A Flexible Framework for Engineering "My" Portals

Fernando Bellas, Daniel Fernández and Abel Muiño
Department of Information and Communications Technologies. University of A Coruña
Facultad de Informática. Campus de Elviña. E-15071. A Coruña. Spain
Tel.: +34 981 167 000
fbellas@udc.es, email@dfernandez.org and abel.muinho@mundo-r.com

ABSTRACT
There exist many portal servers that support the construction of "My" portals, that is, portals that allow the user to have one or more personal pages composed of a number of personalizable services. The main drawback of current portal servers is their lack of generality and adaptability. This paper presents the design of MyPersonalizer, a J2EE-based framework for engineering My portals. The framework is structured according to the Model-View-Controller and Layers architectural patterns, providing generic, adaptable model and controller layers that implement the typical use cases of a My portal. MyPersonalizer allows for a good separation of roles in the development team: graphical designers (without programming skills) develop the portal view by writing JSP pages, while software engineers implement service plugins and specify framework configuration.

Categories and Subject Descriptors
D.2.11 [**Software Engineering**]: Software Architectures – *domain-specific architectures, patterns*; D.2.13 [**Software Engineering**]: Reusable Software – *domain engineering, reusable libraries*.

General Terms
Design, Experimentation.

Keywords
Web Engineering, Web Application Frameworks and Architectures, Portal Technology, Design Patterns, J2EE.

1. INTRODUCTION
Internet portals, such as Yahoo!, Lycos, etc, offer many services (weather, news, sports, etc) providing a huge amount of information to the user. In order to allow users easy access to information, many of these portals also provide personalized versions, the so called "My" portals (my.yahoo.com [15], my.lycos.com, etc). Such portals allow the user to have one or more personal pages composed of a number of personalizable services. The portal typically decorates the response returned by a service with a title and a number of buttons, such as minimize/maximize, edit (personalization) and destroy. The edit button allows the user to access a personalization wizard (form) to personalize the service (e.g. for a news service, selecting the maximum number of headlines and the sections the user is interested in), so that such a service provides information according to the user's preferences (e.g. news headlines corresponding to the selected sections). Usually, the user can also personalize other aspects, such as the layout of services in personal pages and page skins. This portal model is also beginning to appear in corporate intranets, giving users a personalized and restricted view of the company's information.

From an architectural point of view, building a My portal comprises two tasks: building the portal skeleton and integrating the services. The former involves building a web application with support for use cases such as *sign in*, *sign up*, *page creation and destruction*, *selection of service layout in a personal page*, etc. The latter means that for each service, it is necessary to build a personalization wizard, implement support for handling the persistence of the user's preferences and integrate the personalized response in the portal. Furthermore, services may exist before the decision of building the portal has been taken or be supplied by external providers, and in consequence, they can be built with heterogeneous technologies (J2EE, .NET, PHP, etc).

Currently, there exist many software platforms, known as portal servers [19], that support the construction of My portals (BEA WebLogic Portal, IBM WebSphere Portal, Jakarta Jetspeed, etc). These portal servers follow an application-oriented approach, rather than a framework-oriented approach, providing a pre-built portal application where developers can integrate services. As a consequence, they tend to impose a particular model of portal, which results in an inherent lack of generality and adaptability for portal development.

In particular, the persistent objects that must be stored for each user (user registration information, layout of services in a given page, etc), hereafter referred to as "personal persistent objects", are not modeled in a generic way in the pre-built portal. Different portals request different information (login name, password, first name, surname, email address, etc) when a user signs up. Some portals only provide the user with a desktop containing one workspace (page), while others allow the user to create a number of workspaces. Portals also differ in the types of workspace layouts provided to the user (two columns, three columns, one row and two columns, etc).

Some service buttons have states that must be remembered by the portal whenever the user clicks on them. One typical example of such buttons is the minimize/maximize button that almost all portals have. If the user minimizes a service by clicking on the minimize/maximize button, the service should remain minimized until the user maximizes it by clicking on the button again, maybe in a different session. We refer to these buttons as "stateful buttons". Other portals also provide more types of stateful buttons, such as a help button to show help information in place of the normal service response. So, what if the pre-built portal does not provide all the properties the particular portal needs for the user registration information?, what if the pre-built portal does not provide all the types of workspace layouts the particular portal needs?, what if the pre-built portal does not provide all the

WWW 2004, May 17–22, 2004, New York, New York, USA.
ACM 1-58113-844-X/04/0005.

stateful buttons the particular portal needs?, etc. If personal persistent objects are not modeled in a generic way, it will be difficult to support the construction of a portal with personal persistent objects that differ from those provided by the pre-built portal.

Another type of personal persistent objects are those modeling the service preferences. The user's preferences for a news service can be represented as an object storing the maximum number of headlines and the sections the user is interested in. The user's preferences for a stock quote service could be modeled as an object storing the list of stock symbols the user is interested in. Representing such objects in a generic way allows to automate persistence and provide generic support for the implementation of personalization wizards (all service personalization wizards retrieve the appropriate preferences object from the database, modify it with the data specified by the user, and finally store it in the database).

Another consequence of not modeling personal persistent objects in a generic and consistent way is that it is more difficult to exploit user information in an efficient way. As an example, consider that the portal administrator, as part of an advertising campaign, wishes to get the email addresses of all users that live in Spain and have added the news service to one of their personal pages, personalized with the "economy" section. In order to make this possible, assuming that the underlying database is relational, which is usually the case, personal persistent objects should be represented in a generic way and mapped according to the relational data model, so that complex queries can be executed in an efficient way. Finally, since the pre-built portal already provides the portal view, it will not be easy to change it significantly.

This paper extends the short paper [3] presenting the main design decisions of the final architecture of MyPersonalizer [16], a J2EE-based framework for engineering My portals. MyPersonalizer is structured according to the Model-View-Controller (MVC) and Layers architectural patterns [5][17][4], providing generic, adaptable model and controller layers (Figure 1) for building any My portal. The model layer represents personal persistent objects in a generic way, maps them to a relational database, includes a framework for executing model actions and provides an action for each typical use case of the portal skeleton and service personalization. The controller layer builds on Jakarta Struts [10][8]. Struts has become the "de facto" framework for building J2EE web applications structured according to the MVC architectural pattern. The MyPersonalizer controller layer provides a Struts action per use case. Each action gets the HTTP request parameters and invokes the parallel model action. The framework can be used with any standard J2EE web container. In order to assist the developer, a number of command line tools and a web administration tool have been implemented. Such tools are out of the scope of this paper.

