A Software Architecture for Structuring Complex Web Applications

Mark Douglas Jacyntho (*), Daniel Schwabe (*)[§], Gustavo Rossi (**)
(*) Departamento de Informática. PUC-Rio, Rio de Janeiro, Brazil
E-mail: schwabe@inf.puc-rio.br , mark@inf.puc-rio.br
(**) LIFIA - Facultad de Informática. UNLP, La Plata, Argentina
E-mail: gustavo@sol.info.unlp.edu.ar

Abstract

In this paper we present an architecture for building families of rich Web applications. We first characterize current trends in Web applications, from read-only Web sites to sophisticated applications where complex distributed transactions must be supported. We next some design principles for building Web applications, and give the rationale for separating application behavior from navigation and interface issues. We briefly argue the need for developing a product line architecture for simplifying the systematic construction of different families of applications. We next describe the main components of our architecture explaining how we manage to decouple application specific aspects from technological aspects (such as dynamic page generation and persistence) that can be eventually solved by reusing of-the-shelf components. We show how to build application frameworks using this architecture using a concrete example of an electronic CD store.

1 Introduction

Building complex Web applications such as E-commerce applications is difficult, since these applications often act as integrators of distributed data or behavior repositories, and usually support different user profiles. Web applications combine challenging technological issues such as allowing access from mobile devices, or balancing support for current html/xml releases, with more conceptual ones such as implementing new business models - internet-based business may be quite different from traditional ones.

We need to understand the underlying application domain: objects, behaviors, business rules, etc., and come up with flexible and evolvable software architectures in that domain. To make matters worse these applications evolve continuously and, at the same time, they should be deployed quickly and with zero defects. Systematic engineering and reuse-centric approaches are certainly a must.

In the last 6 years we have been using the Object-Oriented Hypermedia Design Model (OOHDM) [Schwabe 98] for designing complex Web applications. We have also explored reuse in Web applications. We have mined architectural and navigation patterns in these genres of applications [Rossi99], and also discovered patterns in specific application domains such as E-commerce [Rossi00].

Although micro-architectural reuse such as afforded by applying design patterns may help to build high-quality Web applications, we need more powerful approaches in

[§] Daniel Schwabe was partially supported by a grant from Conselho Nacional de Pesquisa Científica – CNPq, Brazil

order to deal with complexity, evolution and short development and maintenance times, in particular in the context of families of Web Applications (e.g. virtual museums, electronic stores, etc). We have introduced Web frameworks as a way to introduce domain knowledge in the application's design architecture [Schwabe 01]. Unfortunately, however, when these patterns and frameworks are mapped onto current implementation settings, we can loose most reuse opportunities if we don't use an architectural-centric approach [Bass 98].

In this paper we present a component-based architecture and an implementation framework for building complex Web applications. We first present our view of Web applications as views on Object Models; we next justify the need for improving current Web implementation architectures (such as J2EE) with components that simplify the development of new applications; we next describe OOHDM-Java2, an architecture that allows decoupling design decisions related with the domain model from those related with the navigation and interface architecture. We show how using this architecture we can implement application frameworks for different families of Web applications. We next give an example in the field of E-commerce. Finally we discuss further issues and draw some conclusions.

2 Design Principles for Web applications

It has been argued that good Web applications should be first good hypermedia applications [Baresi 00a], since the Web is based on the hypertext paradigm, in as much as it is composed of pages (nodes) which can be linked to each other through links (URLs). We should use good hypermedia design practices to come up with applications that are easy to use, provide friendly navigational spaces, and seamlessly integrate the underlying transactional behavior, if any, with the familiar metaphor provided by navigational links.

Most mature Web design methodologies, such as HDM2000 [Baresi 00], WebML [Ceri 00] and OOHDM [Schwabe 98], recognize this fact by clearly separating data design from behavioral aspects of the application, and from the navigational and interface concerns. Clear separation of concerns is widely regarded as a key aspect for obtaining design quality and reuse, as well as ease of evolution and maintenance. However, in many cases these benefits are partially lost during implementation, due to poor support for composition and abstraction mechanisms in current implementation platforms.

Mapping design documents into implementation artifacts is usually time-consuming and, in spite of the general acceptance about the importance of software engineering approaches, implementers tend to overlook the advantages of good modeling practices. The relationship among design models and implementation components are lost, making traceability of design decisions, a fundamental aspect for supporting evolution, a nightmare. We claim that this problem is not only caused by the relative youth of Web implementation tools but mainly by:

- A lack of understanding that navigation (hypertext) design is a defining characteristic of Web applications;
- The fact that languages and tools are targeted more to support fine grain programming than architectural design;

- The inability of methodologists for providing non-proprietary solutions to the aforementioned “mapping” dilemma.

In this paper we present a software architecture together with an implementation framework that allows improving the process of mapping a design schema onto a running Web application. The architecture supports and encourages the use of a set of design principles provided by most methodologies, and implements them in the context of the J2EE architecture, thereby improving existing approaches for using J2EE. The salient principles are:

- Applications should be built using layers in which different concerns are taken into account; in particular, application data should be separated from the page’s contents (navigation nodes) and these contents, in turn, should be clearly separated from the interface look-and-feel (pages). In Figure 1 we illustrate a three-layer design architecture following this principle. When we clearly decouple interface from navigation and from application behavior we have better opportunities for reuse
- Nodes should represent logical views on domain objects, thereby allowing the construction of customized applications according to the user profile simply by changing views instead of changing domain objects. This principle refines the previous one by identifying nodes as first-class abstractions that should not be subsumed in interaction code. For example, in an electronic store, the manager view of a product will be different from the customer view, and for each we may have different look and feel depending on the interface device (as shown in Figure 1).

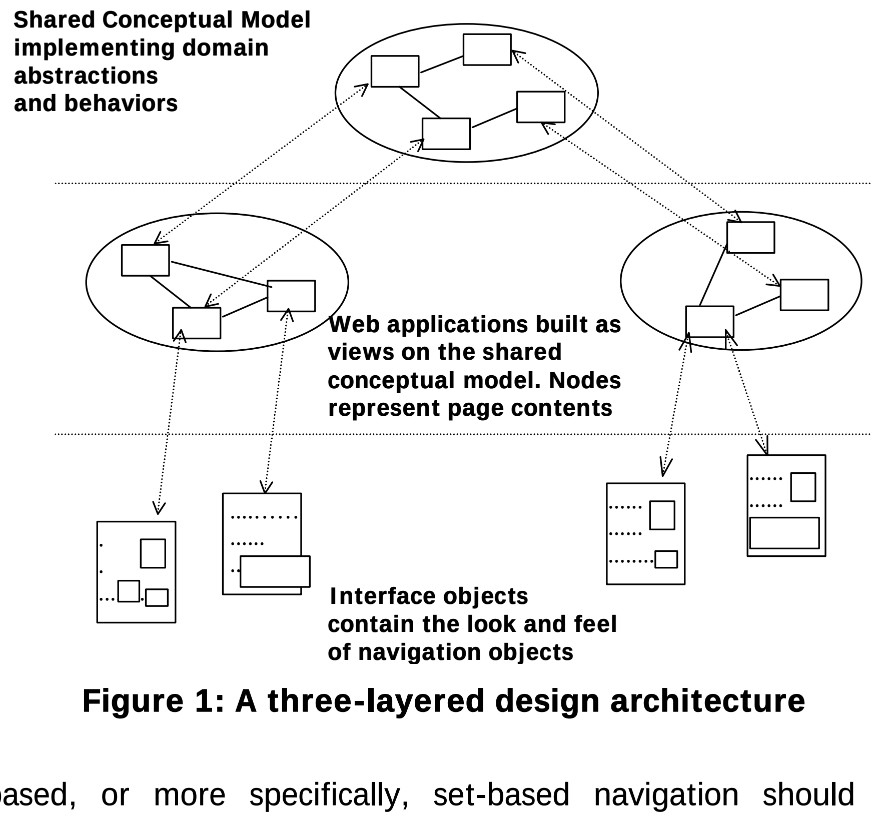

Shared Conceptual Model implementing domain abstractions and behaviors
Web applications built as views on the shared conceptual model. Nodes represent page contents
Interface objects contain the look and feel of navigation objects

Figure 1: A three-layered design architecture

- Context-based, or more specifically, set-based navigation should be provided, simplifying the traversal of collections of related information items. This principle recognizes that navigation always occurs within a context, and treats sets as first-class citizens in Web applications. For example, when we traverse the set of CDs of a rock group, we should have the possibility to move sequentially from one element to the other without backtracking to the index. Since the same CD may also appear in other sets, such as today's recommendations, navigation controls should depend on the set and not only on the node. At the same time, complementary context-related information should ideally be provided; for example for each recommended CD we should explain why we recommend it - and this information would not appear when the CD is accessed within the former set. A different example not involving sets occurs if we want to restrict navigation or other operations in some context: e.g. when we navigate from the shopping cart to a CD we may want to prevent the user from adding the CD again to the cart.

