Turning Portlets into Services: The Consumer Profile

Oscar Díaz
oscar.diaz@ehu.es

Salvador Trujillo
struji@ehu.es

Sandy Pérez
sandy-perez@ikasle.ehu.es

ONEKIN Research Group
University of the Basque Country
San Sebastián, Spain

ABSTRACT

Portlets strive to play at the front end the same role that Web services currently enjoy at the back end, namely, enablers of application assembly through reusable services. However, it is well-known in the component community that, the larger the component, the more reduced the reuse. Hence, the coarse-grained nature of portlets (they encapsulate also the presentation layer) can jeopardize this vision of portlets as reusable services. To avoid this situation, this work proposes a perspective shift in portlet development by introducing the notion of *Consumer Profile*. While the user profile characterizes the end user (e.g. age, name, etc), the Consumer Profile captures the idiosyncrasies of the organization through which the portlet is being delivered (e.g. the portal owner) as far as the portlet functionality is concerned. The user profile can be dynamic and hence, requires the portlet to be customized at runtime. By contrast, the Consumer Profile is known at registration time, and it is not always appropriate/possible to consider it at runtime. Rather, it is better to customize the code at development time, and produce an organization-specific portlet which built-in, custom functionality. In this scenario, we no longer have a portlet but a family of portlets, and the portlet provider becomes the "assembly line" of this family. This work promotes this vision by introducing an organization-aware, WSRP-compliant architecture that let portlet consumers registry and handle "family portlets" in the same way that "traditional portlets". In so doing, portlets are nearer to become truly *reusable* services.

Categories and Subject Descriptors

D.2.13 [**Reusable Software**]: Domain engineering

General Terms

Design, Standardization

Keywords

SOA, portlets, product lines, adaptability, portals, WSRP

1. INTRODUCTION

Portlets are presentation-oriented Web Services which are packed to be delivered through third-party Web applications (e.g. a portal). Portlets are user-facing (i.e. return markup fragments rather than data-oriented XML) and multi-step (i.e. they encapsulate a chain of steps rather than a one-shot delivering). So far, portlets are mainly used as a modularization technique to structure portal content. However, their ability to be delivered through other Web applications, make portlets be the enablers of service-oriented architectures (SOAs) but now at the front end.

From this perspective, portlets strive to play at the front end the same role that Web services currently enjoy at the back end, namely, enablers of application assembly through reusable services. On the portlet case, the difference stems from what is being reused (i.e. which includes the presentation layer) and where is the integration achieved (i.e. at the front end).

This SOA scenario first requires portlet interoperability, whereby portlets developed in, lets say, Oracle Portal, can be deployed at a Plumtree portal, and vice versa. The *Web Services for Remote Portlets* (WSRP) specification [20] brings this interoperability by providing a protocol that decouples portlet providers from portlet consumers. This provides the infrastructure to make feasible a portlet market *à la COST* so that portals can deliver portlets being provided by third parties. Indeed, the *Open Source Portlet Repository Project* has been launched in 2006 to foster the free and open exchange of portlets. The *Portlet Repository* is "*a library of ready-to-run applications that you can download and deploy directly into your portal with, in most cases, no additional setups or configurations*" [4]. Other similar initiatives include *Portlet Swap* (jboss.org) and *Portlet Exchange* (portletexchange.com).

However, this SOA scenario not only requires portlet interoperability (through WSRP) and portlet dissemination (through standard repositories) but also **portlet variability**. Portlets tend to be more coarse-grained than traditional Web services since they encapsulate the presentation layer as well as the functional layer. These coarse-grained components have less chances to be reused and this can jeopardize the vision of portlets as reusable services.

Variability implies two main questions, namely, what can vary and when is this variation considered. The *what* side captures the diversity of the settings where a portlet might be consumed (i.e. the context). Web applications are increasingly becoming context aware, making them ubiquitous with respect to time, location, device or user profiles (see [16] for an overview). Portlets are Web applications, so these aspects are applicable here. Additionally, and unlike "traditional" Web applications, portlets are delivered through third-party applications, and this introduces a new context, **the Consumer Profile**. This Consumer Profile includes not only the consumer's platform (e.g. Oracle Portal, WebSphere, eXo, etc) but also presentation and functional requirements posed by the portal owner that needs to be catered for by the portlet producer.

