mapping used to create the corresponding PSM element.

PIM	**Mapping**	**PSM**
Class Names Class Attributes Class Associations	Enterprise Java Bean Implementation	EJB classes
WebSession Class Operations WebSession Operation Pre Conditions WebSession Operation Post Conditions	Command Implementation	Command classes
WebSession Operations WebSession Operation Pre Conditions	JSP Allocation and Implementation	JSPs
WebSession Operation Pre Conditions	Page Access Control Implementation	Servlet class `Page Access Control` method
WebSession Operation Pre Conditions	Command Access Control Implementation	Servlet class `Command Access Control` method

The following sections take each mapping in turn and investigates how the information held within the PIM can be exploited within the structure of the implementation template, described in the previous section, to facilitate the generation of system code that implements the specified functionality.

4.1 Enterprise Java Bean Implementation

All object classes appearing within the system's PIM become EJB classes within the PSM with the exception of the `System` class. For example, five different classes of object appear in the test system PIM: -

- `System`
- `WebSession`
- `User`
- `Lot`
- `Bid`

The `WebSession`, `User`, `Lot` and `Bid` classes are implemented as individual EJB classes.

The PIM `System` class is not realized as a specific EJB implementation but becomes the EJB server that is accessed by the servlet and command implementations. The `System` class within the PIM is used to enable the separation of local to global information repositories to be formalized within the structure of the `WebSession` operation's OCL expressions. As discussed before, this is a simplification which can map down to the actual system implementation in many ways and in this case appears in the form of an EJB server.

The attributes that each implemented EJB class contains maps directly from the attributes held within the corresponding class in the PIM specification, and any associations the class has with other classes within the model.

For example, consider the following extract from the test system PIM relating to the `Lot` class: -

```
class Lot
  attributes
  category :  String
  description :  String
  reserve :  Integer
  timePosted :  Integer
  auctionDuration :  Integer
end
```

There is also one association specification emanating from the `Lot` class. This association has the `User` as the target with cardinality 1 and given the role name `seller`.

The corresponding EJB bean class implementation and supporting home and remote interfaces can be seen to relate directly from this data held within the PIM. Consider the following extract from the `Lot` EJB implementation: -

```
...
public String category;
public String heading;
public String description;
public Integer reserve;
public Long timePosted;
public Integer auctionDuration;
public User seller;
public Integer id;
...
```

The attributes appearing in the PIM class specification correspond exactly to those appearing in the supporting code. The `seller` attribute maps from information held within the `Lot` / `User` association using the target role name and class type.

It is important to note that the cardinality of the association is also important if it is not simply 1. In that case, the attribute appearing within the EJB implementation would be required to be a collection type such as an array or `Vector`. In a real world situation, however, handling persistent collections raises many questions surrounding efficiency and so a more complicated approach implementing functions such as caching and pre-fetches may be more appropriate in certain situations. This document is aimed at investigating the existence of PIM to PSM mapping and not their optimization.

However, one noticeable difference exists in the form of an extra attribute named `id`. This is because all entity bean classes require one of their attributes to be declared the beans primary key. Therefore, the value the chosen attribute contains must be unique for all entity bean instances of the same class. Using only information contained within the PIM model, it is impossible to ascertain which class attributes will contain unique data. Therefore, the pattern adopted within the test system PSM was to use an extra integer `id` attribute within each EJB class acting as the beans primary key.

Using this approach, the PIM contains all the information required to allow automatic generation of the EJB implementations.

4.2 Command Implementation

There is a direct mapping between the operations appearing within the `WebSession` class specified in the PIM, and the actual command specialization that are implemented within the system.

Consider this extract from the `WebSession` class specification in the test system PIM: -

```
class WebSession
  ...
  operations
    login(username :  String, password :  String)
    logout()
    registerUser(realName :  String, address :  String,
                 e_mail :  String, username :  String,
                 password :  String, repassword :  String)
    postItem(category :  String, heading :  String,
             description :  String, reserve :  Integer,
             days :  Integer)
    browse(category :  String)
    view(aLot :  Lot)
    bid(aLot :  Lot, bid :  Real)
end
```

Seven operations are specified: -

- `login`
- `logout`
- `registerUser`
- `postItem`
- `browse`
- `postItem`
- `viewCommand`
- `bid`

These appear within the test system implementation as separate commands, namely: -

- `LoginCommand`
- `LogoutCommand`
- `RegisterUserCommand`
- `PostItemCommand`
- `BrowseCommand`
- `PostItemCommand`
- `ViewCommand`

- `BidCommand`

Therefore, for each operation within the specification model, a command appears in the implementation class with the same operation name.

