*Journal of Universal Computer Science, vol. 10, no. 4 (2004), 454-472
submitted: 17/10/03, accepted: 2/2/04, appeared: 28/4/04 © J.UCS*

Portlets as Web Components: an Introduction

Oscar Díaz
(University of the Basque Country, Spain
oscar@si.ehu.es)

Juan J. Rodríguez
(University of the Basque Country, Spain
jibrojij@si.ehu.es)

Abstract: Today's organisations are increasingly relying on the web to support their operations and the integration of their processes with their partners'. Portlets, which are distributed web components that encapsulate web applications, are considered a promising breakthrough towards this aim. The goal is to define a component model to enable portlets to be easily plugged into web portals. This article outlines the main challenges associated with the definition and use of portlets. As any component model, portlets should have clear interfaces so that they can be plugged into third-party applications. This includes the communication between the consumer of a portlet and the portlet container, as well as the communication between the portlet container and the portlet itself. Two standards, WSRP and JSR168, address these issues. After outlining them, we conclude with some insights into the implementation of portlets.

Key Words: portal applications, web services, component-based development, portlets
Category: D.2.2, H.4.m

1 Introduction

The increasing growth of web applications in size and complexity requires a systematic way to development, which may be facilitated by following modular engineering practices and using component-based development. Componentware is advocated as a means to manage the development of large, complex and distributed web applications effectively [Repenning et al., 2001]. Szyperski defined the term component as follows [Szyperski, 1998]:

> A unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties.

When looking for components in current web development practices, the most common artefacts are HTML pages. However, it is questionable whether an HTML page is a component or not. It can certainly be deployed independently and be subject to composition by third parties through the `href` construct, but, it is not clear what kind of unity it provides, neither its interface nor its context dependencies. Other common artefacts are servlets and applets, which can both be considered as components although their granularity is rather small and their reuse can be quite limited.

Web services can be seen as a shift from component building blocks to the assembly of services. Their interfaces are specified contractually by means of the Web Service Definition Language (WSDL) [W3C, 2001], and can be subject to composition by third parties. Most web services return raw data, and the caller is responsible for determining how to use them. This allows to reuse the business logic, but it is up to the caller to write the presentation logic, which is not reused or subject to composition by third parties. However, this is a whole layer that deals with issues like state management, error handling or navigation. Coding it is a cumbersome task and it certainly prolongs the development. Portlets, a.k.a. web parts or gadgets, address this problem.

A portlet is a multi-step, user-facing application to be delivered through a web application. They resemble windowed applications since they render markup fragments that are surrounded by decoration containing controls. Portals have embraced this technology quickly, and they are currently the most notable portlet consumers. Hence, a portal page can contain a number of portlets that users can arrange into columns and rows, and minimise, maximise, or arrange to suit their individual needs. So far however, the lack of a common model prevented portlet interoperability. This impedes a portlet developed in, say, Oracle Portal, from being deployed at a Plumtree portal, and vice versa. However, the recent delivery of the Web Services for Remote Portlets (WSRP) specification [OASIS, 2003] promises to overcome this problem. WSRP uses WSDL for portlet specification. Unlike traditional web services, portlet operations might not only return raw data, but fully rendered markup that is to be included within a portal page with very few changes. This leverage the use of web services from functional integration to application integration where user-facing, multi-step concerns are considered. The goal is to define a component model that would enable portlets to be easily plugged into web applications.

The rest of the paper is organised as follows: we introduce a portlet definition by comparing them with web services and web applications [see Section 2]; then, the WSRP and JSR168 standards are outlined [see Section 3]; later, we introduce a few hints on implementing portlets [see Section 4]; finally, we present our conclusions [see Section 5]. The aim of this article is not to provide an exhaustive description of the standards, but an overall picture about this technology.

2 Comparing Portlets with Previous Technologies

2.1 Portlets versus Web Services

Web services provide an enabling technology to deliver on the current promise of Internet-based business-to-business connectivity. Web service standards facilitate sharing business logic, but suggest that the consumers should write their own presentation layers. As an example, consider a web service that offers two operations, namely, `searchFlight` and `bookFlight`. The former retrieves flights that match some input parameters, e.g., `departureAirport` or `flightDates`, and `bookFlight` takes the