In OOHDM for example, a design model comprises a conceptual, a navigational and an interface model. The conceptual model is described using a variant of UML (with use cases, class diagrams, etc). The navigational model is specified with two schemas, the navigational schema showing nodes and links (as views of conceptual objects) and a context schema that shows indexes and contexts in the application.

As an implementation environment, the J2EE (Java 2 Enterprise Edition) [Sun] which has become popular in the market, is a platform for implementation of distributed multi-tier applications

To illustrate how the use of a software architecture improves the implementation of complex applications we can use as an example the J2EE platform. J2EE consists mainly in three tiers: client, middle and EIS (Enterprise Information System). In the client tier we may have a stand-alone application or a Web browser; the EIS tier usually handles persistence, while the middle tier is itself divided in sub-tiers, which use components deployed into containers. There are two containers: the EJB container and the Web container. In the Web container we deploy JSP pages and servlets, whereas in the EJB container we deploy enterprise java beans (EJB) components that implement business rules in a software applications. EJB components may be entity beans, representing entities stored in some persistence mechanism; session beans, representing a client on the J2EE server, i.e., a logical extension of the client on the server; and Message Driven Beans, which permit J2EE applications to process messages asynchronously.

Using J2EE may be extremely difficult, since a programmer may be led to incorrectly distribute software responsibilities. However if we use a software architecture to reason about the construction of Web applications using J2EE, we can simplify the process. For example, using the Model View Controller (MVC) [Krasner 88] architecture we are not condemned to leave most architectural design decisions to the programmer's wisdom or ability.

3 Why we need a software architecture

Complex Web applications usually involve customized navigation topologies and support for triggering transactions in their underlying corporate application, which in turn may contain sophisticated business rules that may affect the application's behavior and look-and-feel.

A naive approach will face some problems during evolution and maintenance. For instance, if the designer includes business rules inside server pages (e.g. JSP); he will have difficulties to change them if the same rules must be used twice in different pages; or if he deletes or changes an interface, he may inadvertedly erase the rule.

Simply encapsulating business rules in application objects and invoking them from interface pages does not solve the problem, since there still is a strong coupling among interface and business-specific behavior.

The evolution of software platforms for building Web applications shows a perceivable trend towards modular architectures, i.e. those in which program components and their interactions follow established software engineering practices, such as separation of concerns, good support for evolution, etc.

For example, we can consider that the use of the Model View Controller (MVC) [Krasner 88] architecture for improving the development of J2EE-based applications enhances earlier proposals in which most architectural design decisions were left to the programmer's savvy or ability. The MVC architecture, summarized in Figure 2, has been extensively used for decoupling the user interface from application data and from its functionality. Different programming environments provide large class libraries that allow the programmer to reuse standard widgets and interaction styles by plugging corresponding classes into his "model".

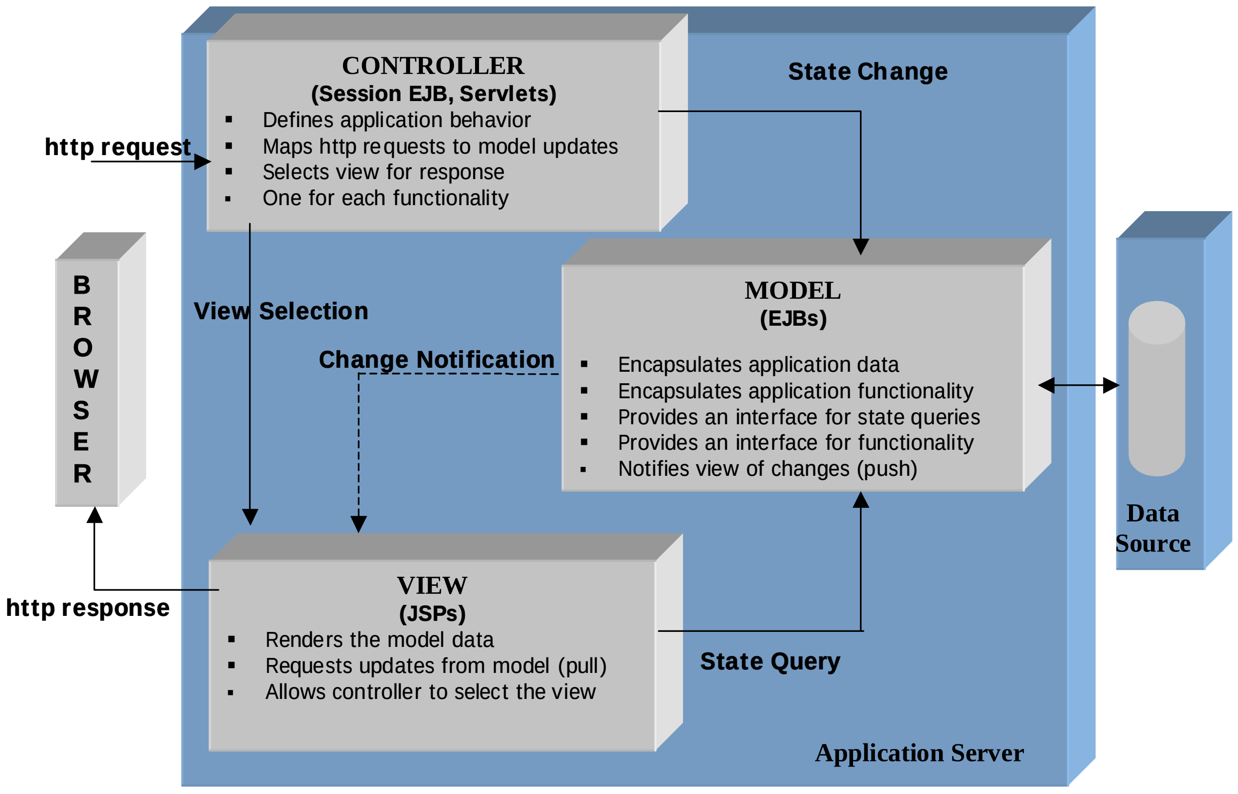

CONTROLLER
(Session EJB, Servlets)
Defines application behavior
Maps http requests to model updates
Selects view for response
One for each functionality
State Change
http request
BROWSER
View Selection
Change Notification
MODEL
(EJBs)
Encapsulates application data
Encapsulates application functionality
Provides an interface for state queries
Provides an interface for functionality
Notifies view of changes (push)
Data Source
http response
VIEW
(JSPs)
Renders the model data
Requests updates from model (pull)
Allows controller to select the view
State Query
Application Server

Figure 2: The MVC architecture

The *model* contains application data and behaviors, also providing an interface for the *view* and the *controller*. For each user interface, a *view* object is defined, containing the information about presentation formats, and is kept synchronized with the *model*'s state. Finally, the *controller* processes the user input and translates it into requests for specific application functionality. This separation reflects well the fact that Web applications may have different views, in the sense that it can be accessed through different clients, such as browsers, WAP clients, Web service clients, etc., with application data separated from its presentation. The existence of a separate module to handle user interaction, the *controller*, (or, more generally, interaction with other systems or user) provides better decoupling between the application behavior and the way in which this behavior is triggered.

Moving to a software architectural approach has many benefits, which will be evident throughout the paper; to mention only some of them (see [Bass 98]) we can:

- Improve communication among stakeholders; as the architecture represents a common high-level abstraction of a system that most stakeholders can use as a basis for creating mutual understanding and forming consensus.
- Represent early design decisions; because architecture represents the manifestation of the earliest and most important decisions with respect to the entire system development cycle.
- Promote large-scale reuse in our applications; because we can apply the same architectural styles to many similar systems regardless to their in-the-small concerns.

These advantages become even more tangible when we can refine our software architecture into a set of smaller-grained architectures for particular families of domain-specific Web applications, such as e-commerce or interactive learning applications. We can build a product-line architecture [Bass 98] and different application frameworks for each domain. The development of a new application will then involve just instantiating and connecting framework components, instead of building everything from scratch each time.

In the following section we introduce the OOHDM-Java2 architecture, explaining its components and the interactions among them, and show an example of use of the architecture.

4 The OOHDM-Java2 Architecture

4.1 Limitations of the MVC architecture

While the MVC provides a set of structuring principia for building modular interactive applications, it does not completely fulfill the requirements of Web applications for providing rich hypermedia structures, since it is based on a purely transactional view of software. Most of all, it does not take into particular consideration the navigation aspects that we have argued should be appropriately supported.

The *view* component subsumes structure and presentation of data, while contents are kept in the *model*. Concretely, in a naive use of the MVC, nodes and their interfaces are handled by the same software component (typically a JSP object).

In addition, the basic idea that navigation always occurs within a context and that context-related information should be provided to the user is absent in the MVC. For example, if we want that the same node has a slightly different structure depending on the context in which it is accessed (e.g. CD in a thematic set or in the shopping cart), we have to use the context as a parameter for the JSP page, and write conditional statements to insert context sensitive information as appropriate. The JSP becomes overloaded, difficult to manage and evolution becomes practically unmanageable. The same problem occurs if we use different JSP pages for different contexts, duplicating code.