It is important to note that these commands only cover the real-time system functionality required in order to implemented the auction system. These functions are those which can be initiated by a client interaction. The real-time functionality can be thought of as those functions that are initiated by the client. Periodic functions, those performed by the server, are not currently specified within the PIM and are therefore not implemented.

As discussed in section 3.3.2, all command specializations are required to extend a super class named `Command`, implementing its abstract methods. By enforcing this inheritance, it ensures that all commands are constructed in the same way, and provides specific areas within the implementation where the developer is required to insert the command specific code. This also provides a means of identifying standardized mappings between the PIM and the areas within the code where information derived from this model must appear.

There are four abstract methods: -

- `getAttributes(...)`
- `checkPreConditions(...)`
- `execute(...)`
- `checkPostConditions(...)`

The sections below discuss the content of each of these methods with respect to the influence the PIM elements have on their implementation.

4.2.1 `getAttributes(...)`

The functionality performed by the `getAttributes(...)` method is to extract the parameters required by the command from the incoming clients request. There is a consistent and identifiable mapping between the system's PIM and the contents of this method. By looking at the relevant operations within the PIM, it can be seen that the parameters that the methods are required to extract are taken directly from the specification. For example, consider the `bid` and `browse` operations from the `WebSession` class specification: -

```
class WebSession
  ...
  operations
    ...
    bid(aLot :  Lot, bid :  Real)
    browse(category :  String)
    ...
end
```

Notice the parameters required by each operation. Now consider the contents of the `getAttributes(...)` methods from each of the command specializations with the test system PSM, which implement the operation functionality: -

BidCommand

```
public void getAttributes(WebSession websession,
                  HttpServletRequest request,
                  HttpServletResponse response)
                         throws Exception{
    // Get parameters from request
    lotId = (String)request.getParameter("lotId");
    bid = (String)request.getParameter("bid");
}
```

BrowseCommand

```
public void getAttributes(WebSession websession,
                  HttpServletRequest request,
                  HttpServletResponse response)
                         throws Exception{
    // Get parameters from request
    category = (String)request.getParameter("category");
}
```

The `browse` and `bid` operation's parameter names map directly into the parameters that the corresponding commands will extract with the exception of the object reference parameter, `aLot`, within the `bid` specification. In the PSM, EJB objects are found through the lookup of their primary key attribute and therefore, within the PIM to PSM mapping, the explicit use of object references is converted into the passing of primary key values.

It is also important to note that the parameter types do not always match because all attributes extracted from the clients request are of type String whereas some parameters within the model are of type Integer. This has implications for the precondition mappings, which will be described below.

4.2.2 `checkPreConditions(...)` / `checkPostConditions(...)`

The content of a commands' `checkPreConditions(...)` and `checkPostConditions(...)` methods is the most obvious sections of within the implementation code where direct mapping from the underlying model occurs. However, is important to establish exactly how the individual OCL statements map into the Java im-

plementation structure architecture. Consider the `checkPreConditions(...)` method taken from the test system's `PostItemCommand`: -

```
public void checkPreConditions(WebSession websession,
                        HttpServletRequest request,
                        HttpServletResponse response
                           throws PreConditionException {
    // PRE_Parameters_Defined
    if(heading.equals("")){
      request.setAttribute("paramheadingblank", "true");
      throw new PreConditionException(
       websession, "PostItem", "heading field blank");
    }

    else if(description.equals("")){
      request.setAttribute("paramdescriptionblank", "true");
      throw new PreConditionException(
       websession, "PostItem", "description field blank");
    }

    else if(reserve.equals("")){
      request.setAttribute("paramreserveblank", "true");
      throw new PreConditionException(
       websession, "PostItem", "reserve field blank");
    }


    // PRE_Valid_Reserve
    try{
      Integer price = new Integer(reserve);
    }
    catch(NumberFormatException e){
      request.setAttribute("invalidreserveformat", "true");
      throw new PreConditionException(
      websession, "PostItem", "reserve field invalid");
    }
}
```