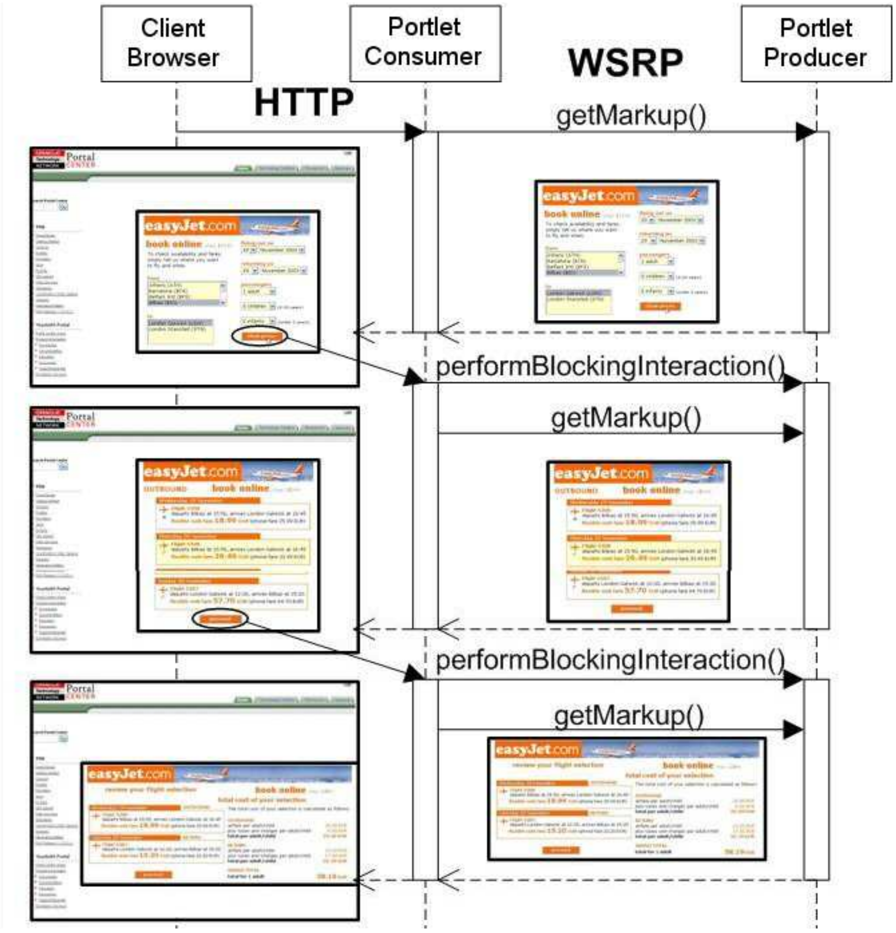

Figure 1: Interaction diagram

selected flight and payment data, and books a seat on this flight. This WSDL-based API can then be used by a consumer application. First, the application would collect the `departureAirport`, `flightDates` and other parameters with an input form within which an HTTP request might support a call to `searchFlight` which, in turn, returns a set of flights whose presentation is left to the calling application. Next, the user selects one of the flights and, with another form, the web application collects the user's information and payment data. This interaction will in turn invoke `bookFlight`. This example illustrates the traditional approach in which web services provide the business logic, and both presentation and control layers are left to the calling application.

This scenario illustrates the traditional use of web services as a function-integration technology whereby an application can invoke code in another application. However, such an approach underscores the presentation layer [Reshef, 2002]: not only addresses this layer aesthetic aspects, but a whole range of concerns like usability, state management, error handling, or navigation. Indeed, most of the aspects that characterise a good web site are related to interactive issues [Marquis, 2002], and re-creating the presentation in each front-end increases time-to-market and can jeopardise a company's corporate image. As the previous example highlights, the reconstruction of the screenshots not only involves aesthetic aspects but also leaves to the consumer application the re-composition of the workflow among the API operations. Therefore, an API-based approach as the one provided by traditional web services, falls short for complex interactive applications whose flow spans over several web pages. The presentation logic realises brand and customer experience strategies that are becoming critical business factors for a company to be ahead of their competitors. Letting the consumer application decide both how parameters are requested or results rendered back, can jeopardise the prestige of the service provider.

What is required is to leverage web services technology as an application-integration enabler? True application integration results from making one application available within the context of another, and this can also include the user interface [Wong, 2001]. Microsoft OLE objects are a case in point. For instance, this technology allows to embed an Excel spreadsheet into a Word document by dragging and dropping an icon. Once embedded, you can work on the spreadsheet from the Word document as if you were within Excel. This is the scenario that portlet proponents aim for web applications.

Consider our flight-booking application again, but delivered as a portlet now. A `flightSearch` portlet is defined to encapsulate the previous sequence of operations (multi-step) and the XHTML fragments (user-facing). This portlet can then be used as a web component to be plugged into third-party applications. [Fig. 1] shows the three actors involved, namely, the end-user, the portlet consumer and the portlet producer. What the consumer is now reusing is a whole application. First, portlet operations do not return raw data but markup, a.k.a. fragments, to be included within the portal page with very few changes. Second, all interactions with a given portlet belong to the same session, and hence, session and state maintenance should be preserved along these interactions. Although it depends on the approach, this can be the duty of the portlet producer. While in the portlet realm, the consumer is relieved from the burden of maintaining complex sessions and control flow.

2.2 Portlets versus Web Applications

The previous comparison presents portlets as full-fledged applications. However, and unlike web applications, portlets have an additional requirement: they can be subject to composition by third parties. This has two important implications: clear interfaces and configurability are needed.

Clear interfaces implies the existence of well-defined and programmatic interfaces for a portlet to be plugged into a consumer application. Moreover, interoperability advises this interface to be generic so that the caller can interact with portlets in an standardised way. This is the endeavour of the WSRP standard. Broadly speaking, the life-cycle of a portlet session begins when the first `getMarkup` request is issued [see Fig. 1]. Once the first markup is rendered, a two-phase protocol is initiated. The `getMarkup` operation retrieves the markup that corresponds to the current state of the portlet. If several `getMarkup` are requested in a row, the same markup should be returned. There is however an exception if the state of the portlet is shared with other portlets that are aggregated from the same producer. Common causes of such shared state include the use of a common backed system, e.g., database and producer-mediated data sharing. For these reasons, there is a "two-step" capability built into the protocol [OASIS, 2003]. As a result, the consumer next invokes `performBlockingInteraction` on the portlet with whose markup the end-user has interacted. This is a synchronous operation that routes the user-enacted interaction to the producer. The consumer has to wait for the response from `performBlockingInteraction` before invoking `getMarkup` on the portlet it is aggregating. The portlet will receive only one invocation of `performBlockingInteraction` per client interaction, except for retries. If this operation ends successfully, the consumer can then retrieve the next markup by invoking `getMarkup` on *all* the portlets within the portal.