Besides *what* is the context, we should also consider *when* should this context be appraised to customize the portlet. At this respect, it is most important to distinguish between adaptability and extensibility. *Adaptability* gives us the ability to adapt a component to different requirements *without changing the code base* (i.e without writing code). Adaptability is built into the services which care for the context automatically (adaptive applications) or semi-automatically through user intervention (adaptable applications). By contrast, extensibility techniques introduce additional code to extend and change a software component to support a specific "custom" behavior.

Portlet development standards (e.g. JSR168) account for adaptability by accessing and storing persistent configuration (a.k.a. initialization parameters), customization data (a.k.a. portlet preferences) and user profile parameters whose values are provided by the portal at runtime. However, the Consumer Profile frequently implies extensions on new markups, controllers or persistent data that would be very cumbersome to develop and, most important, maintain from a single block of code using adaptability approaches to custom dynamically the code to the current profile.

This situation can be better served by extensibility techniques where additional code is introduced to extend the base portlet.

This new scenario where portlets can be extended as well as adapted, changes the role of the portlet provider. Currently, the portlet provider is just a container of end portlets. By contrast, now portlets can be generated on consumer registry, and the portlet provider becomes a portlet assembly line (a.k.a. software product lines).

This work introduces an architecture for portlet product lines and reports on the implications for the WSRP protocol. We do not address here the development of portlet product lines but the implications for WSRP. The architecture has been realized using eXo [9] as the portal IDE (Integrated Development Environment), and WSRP4Java as the portlet provider [10].

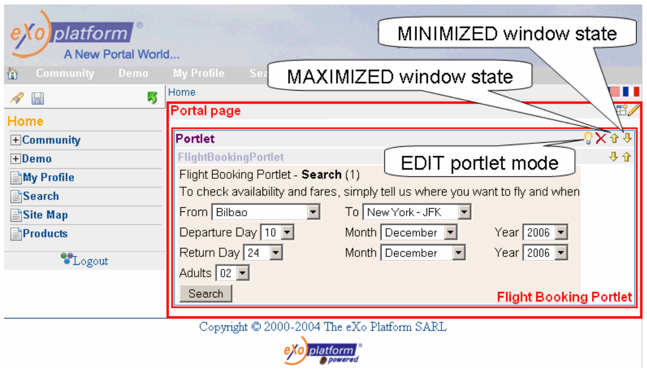
The rest of the paper is structured as follows. Section 2 provides basic background on portlets. Section 3 and 4 motivate the issue by addressing the subject of variations and the time of variations with the help of an example. Section 5 outlines how to handle those variations using product-line techniques. The main contribution of the paper rests on Section 6 that introduces a "portlet-line architecture" using WSRP. Some conclusions end the paper.

2. BACKGROUND

Web service standardization efforts facilitate the sharing of the business logic, but suggest that Web service consumers should write a new presentation layer on top of the business logic. 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*, *flightDates* and so on), while *bookFlight* takes the 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 via an input form. Within the form, 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, through 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 where Web services provide the business logic, and both presentation and navigation strategies are left to the calling application.

But what if now we want to re-use the whole application, i.e. the business logic as well as the presentation and navigation code? It is worth noticing that presentation and navigation realization are very time consuming activities that convey costly marketing strategies that companies are interested in capitalizing when their services



Flight Booking Portlet - **Search** (1)
To check availability and fares, simply tell us where you want to fly and when
From: Bilbao To: New York - JFK
Departure Day: 10 Month: December Year: 2006
Return Day: 24 Month: December Year: 2006
Adults: 02

Figure 1: The *flightSearch* portlet.

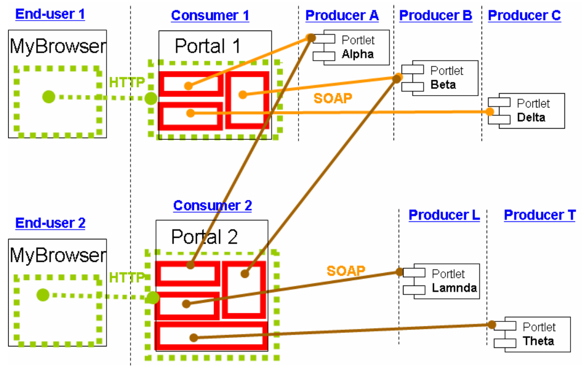


Figure 2: The portlet architecture.

are offered through third-party Web applications. So far, most SOA approaches achieve integration at the back end. Portlets open the door to achieve similar gains but now through front end integration.