The test system PIM contains the following list of preconditions for the corresponding postItem operation: -

```
pre PRE_Has_Logged_In_User:  self.loggedin.isDefined()
pre PRE_Parameters_Defined:  catagory.isDefined() and
                        heading.isDefined() and
                        description.isDefined() and
                        reserve.isDefined() and
                        days.isDefined()
```

These preconditions map directly into the contents of the `checkPreConditions(...)` method with two exceptions: -

Firstly, preconditions specifying the login status of calling clients do not map into the contents of the `checkPreConditions(...)` method. This is because this has already been taken into consideration in the construction of the `Page` and `Command Access Control` structures. Clients with the incorrect login state for particular commands would therefore not be able to execute those commands on the server.

Secondly, as mentioned before, the PIM contains operations with parameters of type Integer. Parameters extracted from an incoming clients request, however, are always of type String. Therefore, it is important to establish at runtime whether these parameters extracted from the request actually contain valid representations of the corresponding parameter types within the original operation specification. Extra preconditions need to be added to the `checkPreConditions(...)` method in order check any parameter casting that may be required.

For example, within the Post Item command's `checkPreConditions(...)` method above, there is an added PRE_Valid_Reserve precondition. This is required because the `reserve` attribute within the operation specification is of type Integer, whereas the parameter extracted from the request is of type String. Therefore, whenever a parameter appears in an operations specification with a type that is not a String, an extra `PRE_Valid_<param>` precondition will appear within the corresponding command's `checkPreConditions(...)` method to test the validity of the actual parameters retrieved from the client's request.

Now consider the `checkPostConditions` method from the `PostItemCommand`: -

```
public void checkPostConditions(WebSession websession,
                        HttpServletRequest request,
                        HttpServletResponse response)
                           throws PostConditionException {
    // POST_Single_Lot_In_System
    // ...check it exists
    Lot theLot = null;
    try{
      LotHome home = (LotHome) getHomeInterface(
       "auction/Lot", LotHome.class);
      theLot = home.findByPrimaryKey(newPK);
    }
    catch(Exception e){ System.out.println(e); }
    if(theLot == null){
      throw new PostConditionException(
       websession, "PostItem",
       "A single new lot is not in the system");
    }
    // ...check attributes are correctly set
    // ...establish what the should be
    User theSeller = null;
    try{
      theSeller = websession.getUser();
    }
    catch(Exception e){ System.out.println(e); }
    String theCategory = category;
    String theHeading = heading;
    String theDescription = description;
    int theReserve = Integer.parseInt(reserve);
    int theAuctionDuration = Integer.parseInt(days);

    // ...get what they actually are
    User setSeller = null;
    String setCategory = null;
    String setHeading = null;
    String setDescription = null;
    setReserve = 0;
    setAuctionDuration = 0;
    try{
      setSeller = theLot.getSeller();
      setCategory = theLot.getCategory();
      setHeading = theLot.getHeading();
      setDescription = theLot.getDescription();
      setReserve = (theLot.getReserve()).intValue();
      setAuctionDuration =
       (theLot.getAuctionDuration()).intValue();
    }
    catch(Exception e){ System.out.println(e); }
```

```
    // ...test all the attributes are what they should be
    if(!setSeller.equals(theSeller) ||
       setCategory != theCategory ||
       setHeading != theHeading ||
       setDescription != theDescription ||
       setReserve != theReserve ||
       setAuctionDuration != theAuctionDuration){
         throw new PostConditionException(websession,
                   "PostItem", "Lot attributes incorrect");
    }
}
```

The test system specification contains the following post condition for the `postItem` operation: -

```
post POST_Single_Lot_In_System:
    (host.lots.select(l | l.catagory = catagory and
                          l.heading = heading and
                          l.description = description and
                          l.auctionDuration = days and
                          l.reserve = reserve and
                          l.seller = self.loggedin))->size = 1
```

This post condition maps directly into the content of the `checkPostConditions(...)` method implementation. However, it is evident that a single post condition specification within the PIM does not necessarily map to a single possible `PostConditionException` within the implementation. This is due to the fact that a post condition may encapsulate many aspects of a systems state, either of which may be untrue in order to cause the post condition to fail. For example, `POST_Single_Lot_In_System` requires the actual Lot to be present within the system as well as all its attributes to be set correctly. Therefore, there are two possible `PostConditionExceptions` implemented. This will allow the developer to pin point more precisely the error within the code that is causing the problem.