As for configurability, it is well-known in the component community that, the larger the component, the more reduced the reuse. Portlets might be coarse-grained components since they encapsulate a whole application. Therefore, mechanisms should be in place to configure the portlet to the environment where the portlet is going to be "hooked on". This includes both the consumer and the end-user environments.

The consumer environment includes the user-profiles being supported by the consumer application. WSRP standardises the structure of user profiles, which has been derived from P3P User Data. Extensibility is supported in both directions: the portal indicates to the portlet producer what set of user profile extensions it supports during registration, and a portlet's meta-data declares what user profile items it uses (including any extended user profile items) [OASIS, 2003]. The consumer environment can also set the amount of page space that the portal will assign to the fragment generated by the portlet, the so-called, `windowState` property. The options WSRP supports include: `normal`, which indicates that a portlet is likely to share the aggregated page with other portlets; `minimised`, which instructs the portlet not to render visible markup, but allows it to include non-visible data such as JavaScript or hidden forms; `maximised`, which specifies that the portlet is likely to be the only portlet being rendered in the aggregated page, or that it has more space than usual; and `solo`, which denotes that a portlet is the only portlet being rendered in the aggregated page. This property is set by the portlet consumer among the values supported by the portlet producer.

Furthermore, the consumer can also be interested in ensuring a common look-and-feel across the portlet markups to be rendered in the same portal page, e.g., similar background, fonts or titles. To this end, the portlet markup should use Cascade Style Sheets (CSSs) [W3C, 1998]. The portlet returns CSS-parameterised fragments that are then processed by the portlet consumer. This process includes providing the actual values for the CSS parameters. Interoperability requires these parameters to be standardised so that the portlet consumer can always expect the same terms regardless of the producer. This is also been set by the WSRP standard.

As for the end-user environment, it traditionally refers to the client agent or browser, and includes locale information, user agent software, or bandwidth. Appropriate parameters are defined in WSRP to pass this data to the portlet producer. However, this is not enough. Portlets should not only consider the features of the client agent, but mechanism should be provided for users to determine the information that they want to see and how they would like it to be presented, i.e., portlet customisation. This is specially important in a portal setting since they are targeted at customary users. This implies that customisation can and must be achieved at a broader extent than in traditional web applications, which need to cope with more occasional users. To this end, WSRP introduces the edit mode. A mode is a way of behaving. Both the content and the activities offered by a portlet depend on its current mode. For instance, when in the view mode, the portlet renders fragments which support its functional purpose, e.g., booking a flight seat. This is what we normally mean by interacting with a traditional web application. In contrast, when in the edit mode, it provides content and logic that allows a user to customise its behaviour. Besides the edit and view modes, the WSRP standard also supports the help and preview modes that instruct a portlet to provide a help screen and to pre-render it before adding it to a portal page.

For the `flightSearch` portlet, the edit mode can be realised as the user selecting a default value for a portlet parameter, e.g., `departureAirport = Madrid`, bookmarking some form values for repetitive portlet invocations, e.g., a link labelled `flightTo-Munich` can directly lead to a fragment with most of the fields already filled in to book a ticket to visit Munich, or providing some aggregate information about previous enactments of `flightSearch`, e.g., total amount spent on flight tickets. When in the edit mode, end-user interactions are targeted at customising a portlet's functional behaviour.

3 The Architecture and the Standards

WSRP portlets lead to a distributed architecture that promotes the logical separation of portlets from portal servers that use their services [see Fig. 2]. In the traditional portal model, portlets ran on the same J2EE application server as the portal server and interact via simple J2EE inter-process communication. However, scalability issues suggest to move portlets to other machines. Furthermore, departments within an organisation often want to keep control over their own portlets, something that is hard to accomplish if portlets must be deployed to a centralised portal server. But WSRP was designed

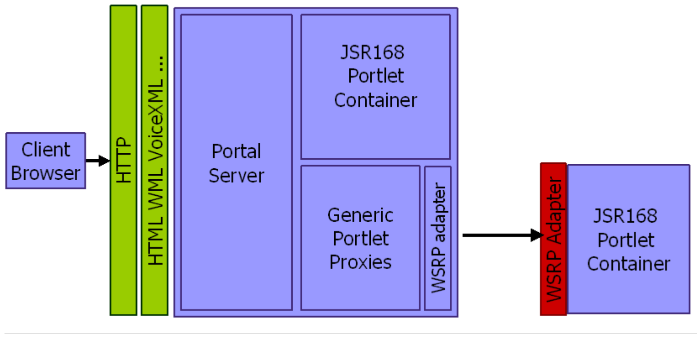

Figure 2: The architecture

Vendor	WSRP Consumer	JSR 168 Container
Plumtree Enterprise Web Suite	supported	supported
IBM WebSphere Portal 5.02	preview	preview
BEA WebLogic Portal 8.1	preview	supported
OracleAS Portal 10g	preview	preview
Sun Java System Portal Server 6.2	announced	preview
Vignette Application Portal 7.0	supported	supported
MS Office Share Point Portal Server 2003	not planned	not planned

Table 1: WSRP & JSR 168 backing

not only to allow remote portlet-to-portal communication. Even in an scenario where portlets are deployed locally, WSRP keeps its value as a platform-independent specification. Thus, a J2EE-based portal server could interoperate with a portlet running on a .Net machine as long as it exposes its functionality via WSRP-compliant web services.