In general, implementing a My portal with MyPersonalizer comprises two tasks: implementing the portal view and getting the personalized service responses. The former involves writing the JSP pages making up the portal view. In order to do so, the developer can make use of the JSP Standard Tag Library (JSTL) [14] and all of the view-related infrastructure provided by Struts (e.g. the HTML tag library), since the MyPersonalizer controller layer builds on Struts. So, rather than building a specific view-related technology for use in JSP pages (e.g. a specific HTML tag library), standard technology can be used. Furthermore, these JSP pages can be written without inserting Java code, which enables graphical designers without programming skills to accomplish this task. In order to get the personalized service responses, software engineers must implement a plugin for each service, which typically acts as a proxy of the service. Therefore, this approach allows for a good separation of roles in the development team.

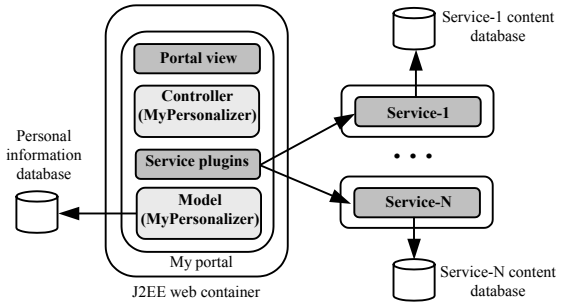

Figure 1. Architecture of a portal built with MyPersonalizer.

The rest of the paper is organized as follows. Sections 2 and 3 present the main design decisions of the model and controller layers, respectively. Section 4 describes an example of portal development. Section 5 compares the framework architecture with recent portal standards. Section 6 presents conclusions and future work.

2. THE MODEL LAYER

2.1 Generic Personal Persistent Objects

Figure 2 shows a class diagram illustrating the classes provided by the framework to model (in a generic way) the persistent objects that must be stored for each user. `ServiceProperty` represents the user's preferences for a service. `ServiceButtonsState` represents the state of the stateful buttons associated to a given service. A `WorkspaceLayout` object specifies the layout of the services in a workspace (page) and also contains the keys of the `ServiceProperty` and `ServiceButtonsState` objects corresponding to the services contained in such a workspace. Similarly, a `DesktopLayout` object specifies the layout of the workspaces the user owns and also contains the keys of the `WorkspaceLayout` objects owned by such a user. Finally, a `UserRegistrationInformation` object contains the user's registration information and the key of his or her `DesktopLayout`.

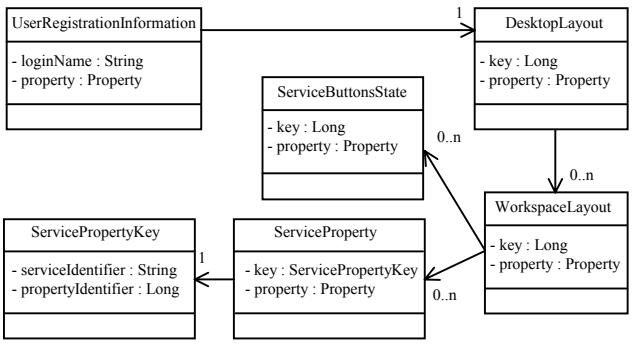

Figure 2. Personal persistent objects provided by MyPersonalizer.

With respect to the keys, `DesktopLayout`, `ServiceButtonsState` and `WorkspaceLayout` have auto-generated numeric keys. The user's login name has been chosen as the key of `UserRegistrationInformation`. And finally, `ServiceProperty` has a compound key consisting of a service identifier (StockQuote, Weather, MyBookmarks, etc) and an auto-generated numeric identifier that uniquely identifies a preferences object for such a service.

Each of the above classes also has a list of properties. As shown in Figure 3, such a list is modeled according to the Composite design pattern [6]. All properties have a simple name and a value. Since some properties (single-valued) can have one value at most (e.g. maximum number of headlines) while another ones (multi-valued) can have a number of them (e.g. sections), the value of a property is modeled as an array of objects. There are two kinds of properties: simple and compound. The value of a simple property (`SimpleProperty`) is an array of objects of basic types (`Integer`, `Float`, etc) or `String`, while the value of a compound property (`CompoundProperty`) is an array of `PropertyStructure` objects, where each object contains a map of properties, which in turn can be simple or compound. All `PropertyStructure` instances are supposed to have the same types of properties. Conceptually, a compound property allows to represent the values of a complex type.

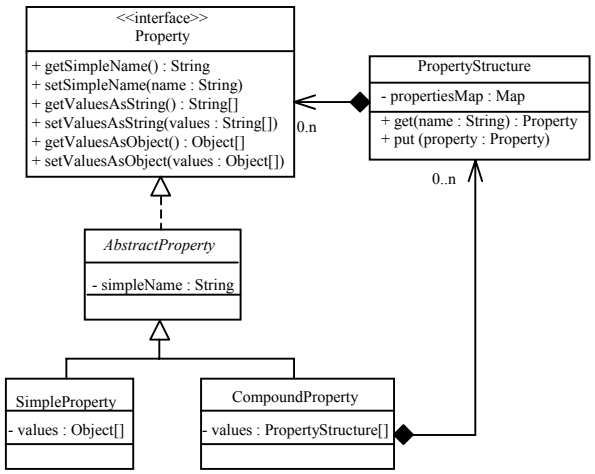

Figure 3. Generic support for property definition.

The `property` attribute contained in each of the classes shown in Figure 2 is a compound property (single-valued or multi-valued) containing the rest of properties. In order to illustrate the flexibility of this system of properties, Figure 4 shows the structure of the properties that could be used for the personal persistent objects in a typical portal. Non-leaf nodes correspond to compound properties, while the leaves of the tree represent simple properties. Multi-valued properties (simple or compound) are marked with an asterisk. Each `PropertyStructure` instance is displayed as a small rectangle below the compound property, with a number representing its index in the array of values.

The properties of a `UserRegistrationInformation` object could be modeled as a compound property containing a simple property for each requested field (login name, password, country, etc), and another one (`dlPropId`) for storing the key of the user's `DesktopLayout` object. The properties of a `DesktopLayout` object could be modeled as a compound property containing a multi-valued simple property (`wlPropIds`) for storing the keys of the user's `WorkspaceLayout` objects and a simple property (`dwlPropId`) for storing the key of the default workspace, that is, the workspace that must be shown when the user accesses the portal at the beginning of a session. Typically, the order in the array of values of the `wlPropIds` property determines the order of the links (usually, presented as tabs) the portal displays in the main page for enabling the user to select a workspace.