While the PIM dictates the content of the pre and post conditions that must appear for each system operation, no standardized OCL to Java conversion process has been specified within the transformation to the PSM. The extent to which the PIM constraints can be converted into actual implementation code is dependant on existing OCL to Java tool support. It is important to note that further investigation into the mechanisms required for the integration of such tools into the mapping process is still required if automated tool support for the transformations described in this document is to be implemented.

4.2.3 `execute(...)`

The `execute(...)` method contains the functionality of the command that is being constructed. Without the inclusion of an action language within the system PIM, the developer is required to implement the content of this method directly. However, there is a identifiable pattern between the type of operation being implemented and the structure of the `execute(...)` implementation. These structures are discussed in the relevant sections below: -

Action / Response As described in section 3.2.2, an `Action / Response` command performs its function and, if successful, forwards the client onto its `Response` page.

Immediately before the forward operation takes place, attributes can be set within the outgoing request to the `Response` page in order to tailor it appearance.

To demonstrate how this process is realized within the command implementation, consider the following extract from the `PostItemCommand`: -

```
public void execute(WebSession websession,
             HttpServletRequest request,
             HttpServletResponse response)
             throws Exception{

  // Perform Command
  // ...lookup user from websession
  User user = websession.getUser();
  // ...get lot home interface
  LotHome home = (LotHome) getHomeInterface(
   "auction/Lot", LotHome.class);
  ...

  // Set page attributes and forward
  request.setAttribute("heading", heading);
  request.setAttribute("reserve", reserve);
  forwardTo("/servlet/JavaAuction?link=itemposted",
   request, response, caller);
}
```

Firstly, the command functionality is implemented using the parameters extracted by the `getAttributes(...)` method. This has been edited from the extract for clarity.

Secondly, attributes are then set within the outgoing request. In this case with the names `heading` and `reserve`, set to the relevant information. The client is then forwarded onto the commands `Response` page. This page can then extracts this information and formats it for display to the client, as described

before.

Note, the `forwardPage(...)` method that is used is implemented within the `Command` super class.

This two step structure is typical for all `Action / Response` operation's `execute(...)` implementations.

Request / Reply `Request / Reply execute(...)` methods differ from their `Auction / Response` counterparts because they are required to generate formatted information that will become part of the reply sent to the calling client. To demonstrate the structure of a `Request / Reply` execute method, consider the following extract from the `BrowseCommand`: -

```
public void execute(WebSession websession,
             HttpServletRequest request,
             HttpServletResponse response)
             throws Exception{

  // Include header
  request.setAttribute("category", category);
  includePage("/servlet/JavaAuction?link=browseresultsheader",
   request, response, caller);

  // Generate dynamic content
  PrintWriter out = response.getWriter();
  out.println("<FONT face=arial size=3>");

  ...

  // Include footer
  includePage("/servlet/JavaAuction?link=browseresultsfooter",
   request, response, caller);
}
```

Firstly, a header file is included to provide a means of maintaining consistency within the presentation of website as a whole. As before, there is the option of setting attributes within the outgoing request in order to customize information within this header.

Secondly, the method implements the command functionality using the parameters retrieved by the `getAttributes(...)` method to extract information from the system data repository. This information is output directly as part of the generated reply.

Finally, a footer file is included, again for consistency.

Note that the `includePage(...)` method is also implemented within the `Command` super class.

This three step structure is used within all `Request / Reply` operation `execute(...)` methods.

4.3 JSP Allocation and Implementation

Section 3.2.2 discussed how the client side interface is realized through the use of JSPs providing thin client functionality. These JSPs are divided into groups named `Page Sets`, each supporting a different system operation as specified in the PIM. The content of these groups is dependant on the operation type, of which two have been identified. This section investigates the extent to which the PIM can be used as a resource for the allocation and content of the JSPs, and the mechanism through which they are created.