Once in a portlet container, the portlets can be developed using different platforms. The JSR168 specification [Java Community Process, 2003] defines a standard set of APIs for portlets to be plugged into J2EE-based portal servers. In the same way that servlets run into a servlet container, JSR168 defines a portlet container that manages portlets. The container is the true interlocutor with the portlet consumer, and the responsible for mapping WSRP requests into JSR168 operations. This allows a portlet programmer to ignore the intricacies of WSRP. The container also offers some infrastructure for personalisation, presentation and security. Several vendors seem to be interested in supporting these standards [see Tab. 1 or `http://www.wsrp.info`].

3.1 The WSRP Standard

WSRP is a joint effort of two OASIS technical committees, namely, the Web Services for Interactive Applications (WSIA) and the Web Service for Remote Portals (WSRP) [OASIS, 2003]. WSRP sits on top of the existing web service stack and uses WSDL for defining a set of interfaces. It standardises the APIs between consumers and producers of portlets, the communication protocol, and some aspects of the component model, e.g., modes, personalisation descriptions, or CSS terms.

The 1.0 specification supports four interfaces, namely:

Service Description This interface allows consumers to ascertain both the capabilities of the producer and the portlets it hosts. The latter includes the meta-data necessary for a consumer to interact with each portlet. It defines the operation `getServiceDescription` for acquiring the meta-data of the producer.

Markup This interface allows consumers to request and interact with markup fragments. This includes the `getMarkup` operation that returns the presentation markup which corresponds to the current portlet state, as well as the `performBlockingInteraction` operation which is the means used by the consumer to route the chosen interaction to the producer. Since WSRP does not require neither the producer nor the consumer to be stateful, these operations carry the state necessary for the portlet to render the current markup to be returned to the consumer. If the producer uses local state, then it will return a session identifier.

Registration A registration shows a relationship between a consumer and a producer. It can include how the service is going to be charged or book-keeping modalities. This optional interface allows consumers to register, unregister and modify this relationship information. The portlet functionality can be dependent on whether a consumer is registered or not.

Portlet Management This optional interface allows consumers to have access to portlet states and property information. It includes operations for getting portlet metadata, i.e., `getPortletDescription`, cloning portlets for further customization, and setting/getting portlet properties.

According to the specification, producers are presentation-oriented web services that host portlets. Hence, we model a WSRP producer as a compound of portlets [see Fig. 3]. Both producer and portlet class attributes correspond to the meta-data as retrieved by the `getServiceDescription` and `getPortletDescription` methods. As an example of a producer meta-data, consider `requiresRegistration`. This field is a boolean which indicates whether or not the producer requires the consumer to be registered previously. On the other hand, `PortletHandle` is a portlet attribute. A handle serves to uniquely refer to the portlet at hand. Finally, a `clones` association is introduced to indicate the association between a portlet and its clones.

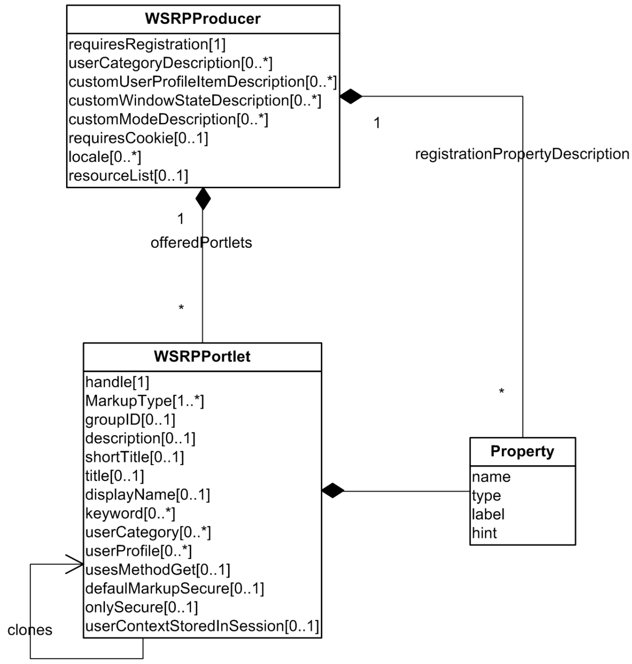

Figure 3: WSRP description concepts

Both producers and portlets can have properties to share information between a consumer and a producer. Properties of the producer, e.g., `billingMethod`, are set at registration time and influence all of the portlets. In contrast, properties of the portlet are set when a portlet clone is created. A property has a name, a type, a label, i.e., a short, human-readable name which is used for display and administering purposes, and a hint, i.e., a short description to be displayed as a tip when the property is edited.

