TOWARDS A GENERIC AND ADAPTABLE J2EE-BASED FRAMEWORK FOR ENGINEERING PERSONALIZABLE "MY" PORTALS

Fernando Bellas, Daniel Fernández, Iago Toral and Abel Muiño
Department of Information and Communications Technologies. University of A Coruña.
Facultad de Informática. Campus de Elviña. 15071. A Coruña. Spain
fbellas@udc.es, email@danigarrido.com, iagotq@lycos.com and abel.muinho@mundo-r.com

ABSTRACT

There exist many portal servers that support the construction of personalizable "My" portals, that is, portals that allow the user to have one or more personal pages composed of personalizable services. The main drawback of current portal servers is its lack of generality and adaptability. This paper presents the approach we are following in the new version of MyPersonalizer, a J2EE-based framework for engineering personalizable "My" portals. MyPersonalizer is being structured according to the Model-View-Controller (MVC) architectural pattern, providing generic and adaptable model and controller layers that implement the typical use cases of a My portal. The controller layer is built upon Jakarta Struts, "de facto" MVC framework for J2EE web applications and is tightly integrated with it. So, in order to build a My portal, developers implement the portal view as JSP pages by using Struts and JSTL tags, integrate personalized service responses, specify portal configuration and maybe redefine some policies if necessary.

KEYWORDS

Web Engineering, Personalized Web Sites and Services.

1. INTRODUCTION

Internet portals, such as Yahoo!, Excite, Lycos, etc., offer many services (weather, news, sports, etc.) providing a huge amount of information to the user. In order to help the user in the access to the information, many of these portals also provide personalized versions, the so called "My" portals (my.yahoo.com, my.excite.com, my.lycos.com, etc.). These portals allow the user to have one or more personal pages composed of a number of personalizable services that the user selects from a palette of available services. Each of these services is displayed with a button bar (edit, maximize/minimize, destroy, etc.) and an information area. The edit button allows the user to access a personalization wizard (form) for customizing the service (e.g. to select sections and the total number of headlines for a news headlines service), so that such a service will present information in its area according to the user's personalization (e.g. the number of news headlines corresponding to the selected sections) whenever the user accesses the page containing it. The user can also personalize other aspects, such as, the layout of services in personal pages and page skins. This portal model is also being used for the construction of personalizable versions of intranet portals, giving users a personalized and restricted view of the company information.

From an architectural point of view, building a My portal involves two key aspects: building the portal skeleton and integrating the services. The former means building a web application with support for use cases such as, signing up and signing in a user, page creation and destruction, and selection of services and their layout in personal pages. The latter means that for each service, it is necessary to build a personalization wizard, implement support for service personalization persistence and integrate the personalized response in the portal. Furthermore, services usually exist before the decision of building the My portal has been taken, and so, they have probably been constructed with heterogeneous technologies (J2EE, .NET, LAMP, etc.).

Currently, there exist many portal servers that support the construction of My portals (e.g. BEA WebLogic Portal, IBM WebSphere Portal, Jakarta Jetspeed, etc.). These portal servers provide a pre-built portal where developers integrate the services. The main drawback of this approach is its lack of generality

and adaptability, since they impose a particular model of portal, with a given user registration information, a fixed set of possible service buttons (e.g. edit, maximize/minimize and destroy), types of page layouts (e.g. two and three columns), etc., which may not be appropriate for all portals. Current portal servers just allow mainly to customize the portal view by using administration applications. Furthermore, they offer little support for service integration, not providing generic support for facilitating the construction of personalization wizards and automating service personalization persistence, forcing the developer to take charge of these time-consuming and error-prone tasks for each service that must be integrated in the portal.

This paper presents the approach we are currently following in the new version of MyPersonalizer, a J2EE-based framework for engineering My portals. MyPersonalizer is being structured according to the Model-View-Controller (MVC) architectural pattern [Crupi et al 2001, Singh et al 2002], providing generic and adaptable model and controller layers (figure 1). The model layer (section 2) represents persistent objects (e.g. user registration information, page layout, service personalization, etc.) in a generic way, handles their persistence in a relational database, includes a framework to execute model actions and provides an action for each typical use case of the portal skeleton and service personalization. The controller layer (section 3) is built upon Jakarta Struts [Jakarta Struts], the "de facto" MVC framework for J2EE web applications and provides a Struts action per use case, that delegates on the corresponding model action. In order to support the construction of the portal view, rather than building a large and specific JSP tag library, controller actions are tightly integrated with Struts, so that the developer can implement the portal view as JSP pages by using Struts and JSTL tags, without inserting Java code. Personalized service responses are integrated in the portal by providing plug-ins as extensions to the controller layer.