As seen before, a JSP `Page Set` consists of a group of JSP pages, each performing a different role. An `Action / Response Page Set` is made up of two pages named `Action Holder` and `Response` whereas a `Reply / Request Page Set` consists of a `Request Holder`, `Header` and `Footer`. It is important to note that it is the `Holder` pages that contain a HTML Form element used to initiate the relevant system operation to which their `Page Set` is related.

The allocation of the JSP `Page Set`s is directly related to the system specification as one `Page Set` will appear within the PSM per PIM operation. However, extra information is required in order to establish the `Page Set` contents.

Firstly, the PIM in its current form has no distinction between the two types of operation that have been observed. Therefore, using only the system PIM as a resource, there is no way of determining wether an operation specification is of type `Action / Response` or `Request / Reply` and therefore no way of facilitating its mapping to a supporting `Page Set` implementation within the PSM.

Secondly, during the implementation of the test system, is was found that some operation `Holder` pages do not require an actual implementation within their operation `Page Set` in the event that a reply generated by the execution of a `Request / Reply` operation contains its initiating HTML form element. For example, figure 7 showed the generated reply from the test system's `BrowseCommand`. This reply contains a form used to initiate the `ViewCommand` and therefore the `view` operation `Page Set` does not require its `Request Holder` to be present. This is because this operation is always preceded by the `browse` operation and is never executed directly. Using only the system PIM as a resource in its present form, it is impossible to ascertain the intended ordering of operation execution and therefore be able to establish the instances where explicit `Page Set Holder` pages are required. For the test system implementation, the extra information required was provided through the use of a simple diagram constructed to illustrate the different paths a client can take through the application's command structure. This diagram can be seen in figure 10. The round entities represent specific JSPs and are given their roles within the implementation architecture specified using an operation and `Page Set` identifer. The arrows that emanate from these pages specify the system operation for which the page is the client side initiator.

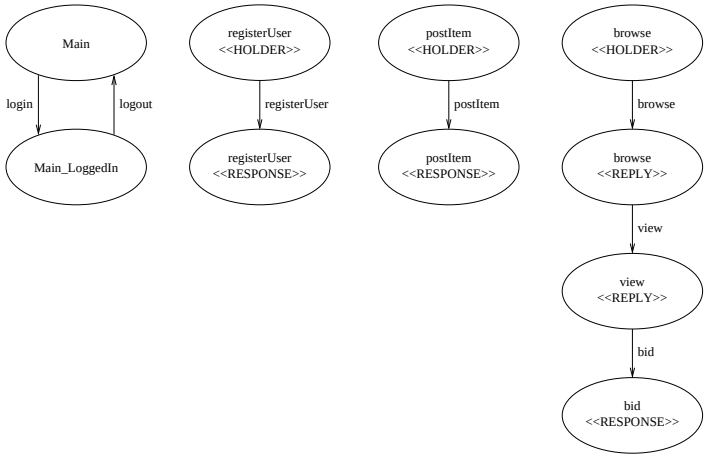

Figure 10: Command Ordering

Using the `BrowseCommand` example from above, it can be seen from the diagram that the page acting as the `BrowseCommand`'s reply page has the `view` operation arrow emanating from it, thereby specifying the `BrowseCommand`'s reply as the holder for the `ViewCommand`. This reply page is therefore required to contain the HTML form element used to initiate the `ViewCommand` and no implementation of a separate `ViewCommand` holder needs to appear within its `Page Set`.

From the diagram it can be seen that the `bid` operation also needs no specific `Page Set` holder for the same reason. In this case, the `ViewCommand`'s reply becomes its initiator.

Section 3.2 discussed the aim of the JSP allocation and implementation method in which a JSP developer could be presented with a document dictating the structure and contents of the JSP pages required to create the client side interface. From the test system PIM it is possible to extract most of the JSP structure and presentation elements required to create a bare minimum implementation of client side user interface. This contains the information required to structure the JSPs with the relevant `link` and `command` initiator functions on the correct pages. Using the test system PIM, a document named the "JSP Technical Overview" [8] was created that contained this information split into sections for each JSP page where each of these sections contain information in four classes: -

- `Page Set`
- `Form`
- `Links`
- `Content Attributes`

The content of each of these sections will now be investigated with respect to the information they provide to the JSP developer and the elements within the PIM that populate them.