[Fig. 4] depicts the model for our running example. Attributes of the meta-model are mapped as tagged values on the model, and properties are modelled as attributes. The example shows an EasyJet producer that offers the `FlightSearch000` portlet. Prior registration is not required (`requiresRegistration = false`). This portlet has a set of tagged values that indicates meta-data about the returned markup. Some examples follow: the supported mime types, e.g, `text/xhtml`, the window states supported for each returned mime type (`markupType = ...`), a brief description of the functionality (`description = ...`), a set of keywords for searching (`keywords...`), a flag that indicates that the generated markup includes the method `get` in an HTML form (`usesMethodGet = ...`), whether the portlet requires secure communication on its default markup (`onlySecure = ...`) and so on. [Fig. 4] also shows how `Flight-Search000` is cloned into the `FlightSearch111` portlet. This allows the initial portlet

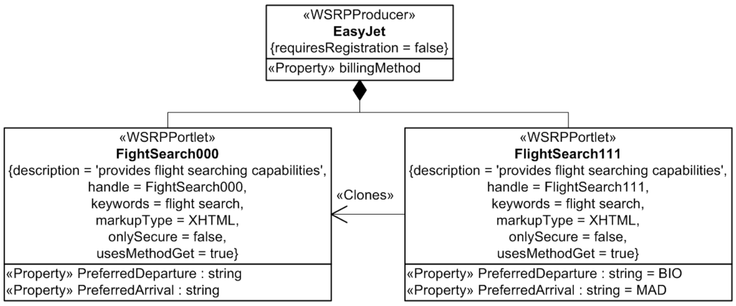

Figure 4: Cloning `flightSearch`

to be configured to fit the idiosyncrasy of a specific portal. Notice that this is consumer-based customisation, i.e., it applies to all invocations made through this consumer, not to be confused with the user-based customisation which is made through the edit mode. Only properties can be configured.

3.1.1 A Conversation Between the Portlet Consumer and the Portlet Producer

1. The consumer finds out about the producer: This amounts to the consumer getting the producer's meta-data with its description of the registration requirements, and, possibly, the list of portlets offered by the producer. A snippet of the returned `ServiceDescription` structure follows:

```
<ServiceDescription xmlns="urn:oasis:names:tc:wsrp:v1:types">
    <requiresRegistration>true</requiresRegistration>
    <requiresInitCookie>none</requiresInitCookie>
    <offeredPortlets>
        <portletHandle>FlightSearch000</portletHandle>
        <markupTypes>
            <mimeType>text/html</mimeType>
            <modes>wsrp:view</modes>
            <modes>wsrp:edit</modes>
            <windowStates>wsrp:normal</windowStates>
            <windowStates>wsrp:solo</windowStates>
        </markupTypes>
        <title lang="us">
            <value>FlightSearch</value>
        </title>
        <description lang="us">
            <value>...</value>
        </description>
    </offeredPortlets>
</ServiceDescription>
```

The `getServiceDescription` operation provides a discovery means for a consumer to ascertain the producer's capabilities. In this example, these characteristics include: registration is required, no cookies are used, a `FlightSearch000` portlet is available that uses `text/html` as the mime type, and so on.

2. The consumer registers with the producer: If registration is permitted, the consumer can state some of its characteristics at this time. For instance, it can restrict the modes or window states it is willing to manage by executing `register` with the following parameters [see Section 7.1.1 of the WSRP specification]:

```
<RegistrationData xmlns="urn:oasis:names:tc:wsrp:v1:types">
    <consumerName>myCompany_Portal</consumerName>
    <consumerAgent>
        ...
    </consumerAgent>
    <methodGetSupported>true</methodGetSupported>
    <consumerWindowStates>wsrp:normal</consumerWindowStates>
</RegistrationData>
```

In the example, the consumer, `myCompany_Portal`, indicates that `normal` is the only window state supported.

3. The consumers find out about the portlets being offered by the producer with which it registered: Through `getServiceDescription`, the consumer knows the set of portlets offered by the producer [see step 1]. Once the consumer is registered, `getPortletDescription` can be used to obtain detailed information about these portlets, along with the preferences set during registration. The answer is a `getPortletDescriptionResponse` document like the following:

```
<getPortletDescriptionResponse>
    <portletDescription>
        <portletHandle>FlightSearch000</portletHandle>
        <markupTypes>
            <mimeType>text/html</mimeType>
            <modes>wsrp:view</modes>
            <windowStates>wsrp:normal</windowStates>
        </markupTypes>
        <title lang="us">
            <value>FlightSearch</value>
        </title>
    </portletDescription>
</getPortletDescriptionResponse>
```

4. The consumer finds out about the properties available to configure the portlet: By this time, the consumer knows the configuration options available. Now, it discovers the properties through which it can set the corresponding amendments. To this end, `getPortletPropertyDescription` returns the following document:

```
<getPortletPropertyDescriptionResponse ...>
    <modelDescription>
        <propertyDescriptions name="preferredDeparture"
                              type="types:AirportsType"/>
        <propertyDescriptions name="preferredArrival"
                              type="types:AirportsType"/>
        ...
    </modelDescription>
    <modelTypes>
        <!-- XMLSchema Type definitions for the properties -->
        <xsd:schema ... >
            <xsd:simpleType name="AirportsType">
                <xsd:restriction base="xs:NMTOKENS">
                    <xsd:enumeration value="BIO"/>
                    <xsd:enumeration value="MAD"/>
                </xsd:restriction>
            </xsd:simpleType>
            ...
        </xsd:schema>
    </modelTypes>
</getPortletPropertyDescriptionResponse>
```

This document states that portlet `FlightSearch000` supports a set of properties whose types indicate the range of values available.