Let's go back to the flight-booking sample, but now delivered as a portlet. A *flightSearch* portlet is defined that encapsulates not only the business logic but also the navigation and presentation realizations. Unlike the traditional Web-service approach, now the consumer of *flightSearch* re-uses both the presentation and the navigation. As for the presentation, portlet operations are still WSDL compliant, but now their XML results might convey not only raw data but rendering markup such as XHTML (known as "fragments" in the portlet parlance). This XHTML fragment is ready to be included within the consumer page. As for the navigation, now all interactions with a given portlet belong to the very same session, and hence, session and state management should be preserved along these interactions. Although different approaches exists, this can be the duty of the portlet producer, and hence, the consumer is relieved from the burden of complex and intricate session management and control flow. Figure 1 shows the *flightSearch* portlet when offered through a portal.

Portlets rest on two main standardization efforts: WSRP [20] and JSR168 [14]. WSRP standardizes the interfaces of the Web services a portlet producer must implement to allow another application (typically a portal) to consume its portlets. As for JSR168, it is a *Java Community Process* that standardizes an API for implementing local, WSRP-compatible portlets. Java portlets run in a portlet container, a portal component that provides portlets with a runtime environment. Therefore, the main actors involved are the WSRP consumer, the WSRP producer, the portlet and the browser agent (see figure 2).

The interaction among these actors goes as follows. First, portlet registration is achieved by the portal administrator normally through a portal IDE (e.g. Oracle Portal, WebSphere, etc), and ends up with a portal being registered to a given portlet producer. Figure 3 outlines the protocol. First, an introductory description of the producer is obtained through *getServiceDescription()*. If registration is required then, consumers must register with a producer before accessing any of the producer's portlets. Once registered, the consumer queries again the producer but now, a detailed description of the available portlets is returned. With all this information, the portal IDE creates a *WSRP consumer*[1].This *WSRP consumer* is within the portal realm.

Once registered, the portal is ready to engage the portlet in conversation to deliver its service. This is achieved through a two-step protocol (see figure 3). To begin with, the very first markup realizing the service is obtained through *getMarkup()*. The returned markup is aggregated to other markup that built up the portal page which is finally rendered to the end user. Whenever the user clicks on a link of the portlet markup, the portal receives the HTTP request which is in turn, forwarded to the portlet producer (by means of the *performBlockingInteraction()*) till it finally reaches the portlet itself. As a result, the portlet can change its state[2]. But no markup is returned to the consumer. This requires the consumer to issue a *getMarkup()* to recover the eventually new markup associated with this new state.

According to the JSR168 specification, a portlet should render different content and perform different activities depending on the current context. Part of this context is the *portlet mode*. A portlet mode is a way of behaving. 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 usually mean by interacting with a traditional Web application. Other modes include the "edit" mode, where the portlet provides content and logic that let a user customize the behavior of this portlet; the "help" mode, where a portlet may provide help screens that explain the portlet purpose, and its expected usage, and, finally, the "preview" mode, which serves to previsualize the portlet before adding it to a portal page. Other non-standard modes include the "config" mode which can be used during configuration to set the appropriate parameter values.

The mode example illustrates how portlets can adapt their behavior to the current context. Besides the mode, this context includes the so-called window state (i.e. the space available for portlet rendering), the user profiles[3], the browser agent, the consumer portal and additional portlet-specific data collected as portlet preferences.

A portlet preference is a named piece of string data. As an example, go back to our *flightSearch* portlet. Its preferences can include *arrivalAirport* with values "San Sebastián", "London" or "New York", and *departureAirport* with values "Madrid". These preferences offer a parameterization-based mechanism to adapt the portlet (in this case, the input forms). These preferences can be changed at configuration time (by the portal administrator) or at enactment time. In this latter case, the values can be automatically set by the portlet itself based on the user profile (adaptive approach) or prompting the current user through the "edit" mode (adaptable approach).

[1]An extended practice is to support the *WSRP consumer* as a local portlet that acts as a proxy to the producer. In WSRP4Java this portlet has two preferences, the producer and the portlet.

[2]This state can be shared with other portlets of the same producer. Therefore, an interaction with a portlet can result in changes in distinct portlets. This is the rational behind this *two-phase* protocol.