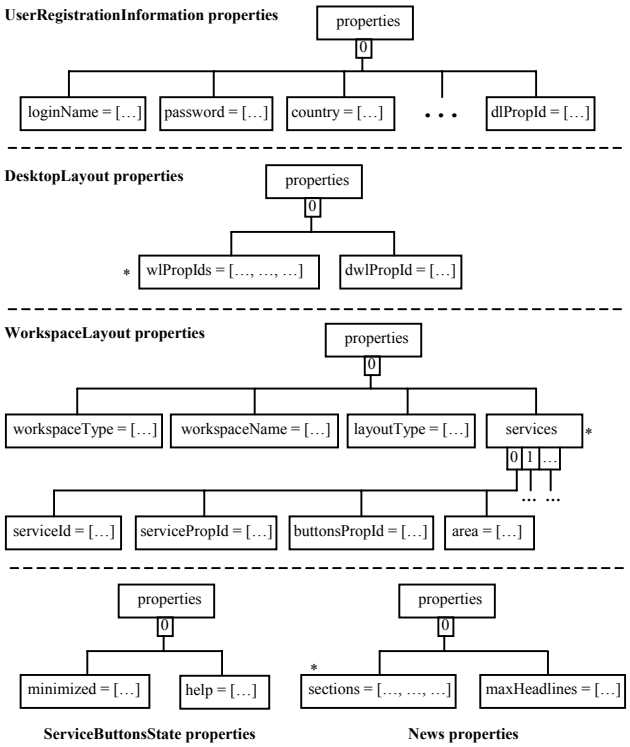

Figure 4. Examples of properties in personal persistent objects.

The structure of properties in a `WorkspaceLayout` object is the most complex one. On the one hand, the root compound property could contain simple properties for storing the workspace type (InternetServices, FinancialIntranetServices, etc), a user-selected name and the type of layout used in such a workspace (2Columns, 3Columns, 1Row2Columns, etc). On the other hand, the root property could contain a multi-valued compound property (`services`), with one value for each service the user has added to that workspace. Each value could contain simple properties for storing the keys of the `ServiceProperty` (`serviceId` and `servicePropId`) and `ServiceButtonsState` (`buttonsPropId`) objects corresponding to that service. There is also an `Integer` simple property (`area`) that specifies the "area" (first column, top row, etc) the service is placed on (e.g. 0 could mean "first column"). There is no need to use a simple property for specifying the position of a service in its area, as long as the order in the array of values of the `services` property is consistent with the positions of services in their corresponding areas.

The properties of a `ServiceButtonsState` object for a portal providing the minimize/maximize and help stateful buttons could be modeled as a compound property composed of two `Boolean` simple properties, one for each button. The properties of the `ServiceProperty` object for a news service could be modeled as a

compound property composed of two simple properties, one for specifying the maximum number of headlines and another one (multi-valued) for storing the news sections the user prefers.

Figure 5 illustrates how to create instances of properties for the news service used above as an example. Even though the creation of properties can seem a little bit complex, it is important to note that the framework hides this complexity from the programmer since properties are created and modified automatically by the framework by using meta-information (Section 2.2). Finally, since values of simple properties can be instances of any basic type (in addition to `String`) and the user types information as character strings, simple properties implement `setValuesAsString` and `getValuesAsString` (Figure 3), which automatically convert from `String` to the appropriate Java type (by using JavaBeans property editors) and vice versa, respectively.

```
Property props = new
  CompoundProperty("properties");
PropertyStructure[] propsValues =
    new PropertyStructure[1];

propsValues[0].put(
  new SimpleProperty("sections",
    new String[]{"international",
      "sports", "economy"}));
propsValues[0].put(
  new SimpleProperty("maxHeadlines",
    new Integer[]{new Integer(3)}));
props.setValuesAsObject(propsValues);
```

Figure 5. Example of property creation.

2.2 Persistence Implementation

In order to abstract the type of persistent storage (relational database, object-oriented database, LDAP, etc) to be used for personal persistent objects and the persistence strategy, the model layer provides an abstract factory [6], `RepositoryAccessorFactory` (Figure 6), that allows to create instances of Data Access Objects (DAOs) [5] (called "accessors" in the diagram). Such DAOs provide methods for finding (by key), creating, updating and removing persistent objects from the database.

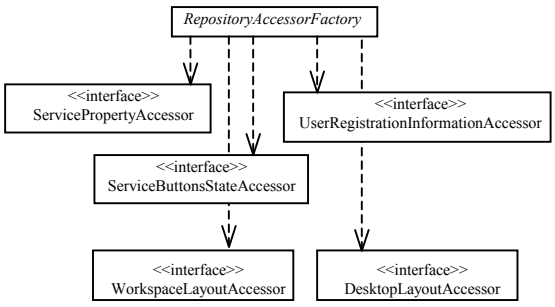

Figure 6. DAO factory.

The framework provides an implementation of the above DAOs that maps objects to a relational database by using a strategy consistent with the relational data model. A compound property is mapped to a table with one column for each single-valued simple property. If the compound property is multi-valued, the table includes a numeric column that allows to maintain the order of values and also acts as the key. A multi-valued simple property is mapped to a table with one column for storing each individual value and a numeric column similar to the one used for multi-valued compound properties. In order to set up relationships in the tree of properties, each table used for a non-root compound property or a multi-valued simple property includes a foreign key referring to the key of the table used by the parent property. As a particular case, the root property includes a column for the key of the corresponding persistent object. This column also acts as the key of the table when the compound property is single-valued.

In order to make possible this mapping, the developer provides the framework with meta-information about the structure of properties in each object. Figure 7 shows the meta-information corresponding to the preferences object (`ServiceProperty`) for the news service used previously as an example. `service-id` specifies the type of persistent object being described: a service identifier if the object corresponds to a service preferences object or a reserved identifier if it corresponds to any other type of object (e.g. `URI` for the user registration information).

```
<service>
    <service-id>News</service-id>
    <persistence-type>RELATIONAL</persistence-type>
    <compound-property>
        <simple-name>properties</simple-name>
        <multi-valued>FALSE</multi-valued>
        <simple-property>
            <simple-name>sections</simple-name>
            <multi-valued>TRUE</multi-valued>
            <java-type>java.lang.String</java-type>
        </simple-property>
        <simple-property>
            <simple-name>maxHeadlines</simple-name>
            <multi-valued>FALSE</multi-valued>
            <java-type>java.lang.Integer</java-type>
        </simple-property>
    </compound-property>
</service>
```

Figure 7. Example of meta-information.