5. The consumer requests a unique configuration of one of the portlets offered by the producer (provided it supports cloning): To this end, the consumer first creates a clone of the desired portlet by executing `clonePortlet`. This operation returns a handle that identifies the portlet clone. Once the clone is created, the consumer configures the clone either through a configuration page or programmatically by setting properties with `setPortletProperty`. The following snippet illustrates the input parameter of this operation:

```
<setPortletProperties>
    <registrationContext>
        <registrationHandle>...</registrationHandle>
    </registrationContext>
    <portletContext>
        <portletHandle>...</portletHandle>
    </portletContext>
    <userContext>
        <userContextKey>...</userContextKey>
    </userContext>
    <propertyList>
        <property name="preferredDeparture">BIO</property>
        <property name="preferredArrival">MAD</property>
    </propertyList>
</setPortletProperties>
```

In this example, the consumer configures the `FlightSearch000` clone by setting the `preferredAirport` to `BIO` and the `preferredArrival` to `MAD`. From now on, the clone will provide markups tune to the consumer's amendment behaviour.

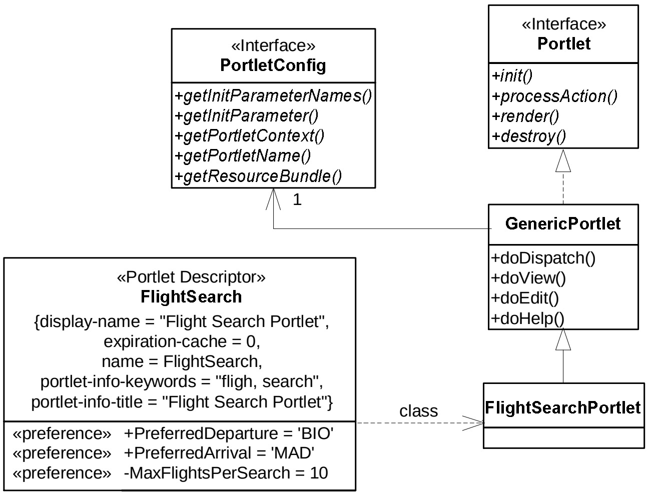

Figure 5: Portlet deployment descriptor model

6. The consumer receives a markup from the producer: This is achieved through the `getMarkup` function. A snipped of the parameter returned follows:

```
<getMarkupResponse ...>
   <markupContext>
      <mimeType>text/xhtml</mimeType>
      <useCachedMarkup>false</useCachedMarkup>
      <markupString><table>...</table></markupString>
   </markupContext>
   ...
</getMarkupResponse>
```

The `<markupString>` tag contains the XHTML.

3.2 The JSR168 Standard

The Java Standardization Request 168 is a Java Community Process initiative to standardise the way portlets are developed within the J2EE framework. It is defined as a set of extensions to the Java Servlets API in which three actors are involved [see Fig. 5]:

Portlet Entity This is viewed as a new web component.

Portlet Application In a J2EE architecture, a web application is an aggregation of JSP pages and servlets that are packaged into a WAR archive with a `web.xml` deployment descriptor. Likewise, a portlet application is a web application that includes portlet components and a `portlet.xml` deployment descriptor that holds all the meta-information the portlet container needs to run it. Hence, a portlet application has both a `web.xml` file and a `portlet.xml` file. [Fig. 6] shows a `portlet.xml` sample file that specifies the mode, the view, the portlet preferences, and so on.

Portlet Container Portlets are managed by portlet containers. They communicate by means of an interface that includes `processAction`, which is intended to process input from a user action, and `render`, which is called whenever a portlet is redrawn. The latter can change its behaviour and output depending on the portlet mode. To this end, this interface is implemented by the `GenericPortlet` class that implements the `render` method, and delegates the call to more specific methods which handle the features of the current mode, e.g., edit, view.

JSR168 portlets can set and get transient data in the following scopes: (i) they can include data such as parameters and attributes in the request; (ii) they can have session data with either global or portlet scope (the former allows other portlets of the same portlet application in the same user session to have access to these data, whereas the latter is private); (iii) they can have context data that include custom portlet modes, custom window states and WSRP user attributes, which allows to share information among portlets deployed on the same portlet application independently from the user session.

Furthermore, JSR 168 considers that the container should support the storage of persistent properties that allow specific portlets to store the personalisation of each end-user. To this end, a portlet can store configuration and personalisation options as portlet preferences. The portlet can define which data the user is allowed to change when in the edit mode, e.g., `departureAirport`, or which parameter can only be changed by an administrator when in configuration mode, e.g., the ticket reservation system to be used. The preferred mode to write preferences is the edit mode, which provides the user with a customisation screen. Nevertheless, preferences can be set programmatically only during `processAction` enactment, and read during the execution of `render`. Preferences can be pre-set with default values in the deployment descriptor (`portlet.xml`).