[3]User Information Attributes Names are derived from the Platform for Privacy Preferences 1.0 (P3P 1.0) by OASIS where attributes are described such as *user.name.given*, *user.business-info.telecom.telephone.intcode* and the like.

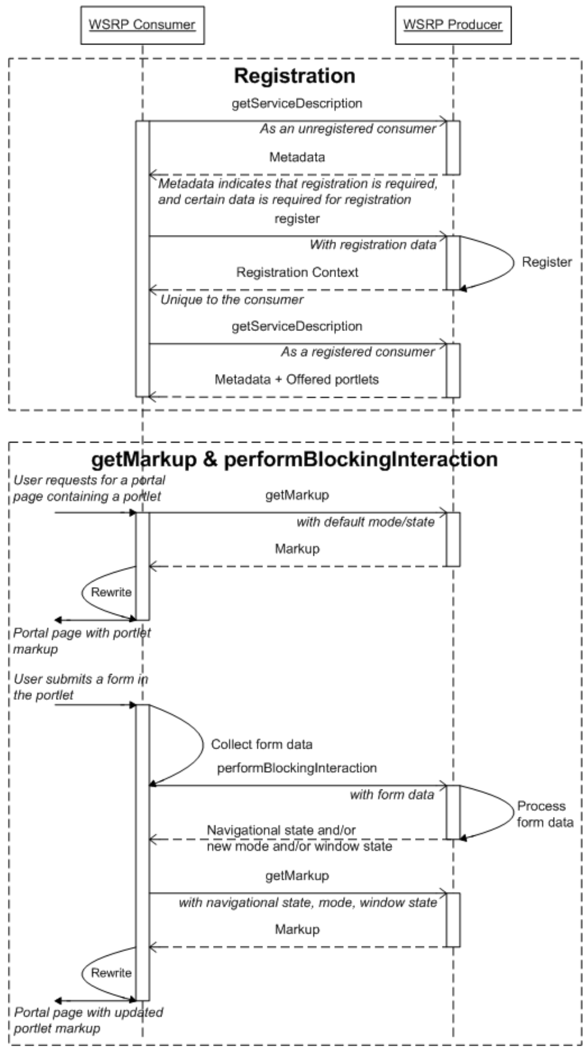
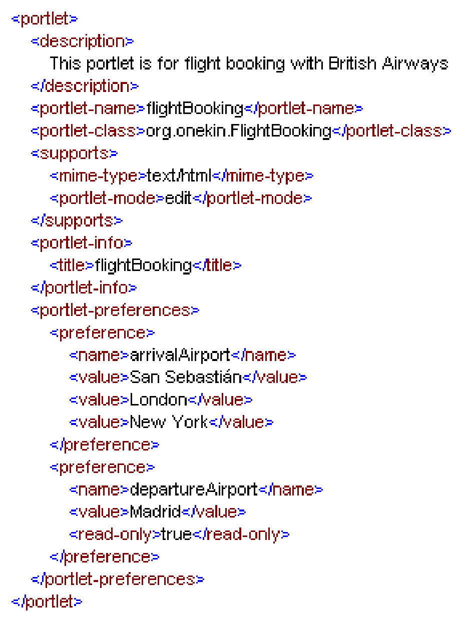

WSRP Consumer
WSRP Producer
Registration
getServiceDescription
As an unregistered consumer
Metadata
Metadata indicates that registration is required, and certain data is required for registration
register
With registration data
Register
Registration Context
Unique to the consumer
getServiceDescription
As a registered consumer
Metadata + Offered portlets
getMarkup & performBlockingInteraction
User requests for a portal page containing a portlet
getMarkup
with default mode/state
Markup
Rewrite
Portal page with portlet markup
User submits a form in the portlet
Collect form data
performBlockingInteraction
with form data
Process form data
Navigational state and/or new mode and/or window state
getMarkup
with navigational state, mode, window state
Markup
Rewrite
Portal page with updated portlet markup

Figure 3: WSRP Protocol

The deployment descriptor *portlet.xml* holds this information. Figure 4 shows a snippet for our sample case that states that *arrivalAirport* can be set as a preference by the user at execution time through the *edit* mode, whereas the *departureAirport* (read-only) can only be set by the administrator at configuration time.