An alternative approach is having one JSP page that generates the information common to all contexts ("basic node"), and one JSP page for each "node in context" that dynamically inserts that common JSP page, adding the context sensitive information. This is still unsatisfactory, since this case, the "basic node" layout becomes fixed and we have lost flexibility.

Summarizing, the main problem is that the navigational logic (node and context management) is dealt with in the JSP page, which mixes it with the interface layout. To solve this problem, we propose the creation of a navigational layer encapsulating all navigational logic. While the JSP is responsible for the page layout structure, the navigational layer manages the node contents and deals with context-specific information.

4.2 Overview of the architecture

The OOHDM-Java2 architecture and the associated architectural framework was designed and implemented with the following objectives:

- It should support and encourage good design and implementation practices;
- It should be easily portable across platforms;
- It should have minimum dependencies on particular programming languages and/or tools (such as databases, operating systems or compilers);
- It should implement the most common used abstractions in Web applications leaving the designer the task of just providing application-specific code;
- It should provide support for building complex transactional Web applications;
- It should be easy to use in read-only applications (such as virtual museums);
- Component reuse should be mainly black-box reuse (i.e. the designer should only know the interface of the architectural components).'

OOHDM-Java2 extends the idea of the MVC by clearly separating nodes from their interfaces, thus introducing the idea of navigation object; it also recognizes the fact that navigation may be context-dependent.

In Figure 3 we present the higher-level components of the OOHDM-Java2 architecture, together with the most important interactions between components when handling a request. Although we explain different configurations of the architecture later in this paper, it is important to stress that not all applications need to use all parts of the architecture; simpler applications, such as read-only navigational applications, may use just a sub-set of these components.

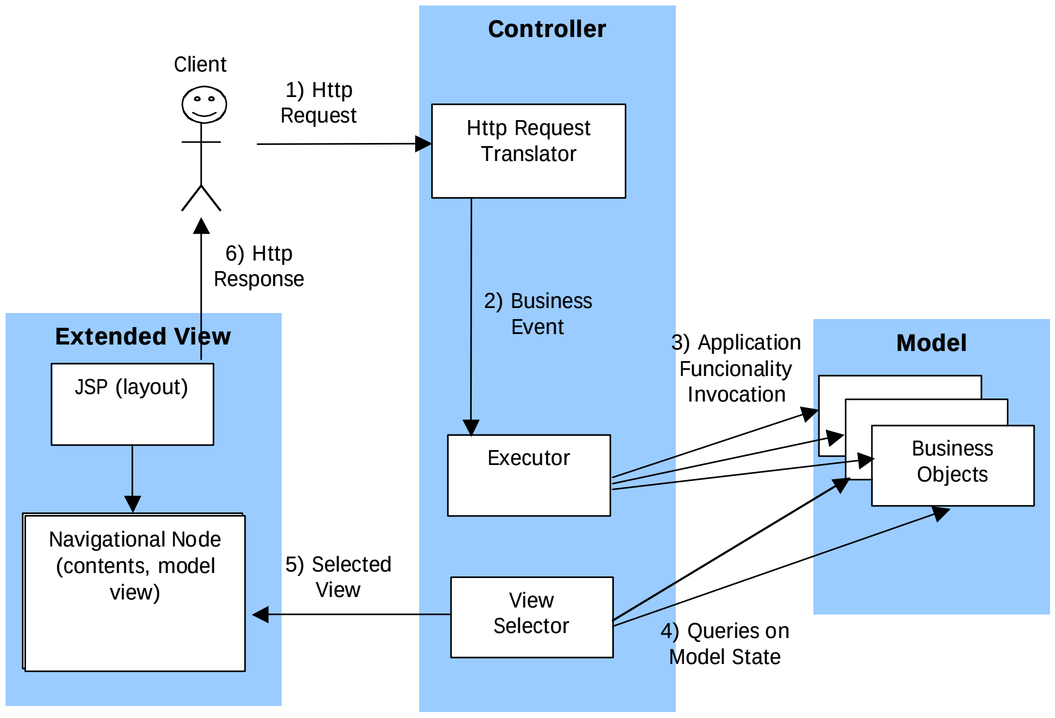


Figure 3: Main components of OOHDM-Java2

The main components of the architecture are summarized below.

Component	Description
ITTP Request Translator Controller)	Every http request is redirected to this component. It translates the user request into an action to be executed by the model. This component extracts the information (parameters) of the request and instantiates a business event, which is an object that encapsulates all data needed to execute the event.
·xecutor (Controller)	This component has the responsibility of executing a business event, invoking model behaviors following some pre-defined logic.
·usiness Object (Model)	This component encapsulates data and functionality specific to the application. All business rules are defined in these objects and triggered from the executor to execute a business event.
·iew Selector (Controller)	After the execution of a business event, this component gets the state of certain business objects and selects the response view (interface).
·lavigational Node (Extended ·iew)	This component represents the product of the navigational logic of the application; it encapsulates attributes that have been obtained

	from some business objects and other navigational sub-components such as indexes, anchors, etc. This component has the contents to be shown by the response interface (JSP).
SP (Extended View)	This component generates the look-and-feel that the client component receives as a response to its request. To achieve this, it instantiates the corresponding navigational node component and adds the layout to the node's contents. Notice that the JSP component does not interact directly with model objects. In this way we can have different layouts for the same navigational node.

4.3 Using OOHDM-JAVA2

We next present a summary of the abstract tasks that a designer must perform in order to instantiate a running application using OOHDM-Java2. In section 5 we refine this explanation referring to concrete classes in an example. This description assumes a more complex application, involving business logic as well as navigation. For read-only applications, the process starts at step 5 below.

1. First, the designer must define the structure and behavior of application business objects, derived from the application's conceptual model; this is one of the points of flexibility of the architecture, analogous to a hot-spot in a framework. Notice that in some domains we can even provide a set of pre-defined classes and behaviors (See section 6), thus providing more specific points of flexibilization.
2. The next step consists in defining the business events in the application, i.e. those objects that encapsulate the parameters needed for that event, coming from the http request. Examples of business events may be adding an item to the shopping cart, creating an order, etc. The component *Http request translator* must be specialized, adding for each different http request, the application specific logic for translating the request into the corresponding business event
3. The *Executor* component is then customized. For each business event object, the designer must indicate the execution logic of this event, i.e. how model objects are invoked to implement the business event.
4. After the execution of the business event it is necessary to send a response to the client. The designer must specialize the *View Selector* component adding the application's specific logic to select the response interface. In some cases this logic is a simple mapping between the http request and the response interface; in other cases it may be necessary to query the model state to select the response interface.
5. Next, or if the application is read-only, the structure of the navigation space is defined by identifying the meaningful contexts (sets) of nodes. These are represented in the architecture by specializing the *Navigational Context* component, specifying the context properties and adding the logic to load the context elements according the context selection and ordering criteria.

6. Once the contexts have been defined, we have to define the structure of nodes in the application, which are usually specified as part of a navigational model in existing methods. This is achieved by refining the *Navigational Node* component, adding the attributes that must be made perceivable to the client.

7. Finally the designer must specify the JSP pages in the application defining the layout for the corresponding navigational node structure. These pages obtain their contents from the navigational nodes defined in step 6.

It should be noted that steps 5-6 can be automatically generated from an XML specification of the design in a DTD such as OOHDM ML [Medeiros 01], since all the information is known at design time.

In Figure 4 we summarize the mapping of an OOHDM design model into the OOHDM-Java2 framework.