4.3.1 Page Set

The name of the JSP page this section refers to is contained with the `Page Set` class, along with the name of the `Page Set` it is contained within.

The specific pages requiring implementation can be obtained by using a method similar to the command ordering diagram described in the previous section.

4.3.2 Form

If the page is used as a commands initiator, the form section will dictate the names of the input fields that should appear within the initiating HTML form element. This will also include the name of the hidden `command` attribute used by the servlet to establish the type of command being requested.

This information is taken directly from the PIM `WebSession` operation specifications.

4.3.3 Links

Since all page linking is governed by the server side system, some method of specifying which sections of the website a client is allowed to request traversal to is required. The links class dictates which sections the client is allowed to request from this JSP page, and therefore tells the JSP developer which values are allowed to be passed to the servlet within a `link` function.

This information can only be taken directly from the PIM if a standardized way of specifying client state requirements is adopted by the `WebSession` operation specifications. This is because these operations each have JSP `Page Set`'s that will be allocated the state requirements of their related system operation and in order to establish which pages a client is able to link to, the state requirements for the target page must be established to ensure a match with the state the client is currently in.

Within the test system implementation the standardization is implemented through the use of two pre conditions named `PRE_Has_Logged_In_User` and `PRE_Has_No_Logged_In_User`. An operation is attributed one of these preconditions so that the login state requirements of a calling client can be established easily. Two main pages can then be utilized, one for logged in clients and one for logged out, each containing links to their respective operation holder pages. The "logged in" main page acts as the `logout` operation holder with the "logged out" main page act as the `login` holder, thereby facilitating the transition between client states. Operation reply or response pages are only allowed to link back to their related main page thereby providing a simple framework that can be easily extracted from the system PIM.

4.3.4 Content Attributes

The setting and retrieval of JSP attributes within page requests is used to facilitate both error notification through dynamic page formatting and the tailoring of the generic page structure through the use of dynamic fields. A method of dictating the names and use of these fields to the developer must be available. This information class is used for this purpose.

The `Content Attributes` section should therefore include the names of all the page formatting attributes that could possibly be set. These can be, in part, derived from the PIM providing one attribute per `WebSession` operation precondition.

For example, to provide some consistency, some standards were adopted during the creation of the test system PSM.

Firstly, all operations contain a precondition, `PRE_Parameters_Defined`, which requires all the operation's parameters to be instantiated. The attribute that would be set within the outgoing JSP request in the event of the precondition failing followed the following pattern: -

```
paramxblank
```

Where x was the parameter name concerned. The JSP page extracting this attribute would therefore format the page to include an error message informing the user of the error that he has made.

Secondly, as described in section 4.2.2, operations containing parameters of types other that strings would require extra `PRE_Valid_<param>` preconditions within their corresponding command implementations. If this precondition failed, an attribute that follows the pattern shown below would be set: -

```
invalidxformat
```

Where x is the parameter name concerned.

The content of a pages `Content Attributes` section relating to any dynamic fields could be partly generated from the underlying model since `Page Set` response pages would most probably always want to include some command specific information confirming the success of a function. However, the exact type and amount of information that is displayed is a matter of aesthetic design and would most probably require some developer intervention.

The "JSP Technical Overview" document for the test system can be found at [9]

4.4 Page Access Control Implementation

Section 3.2.5 described the architecture of the `Page Access Control` method used within the main servlet and its role within the implementation architecture. However, it is important to note the specific elements within the PIM which dictate the contents of this method.

As described before, one section appears within the `Page Access Control` structure per `Page Set`. Each section then contains an entry for each JSP it contains, where section 4.3 described the relationship between the PIM and JSP breakdown. This breakdown has implications for the `Page Access Control` implementation since no page entry within the `Page Access Control` structure need appear for a command page holder that has no specific `Page Set` implementation.