Last but not least, the JSR168 specification also include a JSP tag library to help displaying portlet pages with JSP technology. For instance, a custom JSP tag can automatically declare a portlet request and response objects so that they can be used within a JSP page. Other JSP tags can help construct URLs that refer back to the same portlet.

There is a close correspondence between JSR 168 concepts and WSRP concepts, namely [Hepper and Hesmer, 2003]: (i) Portlet modes and window states are the same; (ii) URL encoding and URLs that point to a portlet are the same; (iii) both standards use a two-phase protocol with action and render phases; (iv) both standards store transient state across requests using sessions; (v) storing persistent state to personalise a portlet's rendering is achieved with properties of arbitrary types in WSRP, whereas JSR168 supports preferences of type string or string array; (vi) WSRP registration data is represented as a `PortalContext` object in JSR168.

```
<portlet-app>
   <portlet>
      <portlet-name>FlightSearch</portlet-name>
         <portlet-class>
            org.atarix.FlightSearchPortlet
         </portlet-class>
         <expiration-cache>0</expiration-cache>
         <supports>
            <mime-type>text/html</mime-type>
            <portlet-mode>view</portlet-mode>
            <portlet-mode>edit</portlet-mode>
         </supports>
         <portlet-info>
            <title>Flight Search Portlet</title>
            <keywords>flight, search</keywords>
         </portlet-info>
         <init-param>
            <name>MaxFlightsPerSearch</name>
            <value>10</value>
         </init-param>
         <portlet-preferences>
            <preference>
               <name>PreferredDeparture</name>
               <value>BIO</value>
            </preference>
            <preference>
              <name>PreferredArrival</name>
              <value>MAD</value>
            </preference>
         </portlet-preferences>
     </portlet>
     ...
  <user-attribute>
     <description>PreferredTimeZone</description>
     <name>timezone</name>
  </user-attribute>
  <user-attribute>
     <description>PreferredLocale</description>
     <name>locale</name>
  </user-attribute>
</portlet-app>
```

Figure 6: A `portlet.xml` sample file

4 Portlet Implementation

Portlet implementation can be an important issue since it encapsulates not only the business logic, but also the code to implement navigation, including state preservation among interactions. A portlet can be seen as a state machine in which the states represent the rendering of a fragment, and the transitions stand for the actions to be executed. These actions are enacted by the end-user when he or she interacts with the fragments.

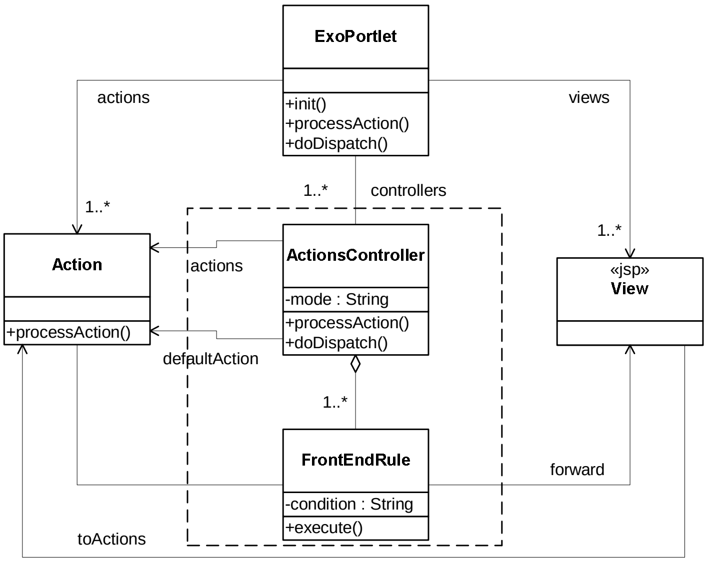

Figure 7: The eXo architecture for portlet implementation

Therefore, a portlet supports a set of end-user interactions. Contrarily to servlets, a separate portlet class to handle each end-user interaction cannot be defined since there is only one service method responsible for handling all HTTP requests. Consequently, this single portlet needs to determine which action is being requested, how to process it, to which state the portlet needs to transit, and what to render back to the user.

Without appropriate decoupling patterns, this code can make the separate evolution of each concern a cumbersome problem. The eXo platform, for instance, uses MVC and introduces three notions [see Fig. 7]: action, that performs `processAction` to attain a change of state, view, which corresponds to JSP files that embed enactments to the corresponding actions, and a controller that links action outcomes to views. An XML document is used to describe this controller [see Fig. 8]. The controller is composed of a set of `<action>` tags that are associated with classes, e.g., `exo.flightSearch.InitSeach`. Each possible outcome of the enactment of this class is described by means of a separate `<forward>` tag that has a name that matches one of the outcomes, and a page that indicates the next view to be rendered as a result of this outcome. Recall that portlets follow a two-phase process where `processAction` and `doDispatch` (an operation invoked by `render`) are interwoven: first, the end-user interacts with the fragment of the portlet being presented, and the associated action is triggered; in turn, this can lead to a change in the state of the portlet; second, the portlet produces the next view based on its current state, which should not change.

```
<controllers>
    <name>FlightSearchMVC</name>
    <identifier>FlightSearchMVC</identifier>
    <view-controller>
        <default-action>ShowFirstViewScreen</default-action>
        <action name="ShowFirstViewScreen"
                class="exo.flightSearch.InitSearch">
            <forward name="success" page="flight_1_Search.jsp"/>
            <forward name="error" page="error.jsp"/>
        </action>
        ...
    </view-controller>
    ...
</controllers>
```