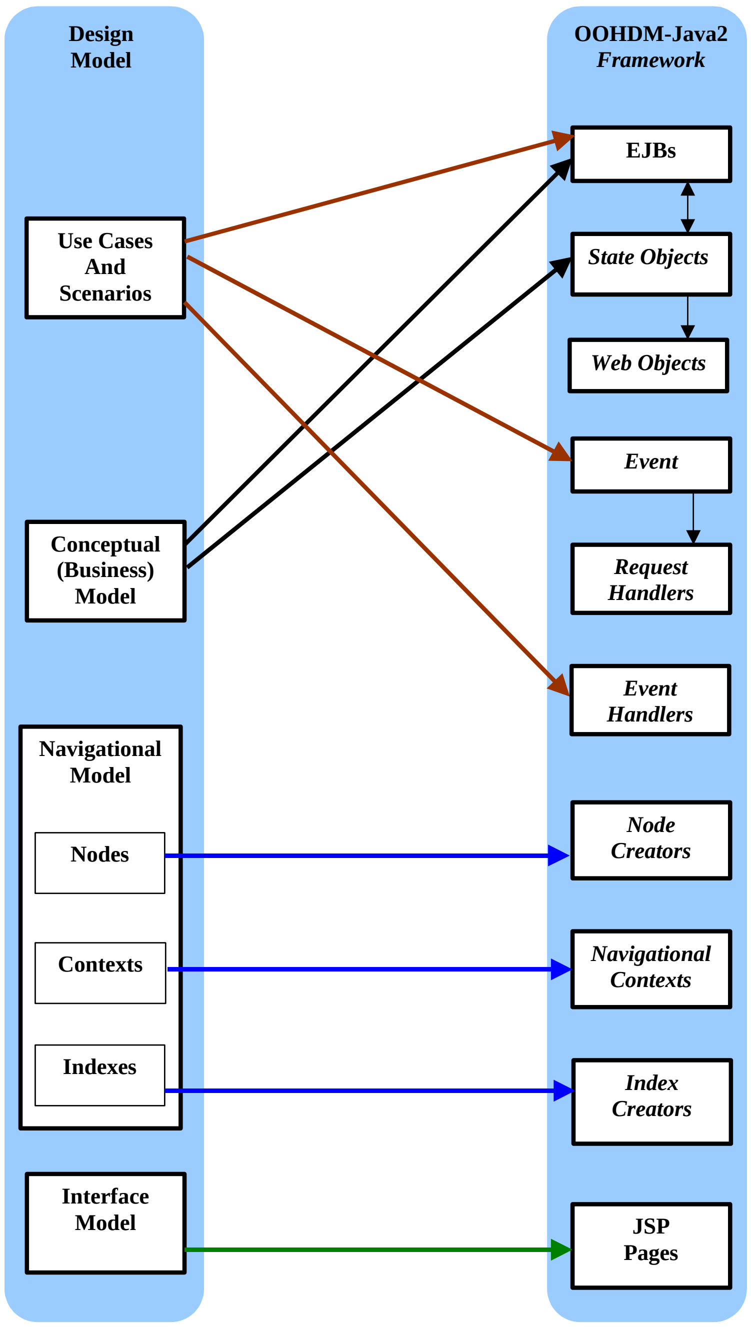


Figure 4: Mapping from a Design Model to OOHDM-Java2

4.4 Event Handling and Processing in OOHDM-Java 2

In this section we refine the description of the architecture by explaining some physical components that implement the previously described logical components. In Figure 4 we show the complete process for treating an http request.

The following description is based in a 3-tier scenario (web tier, EJB tier, and persistence tier). Nevertheless, it is perfectly possible to use this architecture in a 2-tier scenario (web tier, and persistence tier). In the 2-tier scenario the model is composed of simple Java Objects in place of the EJBs, and the EJB Controller core functionality is merged into the Web Controller. The OOHDM-Java2 a architecture implementation has a configuration parameter that must be used to indicate whether is a 3-tier or a 2-tier application. The client tier (the client browser) was omitted for the sake of simplicity.

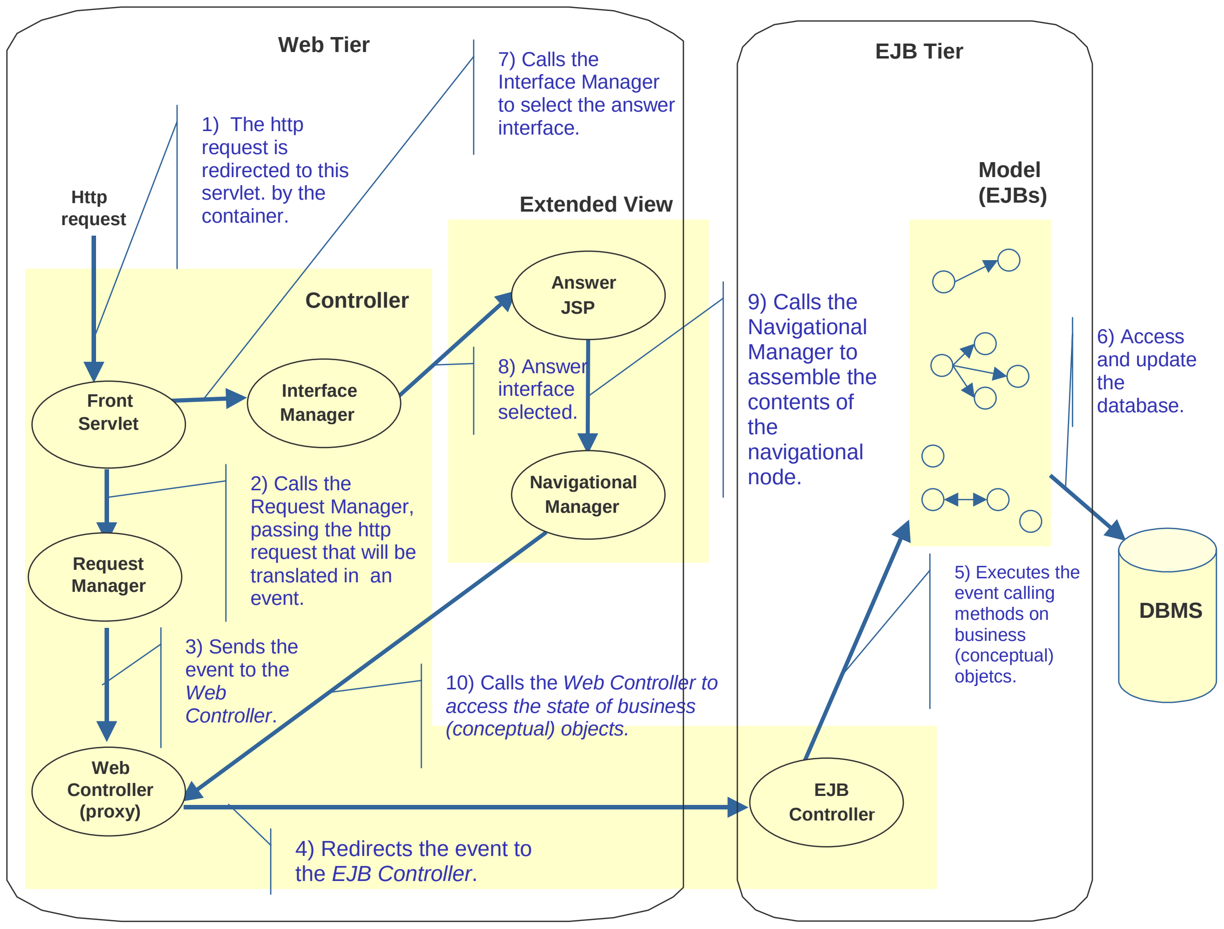

Web Tier
EJB Tier
Http request
1) The http request is redirected to this servlet. by the container.
7) Calls the Interface Manager to select the answer interface.
Extended View
Model (EJBs)
Controller
Answer JSP
9) Calls the Navigational Manager to assemble the contents of the navigational node.
6) Access and update the database.
Front Servlet
Interface Manager
8) Answer interface selected.
Navigational Manager
2) Calls the Request Manager, passing the http request that will be translated in an event.
Request Manager
DBMS
5) Executes the event calling methods on business (conceptual) objetcs.
3) Sends the event to the Web Controller.
10) Calls the Web Controller to access the state of business (conceptual) objects.
Web Controller (proxy)
EJB Controller
4) Redirects the event to the EJB Controller.

Figure 5: Treating an http request in OOHDM JAVA2

All http requests are re-directed by the application server container to the *Front Servlet* component, the single application entry point. When the *Front Servlet* is created it must initialize the application, instantiating all global application components. Treatment of the request occurs in the scope of its main method that is executed within a thread. *Front Servlet* is a black-box component provided by the framework.

The *Front Servlet* sends the http request to the *Request Manager* component, which translates it into a business event. The *Request Manager* uses auxiliary objects called *Request Handler* that know how to handle different http requests. The *Request Handler* contains the logic for generating a business event; an XML configuration file establishes the correspondence between http requests and the Request Handler.

After translating the request into an event, the *Request Manager* sends this event to the *Web Controller*. This is a component that resides inside the scope of the user's session and is instantiated by the *Request Manager* in the first user request. The *Web Controller* is responsible for interfacing with the *EJB Controller* and providing access to conceptual objects.

Upon reception of the event, the *Web Controller* redirects the event to the *EJB Controller*. This object controls the EJB layer, also called business layer. The *EJB*

Controller invokes the appropriate business objects, and notifies the Web Controller of the resulting changes to be reflected in the observer objects.

The EJB Controller invokes the Event *Handler,* which has the specific logic for executing the events, i.e. knows which conceptual objects (EJBs) to invoke, when and how. For each business event there is a corresponding *Event Handler* and this correspondence is defined using an XML configuration file.

At this stage we need to send a response back to the client. The *Front Servlet* invokes a black-box component, the *Interface Manage*r, which selects the response interface. When the mapping between the http request and the response interface is not direct, the *Interface Manager* invokes the *Interface Handler*, containing the logic for selecting the interface based on the request and on the state of conceptual objects. The mapping between the http request and the response interface (and if necessary the Interface Handler) are also specified using an XML file.

In OOHDM-Java2 an interface template is a JSP page containing place holders, known as template parameters. There are two genres of template parameters: those whose value is a text to be inserted directly, and those whose value is another JSP page whose contents are inserted dynamically. Template pages are defined using the custom tag parameters containing an attribute defining the parameter name or place holder of the template. A template plus its parameter's values define an interface. Application interfaces are also defined using XML files.