The framework also provides a command-line tool that takes the meta-information of a given type of object as input, and generates the SQL script for creating necessary tables. Figure 8 shows the tables generated for the meta-information specified in Figure 7. The `sections` simple property has been mapped to a separate table (`News1sections`) since it is multi-valued. The `sections` column stores individual values. The `genId` column corresponds to the key of the table and also allows to maintain the order of values. The `propId` column is a foreign key referring to the `propId` key in the `News` table. `propId` is the auto-generated numeric identifier contained in the `ServicePropertyKey` object (Figure 2), which uniquely identifies each instance of `ServiceProperty` for the news service.

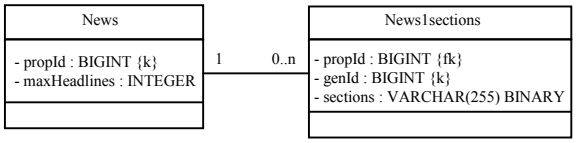

Figure 8. Tables (MySQL) generated for the meta-information specified in Figure 7.

In theory, it should be possible to access any relational database by using JDBC and standard SQL. In practice, shortcomings in the SQL specification and vendor-specific features make it

impossible. The names of the supported column types or the use of sequences versus auto-generated columns are typical examples of incompatibilities. The default implementation of the DAOs provided by the framework and the command-line tool try to be as generic as possible, and ideally any relational database is supported. In order to do so, the specific details of each database (e.g. the default mapping of Java types to the underlying types provided by the database) are isolated in an XML configuration file (one per database). This way, supporting a new database only requires providing its specific configuration file. In particular, the framework has been successfully used with Oracle, PostgreSQL and MySQL.

2.3 Query Language

Figure 9 shows the mapping corresponding to the sample personal persistent objects depicted in Figure 4, except the one corresponding to the news service preferences (Figure 8). `UserRegistrationInformation` maps to the `URI` table. `DesktopLayout` maps to the `DL` and `DL1wlPropIds` (`wlPropIds` multi-valued property) tables. In the same way, `WorkspaceLayout` maps to the `WL` and `WL1services` (`services` multi-valued property) tables. Finally, `ServiceButtonsState` maps to the `SBS` table. Since each object keeps the keys of the related objects (e.g. `dlPropId` property in `UserRegistrationInformation` specifies the key of the user's `DesktopLayout`), relationships are not only set up between the tables a single object is mapped to (`News` and `News1sections`, `DL` and `DL1wlPropIds`, etc), but also between the root tables of related objects (e.g. `dlPropId` in the `URI` table acts as a foreign key referring to the `propId` key in the `DL` table).

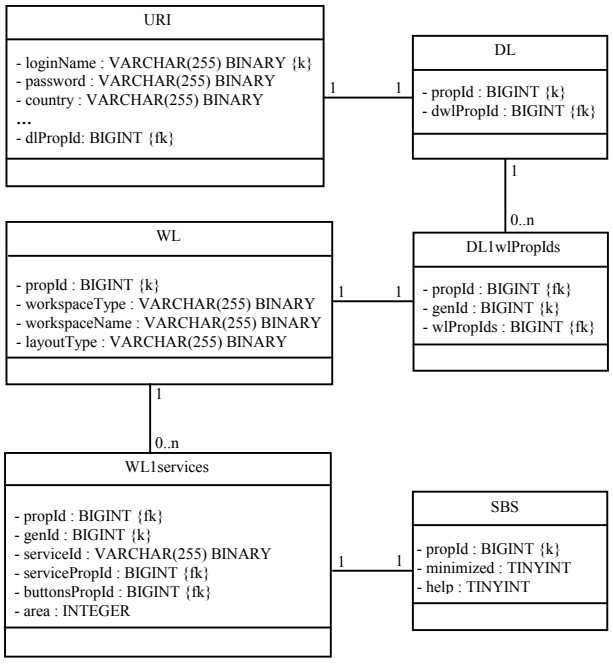

Figure 9. Tables (MySQL) generated for the personal persistent objects depicted in Figure 4, except the news service preferences.

With this strict and consistent object-to-relational mapping for all personal persistent objects, as opposed to a serialization storage strategy, it should be possible to efficiently execute complex queries in terms of personal information. As an example, consider a portal administrator that, as part of an advertising campaign, wishes to send a promotional email (e.g. promoting a new newspaper specialized in Spanish economy) to all the users that live in Spain and have added the news service to one of their personal pages, personalized with the "economy" section. In order to make this possible, the framework must provide (1) a query language, (2) a query engine and (3) a task execution framework.

The query language must allow to retrieve the user registration information corresponding to the users that fulfill a set of conditions in terms of their personal persistent objects. This query language must abstract the type of persistent storage and be easy to use. Rather than proposing a specific query language, we have implemented a small subset (the basic functionality of the `order by` and `return` clauses) of XQuery [22]. XQuery is a rich query language that can be used to query XML data sources. It uses XPath 2.0 [21] expressions to locate nodes in XML data. The data sources do not necessarily store the data in XML. The important thing is that any data source can be seen as XML data. In our case, regardless of the underlying persistent storage, properties in personal persistent objects present a hierarchical structure (Figure 3), and personal persistent objects themselves are hierarchically related (Figure 2). In consequence, personal persistent objects can be seen as XML data sources. As an example, consider the following query:

```
order by /URI/email
return /URI[country = "Spain" and
       desktop/page/services/News/sections="economy"]
```

The above query allows to retrieve all `UserRegistrationInformation` objects, ordered by email, corresponding to the users fulfilling the conditions of the above advertising campaign. In the XPath expressions, some properties correspond to real properties. For example, `country` corresponds to the `country` property in `UserRegistrationInformation`. In order to facilitate the specification of queries, we have allowed the use of virtual properties. For example, `URI` refers to `UserRegistrationInformation`, `desktop` refers to the user's `DesktopLayout`, `page` refers to any of the workspaces the user owns, and so on. Note also that as `sections` is a multi-valued property, the condition `sections="economy"` checks if `economy` is one of the values of such a property.

Figure 10 shows a simplified view of the query engine and the task execution framework. A concrete query engine must implement the `QueryExecutorDelegate` interface that provides operations for executing queries. The framework provides a default implementation for relational databases. Such an implementation translates an XQuery query into a standard SQL query that performs a join of the tables involved in the query. `QueryExecutor` allows the developer to execute a query by selecting the concrete engine specified as part of the framework configuration. `TaskExecutor` allows the execution of a task for all the objects returned by a given query. As part of the web administration tool, we have implemented two tasks: one for removing users and another for sending emails.