Access to the pages contained within `Page Set`s is governed by any client state precondition attributed to the related system operation specification in the `WebSession` class. Within the implementation, this is achieved through a check made on the calling clients related `WebSession` object to test for an associated `User`. Whether finding a logged in client causes success or failure is dependant on the `WebSession` precondition within the PIM.

To illustrate how a PIM `WebSession` operation specification influences the implementation of a supporting `Page Access Control Implementation`, consider the following extract from the main servlet for the `BidCommand`.

```
// BID

// Action Holder
// ...generated
// Response Page
else if(link.equals("bidcreated")){
  if(websession.getUser() == null){
    RequestDispatcher rd =
     sc.getRequestDispatcher("/error.html");
    rd.forward(request, response);
  }
  else{
    RequestDispatcher rd =
     sc.getRequestDispatcher("/bidcreated.jsp");
    rd.forward(request, response);
  }
}
```

The implementation is influenced in two ways: -

Firstly, the command ordering diagram in figure 10 specifies that the `bid` operation is always required to be preceded by `view`. The `ViewCommand` reply therefore becomes the `BidCommand` initiator and so no specific command holder

page needs to appear for the `bid` operation. This explains its absence from the extract above.

Secondly, the `bid` operation specification within the PIM has a precondition specifying only logged in client access and therefore the check on the clients association `WebSession` object is set to fail if the client is currently logged out.

4.5 Command Access Control Implementation

The structure of the `Command Access Control` method within the main servlet was discussed in section 3.2.5 where its role within the implementation architecture was explained. This method is implemented in the same way as the `Page Access Control` method with the exception that only client state requirements apply.

Consider the following extract from the test system PSM for the `BidCommand`:
-

```
// BID

// Action Holder
// ...generated
// Response Page
else if(else if(command.equals("bid")){
  if(websession.getUser() != null){
    BidCommand view =
     new BidCommand(websession, request, response, this);
  }
  else{
    RequestDispatcher rd =
     sc.getRequestDispatcher("/error.html");
    rd.forward(request, response);
  }
}
```

Since the `bid` operation specification specifies only logged in client access, the check on the `WebSession` object is set to fail if the client is currently logged out, in the same way as before.

5 Conclusions

The PIM to PSM mappings described within section 4 demonstrate the clear relationship which exists between the test system specification and supporting implementation. However, this was only possible through the utilization of a specific modelling pattern and corresponding implementation template.

The modelling pattern although technology independent has been designed

to be domain specific to provide support for primarily e-business systems and their specific requirements. This is achieved through the use of `WebSession` objects representing remote client requests providing server side representations for incoming client interactions. These `WebSession`s can provide extra information relating to their allocated client through the use of associations to other data stored on the server. The server side data store is modelled as set of persistent data objects contained within the container class named `System`. In the context of this pattern, a `WebSession` object has access to that data which is global to the system as a whole, as well as a separate repository of client specific information. The use of this PIM address the complexities of designing for multiple implementation platforms which, as Desmond DSouza states in [2], have no clear overarching architecture. A PIM, however, can be utilized as a basis for the development of implementation technology specific PSMs.

During the mapping process, however, some deficiencies within the PIM in its current form become apparent: -

When implementing the access control structures and JSP allocation, it can be seen that a state machine approach to modelling client status would be a useful inclusion at the at the PIM level. If a client's state referees to both the webpages page that the client is currently viewing as well as any server side state that may be attributed, such as login status, a more sophistication notion of access control could be realized. Therefore, more work into this field is required.

Within the PIM, operations are divided into one of two types: those that update the server side information repository, and those which extract information from it. However, real world systems can contain operations that are made up of both these functions. For example, if the login function of an online massaging service displayed the clients inbox messages if successful, two operations have been performed in one action. Investigation into support for these compound operations is therefore required so that the PIM is to not enforce any structural constraints on a system's topology.

A standardized implementation template was developed in order to facilitate the identification of generic mappings from the PIM. These generic mapping act to populate specific areas within the template utilizing information contained within the corresponding PIM elements. A thin client JSP architecture was utilized in order to centralize this target area on the server side, thereby simplifying the mappings and the system's implementation architecture as a whole.