When the response interface contains a JSP page that exhibits a navigational node or an index, it is necessary to instantiate them using components *Navigational Node* or *Index*. This JSP invokes the *Navigational Manager* component using the custom tag *crate_node* or *create_index*. The *Navigational Manager* invokes the corresponding factory object (*Node Creator* or *Index Creator*) containing the specific logic for creating the node or index. These objects access appropriate conceptual objects through the *Navigation View* interface, which is responsible for implementing the view over the conceptual object.

The *Navigational View* interface declares three methods: *getContextNodeIDs*, *getObjectList*, and *getObject*. The first method returns the node identifiers (*NodeIDs*) of a navigational context. The second return the list of objects that contains the data used to create each index entry of an index. The last method returns a state object encapsulating the actual data used to create the navigational node in a context.

Another important component (not shown in Figure 4 for the sake of simplicity) is the *Navigational Context*, which represents the set of nodes the user is currently navigating. This component interacts with the *Navigation View* component to access the conceptual objects corresponding to the nodes in the current context. The *Navigational Manage*r interacts with the *Navigational Context* to provide adequate contextual information for a *Node Creator* or *Index Creator* to instantiate the node or index. This information includes ID of the next and previous nodes, number of elements in the context, the URL(s) where the node in context is exhibited, etc. In other words, when the node is created, the linking information is retrieved and inserted into the node by the *Node Creator*.

The definition of contexts and the correspondence between contexts and the corresponding *Navigational Context* and *Node Creator* are specified using XML configuration files.

Finally, if the http request does not have an associated business event, i.e. it is only a navigational request, the process is fairly simple; the interface response must be defined, the corresponding node (in context) must be created and the JSP must generate the layout. In this case, the *Front Servlet* invokes the *Interface Manager* and processing follows from that point.

5 Instantiating an application

In this section we explain how a designer instantiates our framework for creating a specific application, illustrating each step using the example of an online CD store. Although we show a step-by-step explanation, the process is usually incremental.

We assume that the designer has built design models (for example using OOHDM), and thus he has partitioned the design space into three different models: conceptual, navigational and interface. For the sake of conciseness we do not describe persistence issues in this explanation. We also give a simplified presentation of the application layer because it is straightforward, e.g. mapping design classes into EJBs (entity beans, session beans, state objects, data access objects, etc), or into Java classes.

In figure 6 we present a simplified version of the OOHDM conceptual model of the CD store.

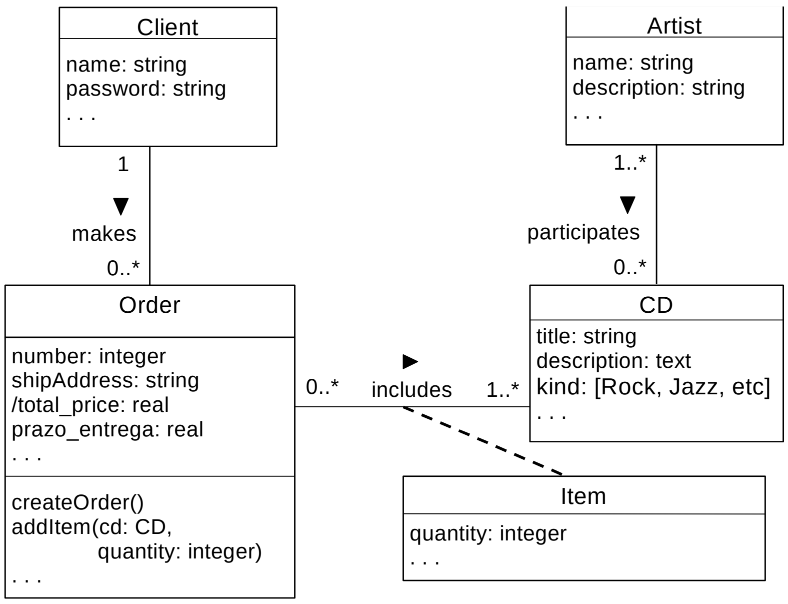


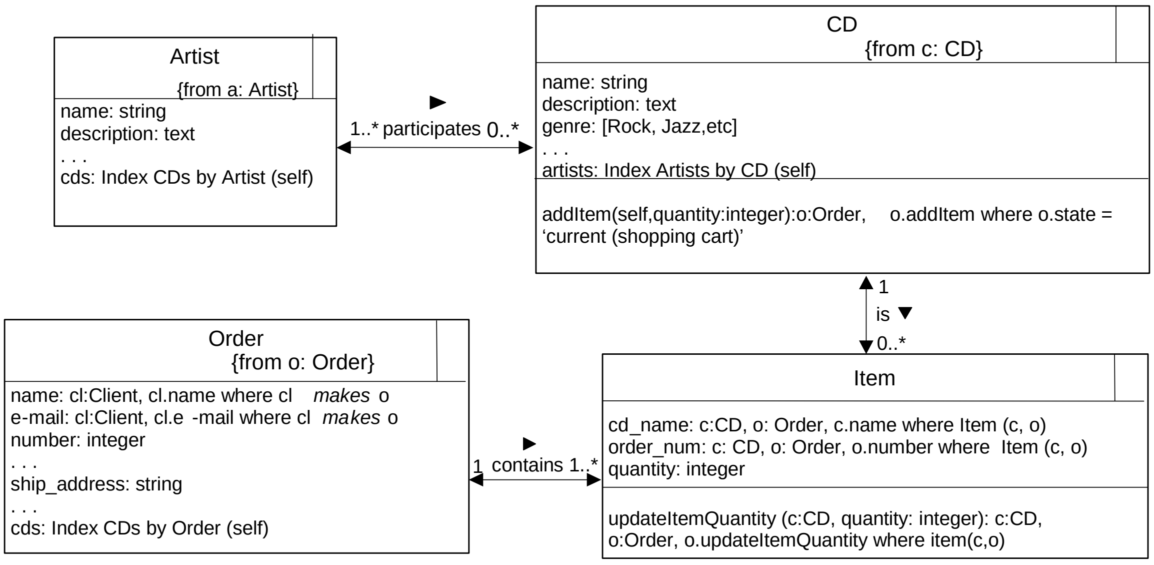
Figure 6 – The OOHDM conceptual model of the CD Store

The EJB layer should only be used for applications with complex transactions and concurrent access to data. Simple database access may be done directly, although we may use pseudo *entity beans* to abstract persistent objects.

For example, analysing the conceptual model and the uses cases of the CD Store the following main objects were identified:

- Catalog – A stateless session bean that provides information about CDs and artists. It is stateless since it is independent of the session or the user.
- Cart – The shopping cart (current order) of the application. It is a stateful session bean, if we assume the shopping cart is not kept in between navigation sessions.
- Account – An entity bean that represents the client account. It is an entity bean since it is persistent.
- Order – An entity bean that represents the client order. It is an entity bean since it is persistent

Once the application model is ready, we must map our navigation objects - nodes, indexes, contexts - into the architecture. The first step is to implement the *Navigational View* interface for those objects (EJBs, State Objects or Web Objects) that must provide information for the creation of nodes and indexes. To implement this interface we must analyse the navigational model to obtain the necessary information. The figure 7 and 8 ilustrate part of the navigational model.



Artist
{from a: Artist}
name: string
description: text
. . .
cds: Index CDs by Artist (self)
1..* participates 0..*
CD
{from c: CD}
name: string
description: text
genre: [Rock, Jazz,etc]
. . .
artists: Index Artists by CD (self)
addItem(self,quantity:integer):o:Order, o.addItem where o.state = 'current (shopping cart)'
1
is
0..*
Order
{from o: Order}
name: cl:Client, cl.name where cl makes o
e-mail: cl:Client, cl.e -mail where cl makes o
number: integer
. . .
ship_address: string
. . .
cds: Index CDs by Order (self)
1 contains 1..*
Item
cd_name: c:CD, o: Order, c.name where Item (c, o)
order_num: c: CD, o: Order, o.number where Item (c, o)
quantity: integer
updateItemQuantity (c:CD, quantity: integer): c:CD, o:Order, o.updateItemQuantity where item(c,o)

Figure 7 – The OOHDM navigational class diagram of the CD Store

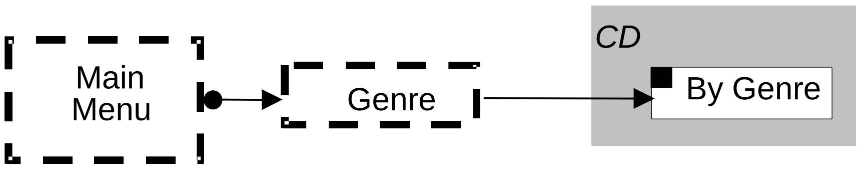


Figure 8 – Part of the OOHDM context diagram (the Genres index and the "CDs By Genre" navigational context)

In our example application, there are several objects that implement the *Navigational View*. For instance, the catalog bean implements this interface to supply conceptual information to all contexts and indexes involving CD or Artist.