It is important to note that we could not have used an existing object-to-relational mapping tool, such as Hibernate [7] or any tool implementing the Java Data Objects (JDO) specification [11], for implementing the mapping of personal persistent objects to the database. These tools are designed for persisting JavaBean classes, which properties (accessible via getter and setter methods) are explicitly defined in the classes, and therefore, can

be discovered by using the Java introspection mechanism. However, MyPersonalizer models the properties of all personal persistent objects by using a generic tree of properties (Figure 3). Each type of object has a particular tree of properties, which can only be discovered by using the corresponding meta-information.

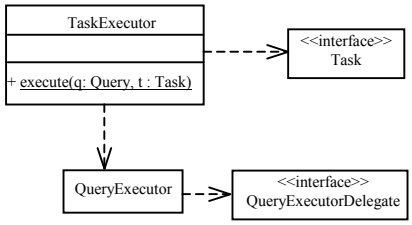

Figure 10. Support for execution of queries and tasks.

2.4 Use Case Implementation

As shown in Figure 11, the model layer provides a default implementation of each typical use case (*sign up*, *sign in*, *change workspace layout*, *change service preferences*, etc) in a separate `Action` class. All actions implement the `execute` method, which receives an event containing the data the use case needs (e.g. the login name and password for the "SignIn" use case) and returns a result. Actions can be transactional or not. Usually, concrete actions extend from `AbstractAction`, which leaves the `execute` method as abstract.

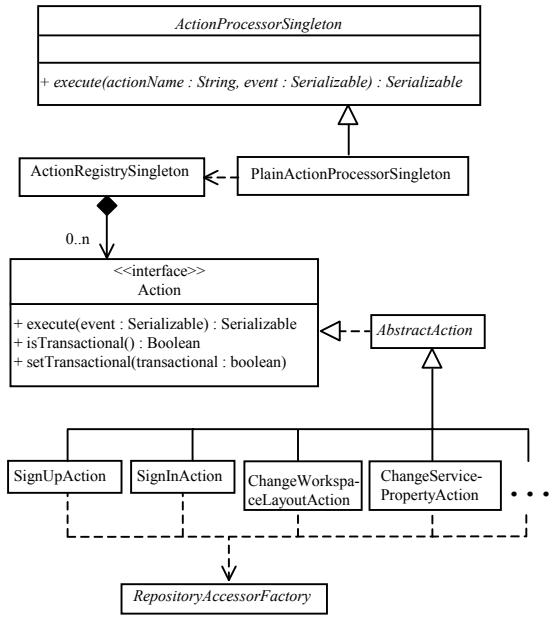

Figure 11. Implementation of use cases in the model layer.

Concrete actions access personal persistent objects by using the appropriate DAOs. As part of the framework configuration, there is an action mapping that specifies for each action: its symbolic name (e.g. "SignIn"), the full name of the action class and whether it is transactional or not. `ActionRegistrySingleton` reads this configuration information and creates a single instance of each action.

Finally, in order to expose an easy interface to the controller layer that hides the details of action execution (and even the action classes themselves), the model layer provides a pluggable facade, `ActionProcessorSingleton`, that corresponds to a variant of the Session Facade and Business Delegate patterns [5][17]. The `execute` method allows to request the execution of an action by using its symbolic name and an event.

The default implementation of `ActionProcessorSingleton` provided by the framework, `PlainActionProcessorSingleton`, makes use of `ActionRegistrySingleton` for getting the requested action and a transaction manager that uses the JDBC basic transaction API for executing transactional actions. Another implementation of `ActionProcessorSingleton` could make use of a different mechanism, such as the Java Transaction API (if available in the web container).

Actions also make use of global pluggable policies. One such policy is the manipulation and the access to the values of some properties required by the framework in the `DesktopLayout`, `WorkspaceLayout`, `ServiceButtonsState` and `UserRegistrationInformation` objects. For example, the framework requires the user registration information to have properties for storing the login name and password. Since the framework can not assume a particular property structure, a mechanism has to be provided that allows to access and modify these properties. In this sense, the framework provides `EditorFactory` (Figure 12), which allows to create an editor for each type of object. Each editor provides methods for modifying and accessing the values of the properties required by the framework. The default editors provided by the framework assume a structure of properties similar to that shown in Figure 4, which is generic enough for most portals.

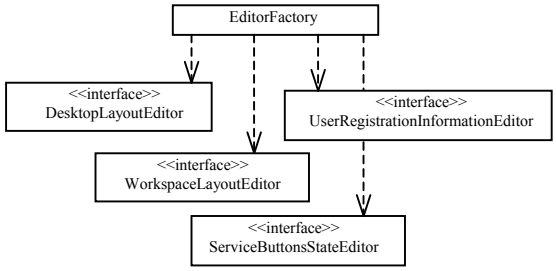

Figure 12. Editors for personal persistent objects.

3. THE CONTROLLER LAYER

3.1 Struts, JSTL and JSF

The controller layer builds on Jakarta Struts [10][8]. Struts has become the "de facto" framework for building J2EE web applications structured according to the MVC architectural pattern. In order to support the implementation of the controller layer, Struts provides a servlet acting as a Front Controller [5] that receives HTTP requests and dispatches them to action classes. Typically, the developer implements an action class per use case. In general, each action gets the HTTP request parameters, invokes the corresponding use case on the model layer (usually invoking a method on one of the facades provided by the model), adds the results of the use case to the request as an attribute (or more), and finally forwards the request on to a JSP page that displays the results.

In order to write the JSP pages displaying results (view layer), the developer can make use of the JSP Standard Tag Library (JSTL) [14], which provides tags for accessing the attributes stored in the

request, iterating over them, checking conditions, etc. In order to write the JSP pages displaying forms, the developer can make use of the Struts HTML tag library that allows to generate data entry fields and treat user errors detected in the server side. Struts also provides other useful components for implementing the view, such as a mechanism for specifying validation rules for data entry fields (Validator) and a template system for defining the screens of the application (Tiles).

These tag libraries together with the controller action framework allow to write JSP pages that do not contain Java code, enabling a good separation of roles in the development team. Graphical designers, without programming skills, write the JSP pages, while software engineers implement the controller and model layers. Regarding the implementation of the model, Struts does not provide any specific support.