The implementation of the test system which led to the creation of the standard template highlighted some deficiencies within the PSM: -

As described within section 3.2.5, the access control mechanisms employed within the test system are very simplistic in only ensuring that the calling client making the incoming request is in the correct login state. Therefore, the access control only checks the client's *server side* state. It's *client* side state, the webpage that the client is currently viewing, is not checked. This would involve the client's `WebSession` tracking the client's path through the website structure. Mechanisms for implementing this extra layer of access control therefore requires further investigation.

From an efficiency point of view, it has been noted that this centralized target area can waste network capacity since all precondition checks take place on the server. Some simplistic preconditions, such as confirming all input fields contain data, could be implemented on the client side using javascript.

Comparing the test system PIM to its corresponding PSM yielded the identification of transformational mappings that cover the generation of the majority of the implementation required to support the system specification. However, further work is required at the PIM level if the generation of the implementation was to become a truly automated process. The PIM requires a richer notation set providing the extra information required by some of the PIM to PSM mappings that were identified as missing from the PIM in its current form.

The problems that require further investigation are: -

- Client state modelings is required within the PIM in order to overcome the command ordering problem discussed in section 4.3 and the access control methods in section 4.2.3.
- To provide the automated generation of actual operation functionality within the command `execute(...)` methods, an action language is required within the PIM that allows high level platform independent implementations of operations to be specified.
- The extent to which the pre and post conditions can be realized within the implementation is dependant on the effective use of existing OCL to Java generators. Investigation into the mechanisms required to integrate these tools into the mapping process is therefore required.

Further work is also required at the PSM level since the current PSM template and supporting mappings only provide support for a single set of implementation technologies, namely Java Servlets, JSPs and EJBs. Further investigation could establish what changes would be required to support mapping to other technologies or indeed to facilitate legacy system integration.

Finally, in order to test the validity of the mappings as generic transformation processes, the implementation of supporting tools is required. These tools will provide a framework where the development of a system's PIM becomes the information repository out of which supporting implementations can generated through the implementation of automated PIM to PSM transformations. The specific tooling strategies and user interaction mechanisms employed by these tool will require investigation.

References

[1] L.B. Arief and N.A. Speirs. A UML Tool for an Automatic Generation of Simulation Programs. In *ACM Proceedings of 2nd International Workshop on Software Performance (WOSP 2000)*, 2000.

[2] Desmond DSouza. OMG's MDA, An Architecture for Modeling. http://www.omg.org/mda/mda_files/mdaDDSouza.pdf, 2001.

[3] F. Galan Morillo J. M. Canete Valdeon and M. Toro Bonilla. Filling the gap between specification and implementation of software ystems by an executable code generator of UML/OCL models. In *Proceedings of ICSSEA'99*, 1999.

[4] Jishnu Mukerji. OMG MDA and HP. http://www.omg.org/mda/mda_files/MDA_Briefing_HP_Jishnu_Mukerji v01-1.pdf, March 2001.

[5] OMG. Model Driven Architectue (MDA). Technical Report ormsc/2001-07-01, July 2001.

[6] OMG. Unified Modeling Language, v1.4. Technical Report formal/01-09-67, July 2001.

[7] Mark Richters. The USE Tool. http://www.db.informatik.uni-bremen.de/projects/USE/.

[8] R. P. Smith. JavaAuction JSP Technical Overview Document. http://www.cs.ukc.ac.uk/people/rpg/rps4/emda/appendix/JSPTechnicalOverview.xls.

[9] R. P. Smith. Supporting website. http://www.cs.ukc.ac.uk/people/rpg/rps4/emda.

[10] R. P. Smith and S. Kent. JavaAuction Platform Independant Model. http://www.cs.ukc.ac.uk/people/rpg/rps4/emda/PIM/JavaAuction_PIM.use.

[11] R. P. Smith and S. Kent. JavaAuction Platform Specific Implementation. http://www.cs.ukc.ac.uk/people/rpg/rps4/emda/PSI/JavaAuction_PSI_src.zip.

[12] R. P. Smith and S. Kent. JavaAuction Platform Specific Implementation Template. http://www.cs.ukc.ac.uk/people/rpg/rps4/emda/PSI/JavaAuction_PSITemplate_src.zip.