Figure 8: Controller sample for the eXo platform

Besides, the use of a pattern that facilitates the development and maintenance of the portlet implementation, other considerations include the following:

HTML considerations A portlet generates markup fragments to be framed by the consumer application. Specifically, the portlet fragment will become the content of an HTML table cell. This restricts the HTML generated by the portlet to any HTML tag that can be included in an HTML `<body>` tag.

Environment considerations Although a portlet can be invoked and its output readily presented to the end-user, in most cases, the portlet will run within a framework provided by the consumer. Portal frameworks are a case in point since they provide a number of functions to obtain user-profiling information or session data.

Presentation considerations Markups produced by different portlets that are rendered together should have common look-and-feel features. Hence, portlet markups might use CSS tags rather than hard-coding actual values into the HTML code. In this case, the consumer should provide the actual values of the CSS tags beforehand.

5 Conclusion

Portlets can leverage existing web application development with the benefits of componentware. The recent delivery of WSRP and JSR168 will certainly facilitate a market for portlets in the medium run that will make the Internet a marketplace of visual web services, i.e., portlets, ready to be integrated into portals as stated in the WSRP proposal. The broad support among portal vendors will certainly fuel the movement towards application syndication as the next wave following the successful use of content syndication in current web applications.

But, there is still work to be done. The items below are planned for inclusion in the upcoming 1.1 and 2.0 versions of WSRP. According to [Anuff, 2004], the 1.1 version is scheduled to be completed in 2004, and 2.0 is likely to be ready by the end of 2005 or the first half of 2006:

- Security was not addressed in the 1.0 specification, except for allowing a portlet to be marked to require a secure connection. The current plan is to incorporate support for WS-Security along with guidelines on how it should be used within WSRP to ensure interoperability among vendors. In version 1.0, it is left up to each individual vendor to determine how to attempt to secure WSRP usage. This will potentially lead to a number of issues when trying to interoperate between vendors and address application security. Support for this is planned in version 2.0.

- UDDI support will allow producers to post information about their services on UDDI servers to make it easier for consumers to search for and find their offerings when the location of the host is not known. Version 1.1 will add simple support of UDDI so that a producer can describe its presence as well as each of the services it offers. This is essentially a subset of the data that is found in the service description. Version 2.0 is expected to introduce more detailed structures to provide support for categorisation, too.

- Portlet integration is more than merely rendering their outputs together. Indeed, much of the value of portals lies in providing a coordinated and seamlessly environment for the end-user to interact with distinct, otherwise detached, applications. WSRP 2.0 plans to provide a mechanism that allows portlets to broadcast event information to other portlets if required. The key use case for this feature is so that portlets can post contextual information about their interaction, and other portlets can use it to tailor the content that they generate.

These issues illustrate that portlet technology, although promising, is still in its infancy. However, the interest of most portal vendors in making their offerings compliant with the standards, suggests that portlets are here to stay.

Acknowledgment

This work was partially supported by the Spanish Ministry of Science and Technology (MCYT) under contract TIC 2002-01442, and the University of the Basque Country under contract UE02/A16.

References

[Anuff, 2004] Anuff, E. (2004). WSRP and the Enterprise Portal. `http://www.sys-com.com/story/print.cfm?storyid=44674`.

[Hepper and Hesmer, 2003] Hepper, S. and Hesmer, S. (2003). Introduction to Portlet Specification. http://www.javaworld.com/javaworld/kw-08-2003/jw-0801-portlet_p.html.

[Java Community Process, 2003] Java Community Process (2003). JSR 168 portlet specification. http://www.jcp.org/en/jsr/detail?id=168.

[Marquis, 2002] Marquis, G. (2002). Application of traditional system design techniques to web site design. *Information and Software Technology*, 44(9):507–512.

[OASIS, 2003] OASIS (2003). Web Service for Remote Portals (WSRP) Version 1.0. http://www.oasis-open.org/commitees/wsrp/.

[Repenning et al., 2001] Repenning, A., Ioannidou, A., Payton, M., Ye, W., and Roschelle, J. (2001). Using components for rapid distributes software development. *IEEE Software*, 18(2):38–45.

[Reshef, 2002] Reshef, E. (December 2002). Building Interactive Web Services with WSIA & WSRP. *Web Services Journal*, pages 2–6.

[Szyperski, 1998] Szyperski, C. (1998). *Component Software. Beyond Object-Oriented Software*. Addison-Wesley.

[W3C, 1998] W3C (1998). Cascading Style Sheet (CSS). http://wwww.w3c.org/Style/CSS/.

[W3C, 2001] W3C (2001). Web Services Description Language(WSDL) 1.1. http://www.w3c.org/TR/wsdl.

[Wong, 2001] Wong, S. (2001). Web Services: The Next Evolution of Application Integration. http://e-serv.ebizq.net/wbs/wong_1.html.