To illustrate this step, we show a fragment of code of the *Catalog* EJB that is responsible for implementing the *Navigational View* interface to return data about CDs and Artists. For instance, the *Catalog EJB* queries the database and returns the list of node objects that encapsulate the necessary data for *Index Creator* to assemble the *Kinds* index. The query is made according to the selection criteria specified in the corresponding index spcecification card.

```
public class CatalogEJB implements SessionBean
{
      . . .
      /**
       * Implementation of Navigational View Interface. Used by Navigational
Context components.
       */
      public ArrayList getContextNodeIDs(String contextID,
                                         HashMap parameters)
                                          throws UnknownIDException
      {
            . . .
            Collection auxRresult = null;
            if (contextID.trim().equals(ContextIDs.CD_BY_KIND_ID))
            {
                  String KindID = parameters == null ?
                              null : (String)
                              parameters.get(HttpParameterNames.KINDID);
                  . . .
                  auxRresult = this.getCDsByKind(new ID(KindID));
            }
            else if (contextID.trim().equals(ContextIDs.CD_BY_ARTIST_ID))
            {
                  String artistID = parameters == null ?
                              null :(String)
                              parameters.get(HttpParameterNames.ARTISTID);
                  . . .
                  auxResult = this.getCDsByArtist(new ID(artistID));
            }
            . . .
            else
            {
                  //unknown ID
                  throw new UnknownIDException("The contextID \""
                                    + contextID + "\" is unknown!");
            }
            if (auxResult == null) return null;
            ArrayList result = new ArrayList();
            for (Iterator i = auxResult.iterator(); i.hasNext();)
            {
                  ConceptualObject co = (ConceptualObject) i.next();
                  result.add(new NodeID(co.getThePrimaryKey()));
            }
            return result;

      }
      . . .
      /**
       * Implementation of the Navigational View Interface. Used by Index
Creator components.
       */
      public ArrayList getObjectList(String indexID,
                                     HashMap parameters)
                                    throws UnknownIDException
      {
            . . .
            if (indexID.trim().equals(IndexIDs.KIND_ALPHA_ID))
            {
                  return (ArrayList) this.getAllKinds();
            }
            else if (indexID.equals(IndexIDs.CD_BY_KIND_ID))
            {
                  String genreID = parameters == null ?
                              null :(String)
                              parameters.get(HttpParameterNames.KINDID);

                  return (genreID == null ?
                        null :
                        (ArrayList) this.getCDsByKind(new ID(genreID)));
            }
            . . .
            else
            {
                  //unknown ID
                  throw new UnknownIDException("The indexID \""
                                          + indexID + "\" is unknown!");

            }
      }
      . . .
      /**
       * Implementation of the Navigational View Interface. Used by Node
Creator components.
       */
      public StateObject getObject(String conceptualObjectName,
                                   PrimaryKey objectID)
                                    throws UnknownIDException
      {
            . . .

            if(conceptualObjectName.trim().
                  equals(ConceptualObjectNames.ARTIST))
            {
                  return this.getFullArtist((ID)objectID);
            }
            else if (conceptualObjectName.trim().
                              equals(ConceptualObjectNames.CD))
            {
                  return this.getFullCD((ID)objectID);
            }
            else
            {
                  throw new UnknownIDException("The conceptualObjectName \""
                                                + conceptualObjectName
                                                + "\" is unknown!");
            }
      }

      . . .
}
```

There are several events in a CD store, corresponding to the business logic. For instance, analysing the sequence of actions and operations in the use case “Select CDs by genre to buy” we may have the event “add CD to shopping cart” or in the use case “Confirm order”, the event “create a new order”.

For each business event we define the structure of corresponding event object (from *Event* component). We must identify the business objects affected by the event because the event structure must contain all data needed for its processing by the corresponding *Event Handler,* which is captured from the http request by the associated *Request Handler*.

For example, the following code defines the event *AddItemEvent* for the CD store. This event adds a CD into the stateful session bean *Cart*.

```
. . .
/**
 * AddItemEvent.
 */
public class AddItemEvent implements Event
{
   //Item
    private CartItem item = null;
     //constructor
    public AddItemEvent(String cdID, int quantity)
    {
     this.item = new CartItem(new ID(cdID), quantity);
    }
    public CartItem getItem()
    {
     return this.item;
    }
}
```

The navigational layer is straightforward. Before showing more details, it should be stressed that most of this code can be derived automatically from an XML representation of the design in a DTD such as OOHDM ML [Medeiros 01].

For each context in the application we must create the corresponding *Navigational Context* object. The implementation of these objects consists in getting context parameters from the http request, accessing the *Navigational View* to retrieve the identifiers of the Nodes in Context and returning these identifiers. For each index, we define the corresponding *Index Creator*, implemented in an analogous manner.

In the former figure 8we present one of the navigational contexts of the CD store application: “CDs By Kind”. The specialization of the Navigational Context component is shown in the following code.

```
. . .
public class CDByKindContext extends NavigationalContext
{
     . . .
     /**
      * Implementation of the load method of the superclass.
      */
     protected ArrayList load(HttpServletRequest request,
                              WebController webController)
                              throws ContextException
     {
          . . .
          return this.getNodeIDs(request, parameters,
                                 CatalogJNDINames.CATALOG_EJBHOME);
     }
}
```

In the code above the *getNodeIDs* (implemented in superclass) method invokes, transparently, the *Web Controller* to get the *Navigational View* whose ID is *CatalogJNDINames.CATALOG_EJBHOME*, that is, the Catalog EJB. The Catalog EJB is invoked to obtain the list of node identifiers of the context.

Finally we must create nodes for each context (component *Navigational node)*. For each node present in the navigational class diagram (base node) and node in context type we define the corresponding *Node creator*.

The implementation of a *Node Creator* for a base node consists in obtaining the ID of the node corresponding to the http request, accessing the *Navigational View* to obtain the corresponding (*State)* object containing the node's information: attributes, anchors and indexes. The *Navigational Manager* is accessed to obtain context-related information about the node. The code illustrating the creation of a CD base node is shown in the appendix.

The following code shows the Node Creator that instantiates the CD node in the "CD by Kind" context.

```
. . .
/**
 * CD Node in CDByKind context
 */
public class CDByKindNodeCreator extends CDNodeCreator
{
   // default constructor
   public CDByKindNodeCreator()
   { super(); }
   /**  Implementation of createNode method of the Node Creator. */
   public NavigationalNode createNode(HttpServletRequest request,
                                      WebController webController)
                                      throws NodeCreatorException
   {
   //base node
   /*  Invoke the createNode of the superclass to obtain the base node */
     NavigationalNode node = super.createNode(request, webController);
     ...
     //add anchor next and previous in current context
     //access the Navigational Manager to obtain the context info
     try
     {/*access the Navigational Manager to obtain the context info */
        ContextInformation infoCDByKind =
          this.navManager.getContextInformation(request,
                                             ContextIDs.CD_BY_KIND_ID,
                                             new NodeID(new ID(cdID)));
        if (infoCDByKind.getNextNode() != null)
        {
          Anchor anchor = new Anchor(AnchorIDs.NEXT);
          anchor.setText("next");
          anchor.setURL(infoCDByKind.getSimpleContextURL());
          anchor.addParameter(HttpParameterNames.CDID,
                      infoCDByKind.getNextNode().toString());
          anchor.addParameter(HttpParameterNames.KINDID, genreID);
          node.addComponent(anchor);
        }
        if (infoCDByKind.getPreviousNode() != null)
        {
           ... /* similar to NEXT above */
   }
     }
     catch(ContextException ex)
     { throw new NodeCreatorException("Error on creating the context "
                             +" node of context: " + ContextIDs.CD_BY_KIND_ID
                             + " Cause: "+ex.getMessage());
     }. . .
     return node;
   }
}
```

Note that in the code above the class *CdbyKindNodeCreator* is a subclass of *CDNodeCreator,* and the first step in the *createNode* method is to invoke the *createNode* of the superclass to obtain the base node.

Once we have defined the navigational nodes and indexes, it is necessary to create the interface objects, that is, the JSP pages that will establish the layout of the nodes and indexes. In the JSP page, the first step we must do is to use the appropriate custom tag (*create_node* or *create_index*) to instantiate the node or index. Then we access the node or index to create its "look and feel" for its information.