Previous paragraphs describe the current situation. Variability wise, adaptability techniques are provided to tune portlet behavior to the user profile and preferences. However, SOA poses more stringent demands on portlet variability. SOA promotes a vision where distinct services collaborate to achieve a more complex offering. But collaboration is not only a matter of interoperability. Of course, standards are needed in order to define protocols that permit services from different providers to interact. But this is not enough. Collaboration often implies to adapt the service to fit into the big picture. The more adaptive is the service, the higher the changes to participate in a SOA. However, portlets are coarse-grained, and this can jeopardize the vision of portlets as reusable services.

Next sections delve into the what, when and how issues posed by portlet variability.



```
<portlet>
  <description>
    This portlet is for flight booking with British Airways
  </description>
  <portlet-name>flightBooking</portlet-name>
  <portlet-class>org.onekin.FlightBooking</portlet-class>
  <supports>
    <mime-type>text/html</mime-type>
    <portlet-mode>edit</portlet-mode>
  </supports>
  <portlet-info>
    <title>flightBooking</title>
  </portlet-info>
  <portlet-preferences>
    <preference>
      <name>arrivalAirport</name>
      <value>San Sebastián</value>
      <value>London</value>
      <value>New York</value>
    </preference>
    <preference>
      <name>departureAirport</name>
      <value>Madrid</value>
      <read-only>true</read-only>
    </preference>
  </portlet-preferences>
</portlet>
```

Figure 4: A sample *"portlet.xml"* deployment file.

3. WHAT CAN VARY

Being full-fledged applications, portlet variations can manifest in any of the three layers: the presentation layer, the functional layer and the data layer. For the presentation layer, variations can imply rebranding the rendering with customer-specific logos and banners, changing the labels and text that appear in the user interface so that they are appropriate and familiar to the employees and customers of the portal, changing the entry fields that are prompted to the user and even, given the consumer the ability to inlay new markup inside portlet's fragments [7]. As for the functional layer, the multi-step nature of portlets indicates the existence of a process that can be tuned to fit the consumer demands which include the existence of optional steps that can be provided in a consumer basis. Finally, distinct functionalities will probably require distinct data.

This large number of variations advices to focus on some specific re-use contexts. An artifact is not universally variable, and making it variable on A can prevent the artifact from being variable on B. Since, it is most important to identify the distinct situations in which the portlet is most likely to be re-used. All these variations are captured through features. A feature is a product characteristic that customers feel is important in describing and distinguishing members within a family. These features, their structure and cardinalities are depicted as a feature model using the notation introduced by FODA [15].

As an example, consider an air carrier that sells tickets through distinct travel agencies. To this end, the *flightBooking* portlet is developed where the air carrier is the portlet provider, and the portals of the travel agencies are the portlet consumers. A feature of the *flightBooking* portlet is any characteristic, placed by the carrier and used by the travel agency to describe how the flight booking process should be tailored to the agency's idiosyncrasies. For our running sample, the following features are considered (see figure 5):

- *Payment,* which indicates how travel agencies are compensated by their cooperation. Alternatives include (1) *clickThrough fees*, where the carrier will pay the agency based on the number of users who access the portlet; *bounties*, where the carrier will pay the agency based on the number of users who actually sign up for the carriers services through the agency portal; and *transaction fees*, where the incomes of the ticket sales are split between the carrier and the agency. These variants are alternatives.

- *Checkin*, which provides the namesake functionality. It is a boolean.

- *FlightTypes*, which offers two variants: *domestic* and *international*. The travel agency should select at least one.

- *PortletPref.* Portlet preferences can be set by the end user or the portal administrator. *PortletPref* permits to tune which parameters are going to be set as portlet preferences (i.e. liable to be provided by these actors). One of the variants of this feature includes *usrSetDepart*. By selecting this variant, the agency (i.e. the portal owner) lets end users set their favorite departure airport through the *edit* mode. Other option is *Arrival* which allows for two compatible variants *usrSetArrival* and *admSetArrival.* This permits to provide a default for the arrival airport to either end users or administrators, respectively.

Moreover, features are not always independent, but dependencies can exist among them (e.g., requires or excludes). For our sample case, the *usrSetMeal* feature depends on the selection of the *international* variant, i.e. it only makes sense to care about the meal if the portlet supports international flights since domestic flights do not offer this option. For a detail account about feature models see [2].