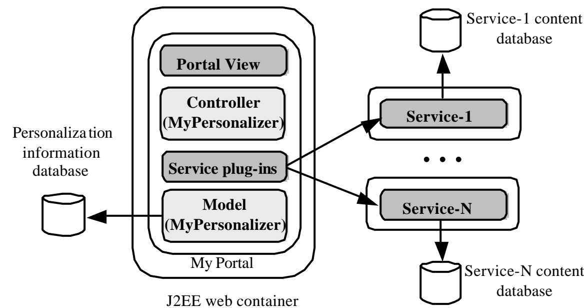

Figure 1. Architecture of a My Portal built with MyPersonalizer

2. THE MODEL LAYER

Figure 2 shows a class diagram illustrating the classes provided by the framework to model the persistent objects that must be stored for each user and the classes supporting generic property definition. `ServiceProperty` represents the personalization of a service. Since some buttons may also have persistent state associated with them, such as, buttons for maximizing/minimizing a service or showing service help information, it is also necessary to store the states of such buttons (`ServiceButtonsState`) for each service. A `WorkSpaceLayout` object represents the layout of the services in a workspace (page) and contains the keys of the `ServiceProperty` and `ServiceButtonsState` objects corresponding to the services contained in such a workspace. Similarly, a `DesktopLayout` object contains an ordered list of the keys of the `WorkspaceLayout` objects owned by a user. Finally, A `UserRegistrationInformation` object contains the user registration information and the key of his/her desktop layout.

Different services have different personalizable properties. Different portals request different information when a user signs up. Some portals only provide the user with a desktop with one workspace, 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, etc.). And finally, although most of portals have a minimize/maximize button, which state is persistent, some of them also provide other types of buttons with persistent state, such as, a button to show help information in place of the normal service response.

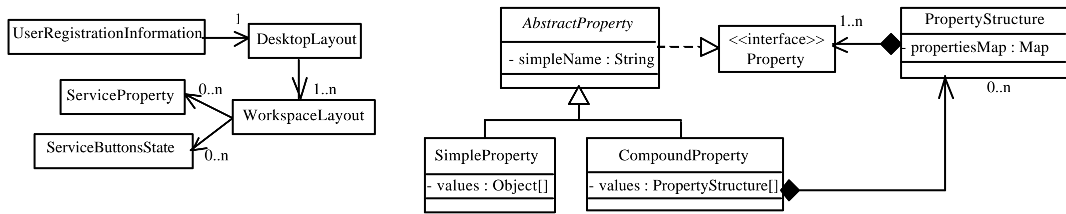

Figure 2. Personal persistent objects and support for generic property definition

So, in order to provide a generic model layer, the classes above commented must be generic enough to accommodate to any portal. Taking aside the keys, all classes must provide support for specifying a number of properties, where each property is defined by a name and a value. Figure 2 also shows a class diagram with the classes provided by the framework for property definition. Such a diagram corresponds to the application of the Composite design pattern [Gamma et al 1994]. A `Property` has a simple name and a value. As some properties may be uni-valued (e.g. number of headlines) or multi-valued (e.g. news sections), the value is modeled as an array of objects, with one element at most for uni-valued properties and zero or more for multi-valued properties. A simple property (`SimpleProperty`) is one which value is an array of objects of basic types (e.g. `Integer`, `Float`, etc.), `String` or `Calendar`. A compound property (`CompoundProperty`) is one which value is an array of property structures (`PropertyStructure`), containing a map of properties (simple or compound). All the classes modeling personal persistent objects are composed of a key and a compound property containing the rest of properties (simple or compound). For instance, the personalization of the news headlines service used as an example along this paper can be modeled with a `ServiceProperty` which compound property is composed of only one property structure, containing a simple uni-valued integer property for the number of headlines and a simple multi-valued character string property for the selected news sections.

As shown in figure 3, the model layer provides an abstract factory [Gamma et al 1994] (`RepositoryAccessorFactory`) that allows to create instances of DAOs [Crupi et al 2001] for accessing to personal persistent objects, and a default implementation for relational databases. In order to map the state of persistent objects to relational tables, this implementation requires the developer, as part of the portal configuration, to provide meta-information about the structure of properties in each type of object. Storing personal objects in a structured way allows to implement complex queries efficiently (e.g. getting the e-mails of all users fulfilling a set of conditions in terms of the personal information, for later sending them an e-mail). The framework provides a default implementation of each typical use case in a separate `Action` class that accesses the user's personal information by using the appropriate DAOs. Actions also delegate part of their job in global plug-able policies (e.g. to provide a default service personalization in function of the user registration information whenever a service is added to a workspace).