The following JSP page excerpt presents the CD node in the "CDs by Kind" context in the CD store application.

```
<%@page contentType="text/html"%>

<%-- imports --%>
. . .

<%@ taglib uri="/WEB-INF/taglib.tld" prefix="oohdmjava2" %>
<%-- Create the node  --%>
<oohdmjava2:createnode context="<%= ContextIDs.CD_BY_KIND_ID %>"/>

<%--  Get the node in the request scope  --%>
<%
    NavigationalNode node =
      (NavigationalNode) request.getAttribute(WebNames.CURRENT_NODE);
%>
          <img border="0"
          src="<%= node.getComponent(AttributeIDs.CD_PHOTO_ID) %>">
          <br>Nome:&nbsp;
            <%= node.getComponent(AttributeIDs.CD_NAME_ID) %>
          <br>Description:<br>
          <%= node.getComponent(AttributeIDs.CD_DESCRIPTION_ID) %>
          . . .
          <br><br><br>Songs:
          <%
            List songs =
                  (List) node.getComponent(ListIDs.SONGS_ID);
            for (int i= 0; i < musics.getTotalEntries(); i++)
            {
                  ListEntry entry = musics.getListEntry(i);
          %>
            <br>
            <%= entry.getAttribute(AttributeIDs.SONG_NAME_ID) %>
          <%
            }
          %>
          . . .
          <% Anchor addCartAnchor = (Anchor)
               node.getComponent(AnchorIDs.ADD_ITEM_CART_ID);
          %>
          <br><br><br><br>
          <a href="<%= addCartAnchor.getDestination() %>">
               <%= addCartAnchor.getText() %>
          </a>
          . . .<br><br>
          <%
               Anchor nextAnchor =
                      (Anchor) node.getComponent(AnchorIDs.NEXT);
               Anchor previousAnchor =
                      (Anchor) node.getComponent(AnchorIDs.PREVIOUS);
    %>
          <%
               if (previousAnchor != null)
               {
          %>
               <a href="<%= previousAnchor.getDestination() %>">
                      <%= previousAnchor.getText() %>
                 </a>
. . .
```

In addition, we must create the templates and other general JSP pages of the application (responses to events, form pages, error pages, etc).

Finally we must configure the whole application defining corresponding XML files, as follows:

- urlmappings.xml - Maps the request URL into the answer interface and, if necessary, into the *Request Handler and/or Interface Handler*..
- interfaces.xml - Defines each interface of the application, that is, for each interface (present in the urlmappings.xml or exceptionmappings.xml) it defines the template and the values of template parameters.
- eventmappings.xml - Maps each event to respective *Event Handler*.
- exceptionmappings.xml - Maps each event exception to respective *Exception Handler* or answer interface.
- contextsmappings.xml - Defines all contexts of the application (navigation type, *Node Creator*, *Navigational Context*, URL(s)). Each URL present here must have a correspondent entry in the urlmappigns.xml file.
- indexmappings.xml - Defines all indexes of the application (*Index Creator*, URL). Each URL present here must have a correspondent entry in the urlmappings.xml file.

6 Moving towards a domain-specific architecture

One of the most important benefits of using an architecture-centric approach is that we can improve large-scale reuse of application components. In [Schwabe 01] we introduced the concept of Web design frameworks (WDF) as “a generic, reusable Web application model in a particular domain, that can be later instantiated into specific applications in that domain”. For example, we can describe a WDF for the field of e-commerce by building a generic conceptual model, plus corresponding generic navigation and interface models. and then refine it to build a particular e-store.

The generic conceptual model is similar to an object-oriented application framework [Fayad 99]; it contains generic classes (Product, Order, Invoice, Customer, etc) and corresponding behaviors (e.g. check-out). Building one e-store may imply adding new sub-classes (types of products), refining some behaviors such as different pricing strategies or check-out processes, to adapt the generic model to the specificities of the application.

A generic navigational model meanwhile implies specifying a generic navigational and context schema that abstracts most common navigational topologies in the application domain. For example we may have generic nodes such as Product and Shopping Cart and generic contexts such as “product by <class property>”**.**

The OOHDM-Java2 architecture allows mapping such reusable application models into a domain-specific architecture in which these generic design artifacts are implemented using the base framework components. For example, generic navigation schema and indexes can be implemented defining objects that implement the *Navigational View* interface in a generic manner, and then refined by specializing these generic objects adding the application specific behavior (for example, adding a new context or customizing a state object). In addition, we may define generic *Node Creators* and *Index Creators* to assemble generic contents and then specialize them adding specific attributes of the application.

7 Concluding Remarks

In this paper we have presented the OOHDM-Java2 architecture and its associated component framework. We have described the overall structure of the architecture and the process of building a new application.

The architecture implements an extension of the MVC model and provides the following features:

- It supports a clear separation between application and presentation logic
- Further separation between navigation logic and interface aspects
- Support for navigational contexts and set-based navigation
- Decoupling between JSP pages and business events
- Centralized control of http requests (in particular of the translation of http requests into business events)
- Centralized control of business events execution
- Centralized control on the selection of response interfaces
- Single entry points (Façades) to business objects, both in the Web and EJB layers.
- Single entry point for serializing requests of the same user
- Centralized mapping of business events into corresponding execution logic
- Centralized control of navigation logic

Many of the above features impact positively in the evolution and maintenance of applications built on top of OOHDM-Java2. In particular:

- We can easily alter the interface response
- Changes in the navigational structure do not impact on the business logic
- Similarly, changes in the business logic are transparent to the navigational layer.

- We can easily implement indexes and contexts without writing complex code structures

As applications are configured declaratively using XML files, we get further benefits:

- Each XML file constitutes a point of control
- The navigational and interface application behavior can be altered just by altering or adding XML attributes
- Each XML file provides a map of some specific application functionality, easing its maintenance
- The definition and modification of Contexts and Indexes can be done easily.

There is an evident trade-off between these benefits above and the complexity of the architecture, in particular the learning curve for instantiating an application. We have designed it in order to optimize this compromise. For example, for simple read-only applications, a designer must only use a minimal sub-set of the previously described constructs (see Section 4.3), and a large part of it may be automatically generated from XML specifications using the OOHDM-ML DTD. For complex, transactional applications, we consider that the learning endeavor is similar to the need to use the J2EE platform. From the point of view of design complexity, the overload caused by of OOHDM-Java2 components is then balanced by the benefits mentioned above.

Our current research includes defining frameworks for particular domains such as e-stores catalogs; implementing the translation between OOHDM ML specifications and OOHDM Java 2 implementation of navigational nodes, indexes and contexts; creating *custom tags* that encapsulate the data retrieval from the node in the JSP page; *and* exploring extensions to integrate web services defined using WSDL.

8 References

[Baresi 00] L. Baresi, F. Garzotto, P. Paolini, and S. Valenti: "HDM2000: The HDM Hypertext Design Model Revisited", Tech. Report, Politecnico di Milano, Jan. 2000.

[Baresi 00a] L. Baresi, F. Garzotto, P.Paolini,"From Web Sites to Web Applications: New Issues for Conceptual Modeling", Proc. ER'2000 Workshop on Conceptual Modeling and the Web, Lecture Notes in Computer Science 1921, Springer Verlag, Heidelberg, 2000, pp.89-100

[Bass 98] L. Bass, P. Clements, R. Kazman: "Software Architecture in Practice", Addison Wesley 1998.

[Ceri00] S. Ceri, P. Fraternali, S. Paraboschi, A. Bongio: "Web Modeling Language", (WebML): a modeling language for designing Web sites. Proceedings of the 9th. International World Wide Web Conference, Elsevier 2000, pp 137-157.

[Krasner 88] G. Krasner, S. Pope, A cookbook for using the model-view controller user interface paradigm in Smalltalk-80, Journal of Object-Oriented Programming, 1(3), August/September 1988, 26-49 MVC-based Architecture for e-commerce.Journal.doc 22/22

[Medeiros 01] Medeiros, A. P.; "Declarative Specification and Implementation of

Hypermedia Applicatons in the Web”, MSc Thesis, Dept. of Informatics, PUC-Rio, 2001 (in Portuguese).

[Rossi99] G. Rossi, D. Schwabe, F. Lyardet: “Improving Web Information Systems with Navigational Patterns”, Proceedings of the 8th. International Conference on the World Wide Web, Elsevier 1999, pp 589-600.

[Schwabe98] D. Schwabe, G. Rossi: “An object-oriented approach to web-based application design”. Theory and Practice of Object Systems (TAPOS), Special Issue on the Internet, v. 4#4, pp.207-225, October, 1998.

[Schwabe 99] Schwabe, D.; Pontes, R. A.; Moura, I.; “OOHDM-Web: An Environment for Implementation of Hypermedia Applications in the WWW”, SigWEB Newsletter, Vol. 8, #2, June 1999.

[Schwabe01] D. Schwabe, G. Rossi, L. Esmeraldo, F. Lyardet: “Engineering Web Applications for reuse”. To appear, IEEE Multimedia, Spring 2001.

[SUN] SUN Microsystems, Java 2 Enterprise Edition (J2EE) Official site, http://java.sun.com/j2ee/

9 Appendix

For the sake of completeness, we include here additional examples of Java code and XML specifications, mentioned in the main text of the paper.