In relation to the Struts HTML tag library, the upcoming JavaServer Faces (JSF) specification [13] defines a user interface component model for developing web applications much the same way as the Swing component model does for graphical standalone Java applications. In particular, it provides a tag library that represents a standard alternative to the Struts HTML tag library. Currently, Struts provides a beta version of an integration library with JSF that allows to use the JSF tags in substitution of those provided by Struts. This integration library makes the controller of a Struts-based web application independent of the use of any of these two HTML tag libraries in the view layer.

3.2 Use Case Implementation

The MyPersonalizer controller layer (Figure 13) provides a Struts action per use case that requests the execution of the parallel model action by using `ActionProcessorSingleton`. Controller actions also make use of global pluggable policies (e.g. cookie management). The URLs the graphical designer specifies in the links and forms in the JSP pages point to the corresponding controller actions.

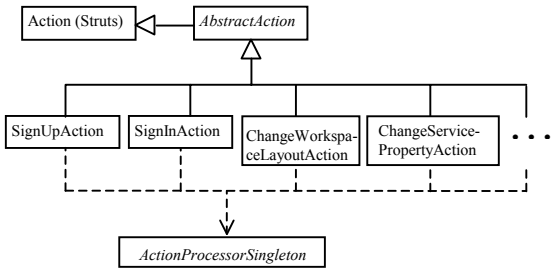

Figure 13. Implementation of use cases in the controller.

Implementing the controller layer upon Jakarta Struts does not only allow to reuse its controller framework, but also allows the graphical designer to make use of all the Struts view-related infrastructure for writing the JSP pages making up the portal view (the HTML tag library, the Validator and Tiles).

3.3 Service Response Integration

Personalized service responses are integrated into the portal by providing plugins as extensions to the controller layer (Figure 14). Each plugin implements the `ServiceController` interface that specifies the `getPersonalizedReply` method. This method takes the service preferences object and the state of the buttons as parameters, and returns the personalized HTML response.

In order to facilitate the implementation of the `ServiceController` interface, the framework provides the `DefaultServiceController` abstract class that implements the `getPersonalizedReply` method as a Template Method [6] by calling the `getPersonalizedReplyRequest` abstract method. This method takes the same parameters as `getPersonalizedReply` and returns a `PersonalizedReplyRequest` object.

A concrete plugin implements `getPersonalizedReplyRequest` by returning a `PersonalizedReplyRequest` object that contains the URL of the service and the parameters for obtaining the personalized response. The plugin constructs the parameters by using the service preferences object and the state of the buttons. The implementation of `getPersonalizedReply` in `DefaultServiceController`, after calling `getPersonalizedReplyRequest`, sends an HTTP request to that URL with the corresponding parameters and sets a timeout. The URL typically corresponds to a component local to the portal, that acts as a proxy of an external XML/HTTP or SOAP/HTTP service, and finally formats the response in HTML. Section 4.3 provides implementation examples of typical plugins.

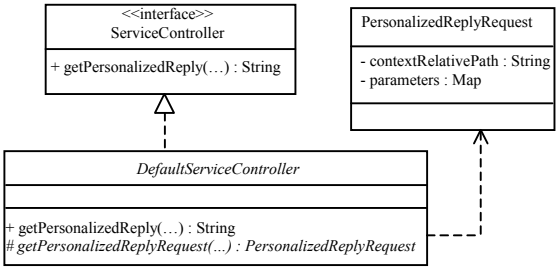

Figure 14. Service response integration.

3.4 Controller Cache

The access to a personal page is a resource-intensive operation since it requires: (1) retrieving the `WorkspaceLayout` object associated to such a page and the preferences object together with the state of the stateful buttons for each service in the page, and (2) sending a request to each service (possibly remote) to obtain the personalized response. Each service in turn has to access its content database. As the user can access several times any personal page during a session, it is important to optimize the execution of this use case. In this sense, for each page the user accesses, the controller caches the `WorkspaceLayout` object together with the `ServiceButtonsState` objects in the session (`HttpSession`). The controller also caches HTML service responses in the database. This cache strategy represents a trade-off between efficiency and scalability. Caching all data in the session is the most efficient strategy, but it does not scale when the number of concurrently connected users increases.

4. PORTAL DEVELOPMENT

4.1 MyPortal: A Sample Portal

In order to test the framework, a sample portal, MyPortal [16] (Figure 15), has been developed. The portal integrates seven services. Two of them, BBC World News and BBC Tech News, are XML/HTTP services provided by the BBC. Another two, Stock Quote and Stock News, are SOAP/HTTP services provided by Xignite. The rest of services, My Bookmarks, Tip Of The Day and Weather, are local services (Weather service only prints

HTML links to weather images provided by Weather Underground). Jakarta Tomcat is used as web container.

Figure 15. MyPortal - A sample portal.

4.2 The Graphical Designer

The graphical designer must write the JSP pages making up the portal view. Basically, there are two kinds of pages: those displaying data and those displaying forms. The former correspond to the pages displaying service responses and to the main page of the portal (that aggregates the responses of the services the user has in the workspace he or she is using). These pages display the data that has been stored in the request as attributes by the corresponding controller actions. The graphical designer makes use of JSTL tags for accessing attributes, iterating over collections, checking conditions, etc.

```
<html:form action="/services/bbcwn/DoBBCWNEdit.do">
  <myper-html:wizard>
    <myper-html:wizard-field name="maxHeadlines"/>
  </myper-html:wizard>
  <table border="0" cellspacing="10"
         class="formTable">
    <tr>
      <td align="right" width="200">
        <b><fmt:message
          key="Wizards.BBCNews.maxHeadlines"/></b>
      </td>
      <td align="left">
        <html:select property="maxHeadlines">
          <html:option value="1">1</html:option>
                  [...]
          <html:option value="10">10</html:option>
        </html:select>
        <html:errors property="maxHeadlines"/>
      </td>
    </tr>
    <tr>
      <td align="center" colspan="2">
        <html:submit>
          <fmt:message key="Buttons.update"/>
        </html:submit>
      </td>
    </tr>
  </table>
</html:form>
```

Figure 16. Excerpt of the JSP page displaying the BBC World News personalization wizard.

The pages displaying forms correspond to those requesting data for use cases such as *sign up*, *sign in*, *change service preferences*, etc. The graphical designer writes each form by using the Struts HTML tag library. The URL specified in the form (`action` attribute) points to the appropriate controller action. The graphical designer has to use just one MyPersonalizer-specific tag (`myper-html:wizard`) that that sets MyPersonalizer-required internal parameters as hidden fields. As an example, Figure 16 shows an excerpt of the page displaying the personalization wizard corresponding to the BBC World News service (Figure 17). The Struts configuration file contains an entry specifying that HTTP requests directed to `/services/bbcwn/DoBBCWNEdit.do` must be processed by the MyPersonalizer controller action `ChangeServiceProperty`.