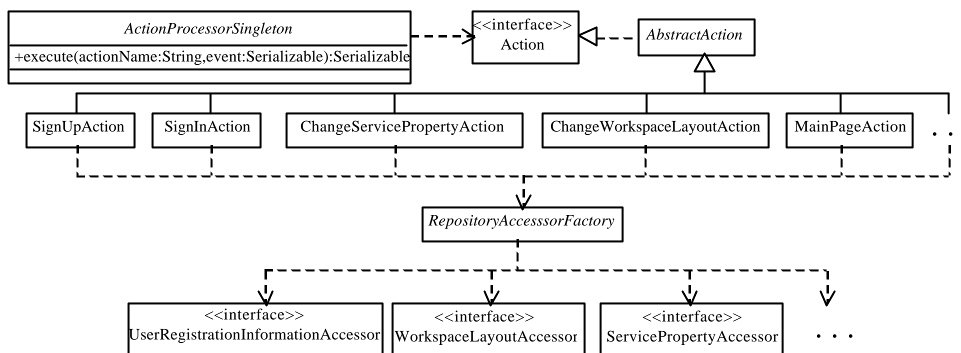

Figure 3. Implementation of use cases in the model layer

Finally, the model layer exposes a minimal interface to the controler through a plug-able facade, `ActionProcessorSingleton`, that corresponds to a variant of the Session Facade pattern [Singh et al 2002], providing only one method to execute an action by its name. The framework provides two implementations of this facade, one for local execution of actions and another one for remote execution of actions by using a stateless EJB.

3. THE CONTROLLER LAYER

The controller layer is built upon Jakarta Struts and provides a Struts action per use case. Each action accesses the HTTP request parameters, executes the parallel model action, adds necessary attributes to the request for errors or action results, and forwards or redirects to the next URL, normally a JSP page. Controller actions are tightly integrated with Struts, so that the developer can implement the portal view as JSP pages by using Struts and JSTL tags, without inserting Java code. For example, in order to implement the view of the personalization wizard corresponding to the example news headlines service, the developer writes a JSP page by using Struts and JSTL tags that generates an HTML form allowing the user to edit the number of headlines and sections. He/she also specifies in the Struts configuration file a Struts `ActionForm` (probably dynamic) to do basic validation and associates the URL in the HTML form action with the controller action `ChangeServicePropertyAction` provided by the framework. Such an action accesses the values in the `ActionForm`, and calls on the parallel model action, which modifies persistently the simple properties `numberOfHeadlines` and `sections` in the corresponding `ServiceProperty` object.

Personalized service responses are integrated in the portal by providing plug-ins as extensions to the controller layer (figure 1). Each plug-in implements an interface with an operation that receives the service personalization and the states of service buttons, and returns the personalized HTML response. A plug-in is typically implemented as a proxy of the real service, calling the service with the service personalization (at least) as parameter. Real services are normally HTTP services, returning its reply in HTML or XML, or SOAP web services. In these last two cases, the plug-in must format the reply to HTML. The controller layer provides an abstract implementation of a plug-in for facilitating the integration of services.

4. CONCLUSION

The main contribution of the approach we are following with regard to current portal servers is the generic and adaptable MVC architecture of MyPersonalizer. The framework provides a generic implementation of each typical use case in a My portal, where each use case is implemented a Struts action in the controller that delegates in a parallel model action. Both actions (in the controller or model) and global policies can be redefined if necessary. The framework also tries to be easy to use and to allow for a good separation of roles in the development team. Portal view, which represent the most time-consuming task, can be developed by graphical designers with skills on Struts and JSTL tags. The rest of tasks, mainly, integration of personalized service responses and portal configuration can be undertaken by software engineers. We hope to complete a full OpenSource version by the end of 2003. As future work, we will replace Struts integration with JavaServer Faces integration, the future standard MVC framework for J2EE web applications. We will also implement upcoming standards for integration of services in portals ([Portlet API] and [WSRP]).

ACKNOWLEDGEMENT

This work has been supported by the Spanish program CICYT (TIC2001-0547).

REFERENCES

Crupi, J., Alur, D. and Malks, D., 2001. *Core J2EE Patterns*. Prentice Hall.

Gamma, E., Helm, R., Johnson, R. and Vlissides, J., 1994. *Design Patterns: Elements of Reusable Object-Oriented Software*, Addisson-Wesley.

Jakarta Struts. http://jakarta.apache.org/struts/index.html.

Portlet API. http://www.jcp.org/jsr/detail/168.jsp.

Singh, I., Stearns, B. and Johnson, M., 2002. *Designing Enterprise Applications with the J2EE Platform, Second Edition*. Addison-Wesley.

Web Services For Remote Portals (WSRP). http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wsrp.