9.1 Page Template

In the CD store application there is only one template page, shown below.

```
<%@page contentType="text/html"%>
<%@ taglib uri="/WEB-INF/taglib.tld" prefix="oohdmjava2" %>

<html>
   <head>
     <title> <oohdmjava2:parameter name="TitleContents"/>
     </title>
   </head>
   <body>
     ********************** OOHDM-Java2 ***************************
     <br>
     <a href="main">Home</a>
     <br><br>
<table border="1" height="85%" width="100%" cellspacing="0" border="0">
         <tr>
           <td valign="top"><oohdmjava2:parameter name="BodyContents"/>
           </td>
         </tr>
         <tr>
           <td valign="bottom"> <oohdmjava2:parameter name="FooterContents"/>
           </td>
         </tr>
     </table>
     <br><br>
     <a href="main">Home</a>
     <br>
       ********************** OOHDM-Java2 ***************************
     <br><br>
   </body>
```

```
</html>
```

9.2 Creation of CD Base Node

The following code illustrates the creation of the CD base node in the CD store application.

```
. . .
/**
 * Base Node CD.
 */
public abstract class CDNodeCreator extends NodeCreator
{
      //default constructor
    public CDNodeCreator()
    { super();}
    /**
       * Implementation of the creatNode method of the superclass.
       */
    public NavigationalNode createNode(HttpServletRequest request,
                                WebController webController)
                                throws NodeCreatorException
    {
      //get the CD ID
      String cdID = this.getParameter(request, HttpParameterNames.CDID);
      //verify the ID
      if (cdID == null || cdID.trim().equals(""))
         throw new NodeCreatorException("The http parameter \""
                                    + HttpParameterNames.CDID
                                + "\" is null!");
      //get the conceptual Object
      /*call the method of superclass. This method calls the Navigational View
        passed as parameter (CatalogJNDINames.CATALOG_EJBHOME)*/
      FullCD cd = (FullCD)
         this.getConceptualObject(request, ConceptualObjectNames.CD,
                             new ID(cdID),
                             CatalogJNDINames.CATALOGO_EJBHOME);

      //instantiate the navigational node
      NavigationalNode node = new NavigationalNode();
      //add the attributes
      node.addComponent(new Attribute(AttributeIDs.CD_NAME_ID, cd.getName()));
      node.addComponent(new Attribute(AttributeIDs.CD_DESCRIPTION_ID,
                             cd.getDescription()));
      node.addComponent(new Attribute(AttributeIDs.CD_YEAR_ID, cd.getYear()));
      node.addComponent(new Attribute(AttributeIDs.CD_PRICE_ID,
                             cd.getPrice()));
      node.addComponent(new Attribute(AttributeIDs.CD_DISPONIBILITY_ID,
                                  cd.getDisponibility()));
      node.addComponent(new Attribute(AttributeIDs.CD_PROCEDENCE_ID,
                             cd.getProcedence()));
      node.addComponent(new Attribute(AttributeIDs.CD_SALE_ID,
                             cd.getSale() ? "Yes" : "No"));
      node.addComponent(new Attribute(AttributeIDs.CD_PHOTO_ID,
                             cd.getPhoto()));

      //performers
      List performersList = new List(ListIDs.PERFORMERS_ID);
      for (int i=1; i <= cd.getTotalPerformers(); i++)
      {
         ListEntry listEntry = new ListEntry();
         Performer performer = cd.getPerformer(i);
```

```
          listEntry.addAttribute(
            new Attribute(AttributeIDs.PERFORMER_NAME_ID,
                          performer.getName()));

          performersList.addListEntry(listEntry);
      }
      node.addComponent(performersList);
      //songs
csList = new List(ListIDs.SONG_ID);

      for (int i=1; i <= cd.getTotalMusics(); i++)
      {...    }
      ...
      node.addComponent(genresList);

      //index of artists pointing to ArtistByCD context
      try
      {
          //access the Navigational Manager to get index
          node.addComponent(this.navManager.getIndex(request,
                                        IndexIDs.ARTIST_BY_CD_ID));

      }
      catch(IndexException ex)
      { ... }
      //anchor to add to shopping cart
      Anchor anchor = new Anchor(AnchorIDs.ADD_ITEM_CART_ID);
      anchor.setText("Add to Shopping Cart");
      anchor.setURL(URLs.ADD_ITEM_CART);
      anchor.addParameter(HttpParameterNames.CDID, cd.getCDID().toString());
      node.addComponent(anchor);
      return node;

   }
}
```

9.3 XML Mappings Specifications

To illustrate the XML configuration files, we present part of some of these files in the CD store application.

Urlmappings.xml:

```
<?xml version='1.0'?>

<!DOCTYPE url_mappings
   SYSTEM "http://localhost:8000/cdstore/dtds/urlmappings.dtd">

<url_mappings>
   <!-- Main Page -->
   <url_mapping path = "/main" interface = "MAIN"/>

   <!-- *************** Contexts ****** -->

. . .
   <!-- CDs By Kind-->
   <url_mapping path = "/cdbygenrecontext" interface = "CD_BY_KIND_CONTEXT"/>
...
   <!-- *********** Indexes ***-->
```

```
    . . .
    <!-- hierarchical index Kind:CD -->
    <url_mapping path = "/genrecdhierarchicalindex"  interface =
                        "KIND_CD_HIERARCHICAL_INDEX"/>

    <!-- *********** Events ****-->
    <!-- AddItemEvent -->
    <url_mapping path = "/additemcart" interface = "CART_CONTEXT">
<request_handler class=
"pucrio.inf.oohdmjava2.cdstore.web.requesthandlers.cart.AddItemRequestHandler
"/>
    </url_mapping>
    . . .
    <!-- new user (client) form -->
    <url_mapping path = "/enteruserdata" interface = "ENTER_USER_DATA"/>
    <!-- NewUserEvent - Trigged by the new user form submit-->
    <url_mapping path = "/createnewuseraccount"
                        interface = "NEW_USER_ACCOUNT_CREATED">
      <request_handler class=
"pucrio.inf.oohdmjava2.cdstore.web.requesthandlers.client.account.
CreateUserRequestHandler"/>
    </url_mapping>
    ...
</url_mappings>
```

interfaces.xml:

```
<?xml version='1.0'?>

<!DOCTYPE interfaces
    SYSTEM "http://localhost:8000/cdstore/dtds/interfaces.dtd">

<interfaces>
    <interface name="MAIN" template="/main.html"/>

    <!-- ******** Contexts *****-->
    . . .
    <interface name="CD_BY_KIND_CONTEXT" template="/template.jsp">
      <parameter name="TitleContents"
          value="CDs by Kind." as_is="yes"/>
      <parameter name="BodyContents"
          value="/cdByKindContext.jsp" as_is="no"/>
      <parameter name="FooterContents"
value="OOHDM-JAVA2 CD Store - Context: CDs By Kind." as_is="yes"/>
    </interface>
    . . .
    <!-- ********** Indexes ***-->
    <interface name="ARTIST_ALPHA_SIMPLE_INDEX" template="/template.jsp">
    <parameter name="TitleContents"
         value="All artists ordered by name." as_is="yes"/>
    <parameter name="BodyContents"
                       value="/artistAlphaSimpleIndex.jsp" as_is="no"/>
      <parameter name="FooterContents"
          value="OOHDM-JAVA2 CD Store - Index: All artists
          ordered by name." as_is="yes"/>
    </interface>
    . . .
    <!-- *************Events *****-->
    <interface name="ENTER_USER_DATA" template="/template.jsp">
      <parameter name="TitleContents"
```

```
          value="New User Form." as_is="yes"/>
       <parameter name="BodyContents"
          value="/enterUserData.jsp" as_is="no"/>
       <parameter name="FooterContents"
          value="OOHDM-JAVA2 CD Store - New User Form."  as_is="yes"/>
    </interface>
    . . .
/interfaces>
```

eventmappings.xml:

```
<?xml version='1.0'?>

<!DOCTYPE event_mappings
    SYSTEM "http://localhost:8000/cdstore/dtds/eventmappings.dtd">

<event_mappings>
    <!-- ********Cart - Stateful Session Bean * -->
    <!-- AddItemEvent -->
    <event_mapping
     event_class="pucrio.inf.oohdmjava2.cdstore.event.cart.AddItemEvent"
     handler_class=
"pucrio.inf.oohdmjava2.cdstore.ejb.eventHandlers.cart.AddItemEventHandler"/>
    . . .
    <!-- ********** Order - Entity Bean ********* -->
    <!-- NewOrderEvent-->
<event_mapping event_class=
     "pucrio.inf.oohdmjava2.cdstore.event.client.order.NewOrderEvent"
     handler_class=
     "pucrio.inf.oohdmjava2.cdstore.ejb.eventHandlers.client.order.
     NewOrderEventHandler"/>
</event_mappings>
```

contextmappings.xml:

```
<?xml version='1.0'?>

<!DOCTYPE context_mappings
    SYSTEM "http://localhost:8000/cdstore/dtds/contextmappings.dtd">

<context_mappings>
    . . .
    <!-- Context: CD by Kind -->
    <context_mapping context_id = "cdByKindContext" navigation_type = "SI"
     context_class="pucrio.inf.oohdmjava2.cdstore.web.navigational.cd.contexts
     .bygenre.CDByKindContext">
<simple url_path = "/cdbygenrecontext"
 node_creator_class="pucrio.inf.oohdmjava2.cdstore.web.navigational.cd.
 contexts.bygenre.CDByKindNodeCreator"/>
    </context_mapping>
    . . .
</context_mappings>
```