When the user clicks on the Update button, the request is received by the Struts Front Controller, which delegates the processing of the request to the MyPersonalizer controller action `ChangeServiceProperty`. This action in turn gets the form's parameters (in this case, `maxHeadlines`) and requests the execution of the MyPersonalizer model action `ChangeServiceProperty`. This action retrieves the preferences object from the database, gets the properties to modify (in this case, `maxHeadlines`), modifies them and updates the preferences object in the database.

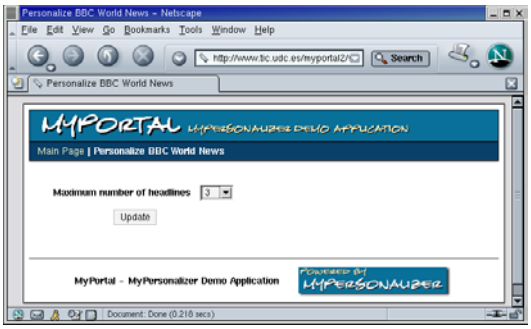

Figure 17. BBC World News personalization wizard.

4.3 The Software Engineer

The software engineer must specify the portal configuration (meta-information about personal persistent objects, Struts configuration, etc) and implement the service plugins that get the personalized responses. Figure 18 shows the typical implementation of a plugin for an XML/HTTP service: the BBC World News service. The software engineer implements the `BBCWNController` plugin by extending from `DefaultServiceController` (Figure 14). `BBCWNController` implements the `getPersonalizedReplyRequest` method. Such a method returns the URL corresponding to `BBCWNResponse.jsp` and the `maxHeadlines` parameter with the value personalized by the user. Next, the `getPersonalizedReply` method (inherited from `DefaultServiceController`) sends a request to such a URL and gets its response. `BBCWNResponse.jsp` connects to the BBC World News service that returns the XML response. Finally, the JSP page formats the XML response by applying an XSL transformation or by using the XML tags provided by JSTL.

The integration of a SOAP/HTTP service differs from an XML/HTTP service in that the URL returned by `getPersonalizedServiceReply` typically points to a Struts action that calls one of the operations of the service. Then, the action adds the service result to the request as an attribute and forwards the request on to a JSP page that formats the result. Figure 19 illustrates the execution of the plugin for the Xignite Stock Quote service.

In the general case, the integration of a local service is similar to the integration of a SOAP/HTTP service with the difference that the Struts action calls on a service model facade that accesses a local content database, rather than calling on a remote web service. If the local service (e.g. My Bookmarks) does not display content other than the service preferences, there is no need to develop a Struts action, and the URL returned by the plugin points to the JSP displaying the personalized service response.

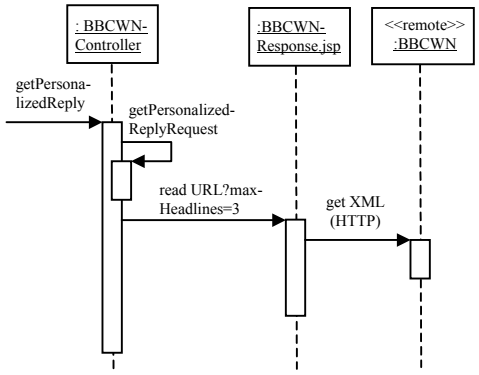

Figure 18. Personalized response integration for an XML/HTTP service.

In all cases, the software engineer implements the necessary Java classes, and the graphical designer writes the JSP pages displaying the service responses.

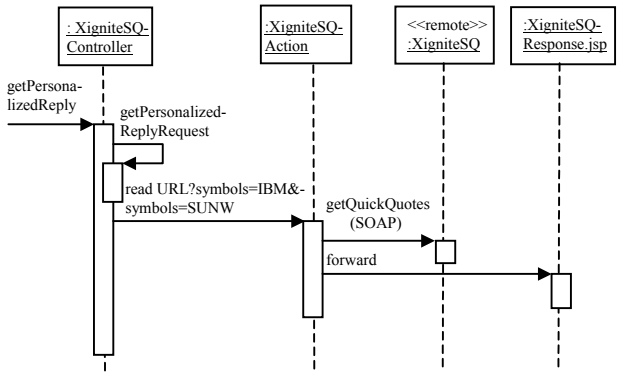

Figure 19. Personalized response integration for a SOAP/HTTP service.

5. COMPARISON WITH RECENT PORTAL STANDARDS

Current remote XML and SOAP services consumed by portals provide only content, rather than the user interface (HTML). This forces portal developers to implement the user interface of each service (e.g. the personalization wizard and the integration of the personalized response) in each consuming portal. To face this problem, in September 2003, OASIS released the first version of the Web Services for Remote Portlets (WSRP) standard [18][2]. In the WSRP standard, a portlet is a remote, interactive web mini-application that renders markup fragments that remote portals can aggregate into a page. A portlet provider, known as WSRP producer, implements a number of standard web service interfaces that allow remote portals to interact with the portlets the provider owns and get the markup.

In October 2003, the Java Community Process released the first version of the Java Portlet Specification (JSR 168) [12][2]. This specification standardizes a Java API for developing portlets that run in a portlet container, which typically is part of a Java portal server. In order to allow the use of Java web technologies, portlets can delegate content generation to servlets and JSP pages.

Figure 20 shows the typical architecture of a Java portal server that supports WSRP and the Java Portlet Specification. The portlet container component manages the life cycle of local portlets much the same way as a J2EE web container manages the life cycle of servlets. The Java Portlet Specification standardizes the API this component exposes to local portlets. The portlet container is also responsible for managing the persistence of portlet preferences. The portal web application component is a web application that defines the portal information model (e.g. user registration information, desktop layout, workspace layout and stateful buttons) and implements the use cases the end user needs (*sign in*, *sign up*, *select portlet layout*, *aggregate portlet responses into a page*, etc). Whenever it needs to interact with the local portlets, it calls on the specific API provided by the portlet container.

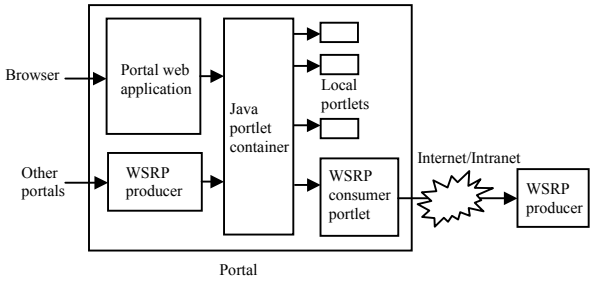

Figure 20. Typical architecture of a Java portal server supporting WSRP and the Java Portlet Specification.

Even though Java portlets are local to the portlet container, the portal server can include a WSRP producer component that provides an implementation of the WSRP interfaces, allowing other WSRP-compatible portals to access local Java portlets. In fact, the Java Portlet Specification can be considered as an API for implementing WSRP portlets in Java. A portal server can also include a WSRP consumer component. This component is implemented as a Java portlet that acts as a proxy of any WSRP producer, allowing the portal to consume remote WSRP portlets.

Compared to the architecture illustrated in Figure 20, MyPersonalizer corresponds mainly to the portal web application component. Portal servers typically implement this component with an application-oriented approach, which results in an inherent lack of generality and adaptability for portal development, as explained in Section 1. MyPersonalizer provides generic, adaptable model and controller layers that allow to develop this component as needed for a particular portal. Unlike current portal servers, MyPersonalizer also allows to exploit user information in terms of personal persistent objects. This has been possible because objects are modeled in a generic way and mapped to a relational database according to the relational data model.

Like a Java portlet container, MyPersonalizer allows to develop portlets. However, as we have developed the framework at the same time the Java Portlet Specification was being standardized, MyPersonalizer does not support the standard Java portlet API. Portlets preferences in the standard Java portlet API are modeled as a set of name-value pairs, where the value of a preference is an

array of `String` objects. WSRP defines a richer model for portlet preferences, where the value of a preference is an array of objects. Object types are defined in an XML schema. In consequence, basic and complex types are supported. This model is equivalent to the system of properties provided by MyPersonalizer. Each preference is represented as a property, which value is an array of objects of basic types or `String` (simple properties), or complex types (compound properties).

Again, as we have developed the framework at the same time WSRP was being standardized, MyPersonalizer does not provide the WSRP producer and consumer components. In consequence, MyPersonalizer portlets can not be consumed by remote WSRP-compatible portals, and MyPersonalizer portals can not consume WSRP portlets.

We are currently refactoring controller and model actions so that a Java portlet container can be plugged into the framework. In particular, we are integrating Jakarta Pluto [9], which is the reference implementation of a Java portlet container. We will also integrate Apache WSRP4J [20], which builds on Jakarta Pluto, and provides the WSRP producer and consumer components.

6. CONCLUSIONS AND FUTURE WORK

Current portal servers provide a pre-built My portal where developers can integrate services. We have followed a different approach for engineering My portals. MyPersonalizer is a framework that provides generic, adaptable model and controller layers that implement the typical use cases of a My portal. Our approach presents the following advantages:

- The model layer represents personal persistent objects in a generic way. This lets developers specify the properties the particular portal needs for each type of object.
- The model layer maps personal persistent objects according to the relational data model. This lets administrators exploit user information, since complex queries can be executed in an efficient way.
- There is a clear separation of roles in the development team. Graphical designers (without programming skills) develop the portal view. Software engineers implement service plugins, redefine model or controller policies (if necessary) and specify framework configuration.
- The controller layer builds on Jakarta Struts. This lets graphical designers use all the Struts view-related infrastructure for implementing the view.

We are currently adding support for WSRP and the Java Portlet Specification in MyPersonalizer, and replacing the Struts HTML tags with the standard tags provided by JavaServer Faces in the MyPortal demo portal (this should not affect MyPersonalizer). With respect to the query language, we are considering what other constructions of XQuery could be useful for exploiting user information. We are also considering a number of future lines of research, in particular, (1) supporting other types of persistent stores (e.g. LDAP) and comparing their performance, (2) using AspectJ [1] (an extension to Java that adds aspect-oriented programming capabilities) to simplify the implementation of crosscutting concerns (logging, security, etc), and (3) wrapping web applications as portlets.

7. ACKNOWLEDGEMENTS

This work has been supported by the Spanish Science and Technology Ministry under contract TIC2001-0547.

8. REFERENCES

[1] AspectJ. http://www.eclipse.org/aspectj.

[2] Bellas, F. Standards for Second-Generation Portals. IEEE Internet Computing, vol. 8, no. 2, March/April 2004.

[3] Bellas, F., Fernández, D., Toral, I., and Muiño, A. Towards a Generic and Adaptable J2EE-based Framework for Engineering Personalizable "My" Portals. Proceedings of the IADIS International Conference "WWW/Internet 2003", pp. 789-792, Algarve, Portugal, 2003.

[4] Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., and Stal, M. Pattern-Oriented Software Architecture: A System of Patterns. John Wiley and Sons, 1996.

[5] Crupi, J., Alur, D., and Malks, D. Core J2EE Patterns, 2nd edition. Prentice Hall, 2003.

[6] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, 1994.

[7] Hibernate. http://www.hibernate.org.

[8] Husted, T., Dumoulin, C., Franciscus, G., and Winterfeldt, D. Struts in Action. Manning, 2003.

[9] Jakarta Pluto. http://jakarta.apache.org/pluto.

[10] Jakarta Struts. http://jakarta.apache.org/struts.

[11] Java Data Objects Specification. http://access1.sun.com/jdo.

[12] Java Portlet Specification. http://jcp.org/aboutJava/communityprocess/final/jsr168.

[13] JavaServer Faces Specification. http://java.sun.com/j2ee/javaserverfaces.

[14] JavaServer Pages Standard Tag Library. http://java.sun.com/products/jsp/jstl.

[15] Manber, U., Patel, A., and Robison, J. Experience with Personalization on Yahoo!. Communications of the ACM, vol. 43, no. 8, pp. 35-39, August 2000.

[16] MyPersonalizer. http://www.tic.udc.es/~fbellas/mypersonalizer.

[17] Singh, I., Stearns, B., and Johnson, M. Designing Enterprise Applications with the J2EE Platform, 2nd edition. Addison-Wesley, 2002.

[18] Web Services For Remote Portlets Specification. http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wsrp.

[19] Wege, C. Portal Server Technology. IEEE Internet Computing, vol. 6, no. 3, pp. 73-77, May/June 2002.

[20] WSRP4J. http://ws.apache.org/wsrp4j.

[21] XPath 2.0. http://www.w3.org/TR/xpath20.

[22] XQuery. http://www.w3.org/TR/xquery.