This feature model conforms **the Consumer Model**. This model acts as a catalog of the variability space offered by the portlet to accommodate the idiosyncrasies of the consumer organization. A **Consumer Profile** instantiates the Consumer Model for a particular organization.

4. WHEN CAN IT VARY

Once features have been identified, we need to indicate for each feature when it needs to be committed to a particular variant of the feature (a.k.a the binding time) [23]. The following options are considered for the portlet case:

- *development time*, where the decision is taken when the portlet is being compile, adding the components required to supply the selected variant,

- *configuration time*, where the decision is set by the portal administrator any time during the lifetime of a running portlet,

- *runtime*, where the decision is resolved during the enactment of the portlet either automatically (e.g. based on the user profile) or by prompting the end user (e.g. through the *edit* mode). The terms "adaptive" and "adaptable" are used throughout the paper to refer to this two kinds of runtime binding.

One extreme approach could be to defer all decisions till runtime, making the system totally adaptive, provided this is technically possible. However, as pointed out in [23] *"when determining when to bind a variant feature to a particular variant, what needs to be considered is when binding is absolutely required. As a rule of thumb, one can in most cases say that the later the binding is done, the more costly (e.g. in terms of performance or resource consumption) it is. Deferring binding from product architecture derivation*

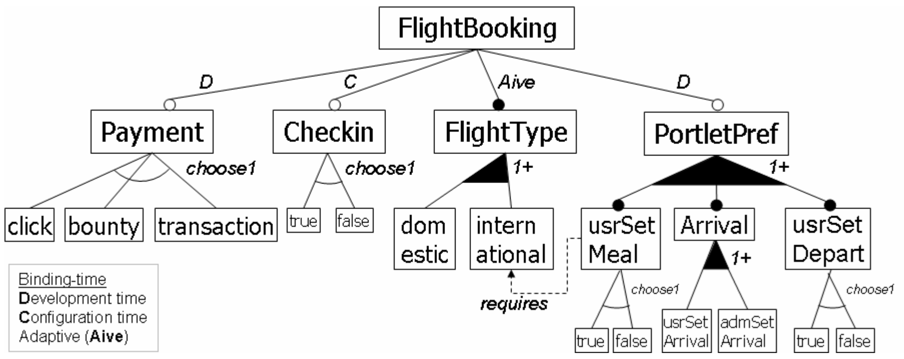


Figure 5: The Consumer Model.

to compilation means that developers need to manage all variants during implementation, and deferring binding from compilation to runtime means that the system will have to include binding functionality. This introduces a cost in terms of, for example, performance to conduct the binding".

This decision can also be influenced by business strategies, delivery models and development processes. For instance, if your business strategy advices *payment* variants to be open for discussion rather than being a fix range of alternatives then, this feature could not be bound at compilation time but deferred till registration time. It is also worth noting that the binding option not only has implementation implications, but it also influences who takes the decision of which variant is finally selected. And this has to do with the business model.

Back to our sample case, figure 5 is extended with annotations to reflect the binding strategy. In this way,

- *Payment* is set at development time (**D**),
- *Checkin* is resolved at configuration time (**C**),
- *FlightTypes* are decided based on the user profile at runtime execution (**Aive** stands for adaptive) (e.g. only users with the CEO profile can book for international flights), and
- *PortletPref* is decided at development time (**D**).

5. HOW IS IT SUPPORTED

Features serve to scope the organization context. They relate to requirements, but do not preclude how the portlet is finally designed or implemented. A first approach is to use some kind of parameterization technique. Even if this were possible, the resulting code could be very cumbersome to develop and maintain. As an example, consider our sample case. Making a single, adaptive portlet that could handle all variants at runtime would make the implementation too complex as the number of possible variant combinations goes quickly above fifty.

This advices to have distinct "versions" of the portlet at least for those features whose decisions can be resolved at development time (e.g. *Payment* and *PortletPref* in our sample case). Nevertheless, the number of combinations still goes up to six different versions, and this for just two features!

If it is necessary to maintain a portlet version for each combination of all these potential variants, portlets will grow in size and number. The cumulative effect of this uncontrolled growth may make to reuse portlets prohibitive [13]. More to the point considering that Web applications are reckoned to be in continuous evolution, and shorter life cycles are commonly achieved at the cost of maintainability [11]. Therefore, the Web setting can not always afford the high maintenance cost that goes with the versioning approach.

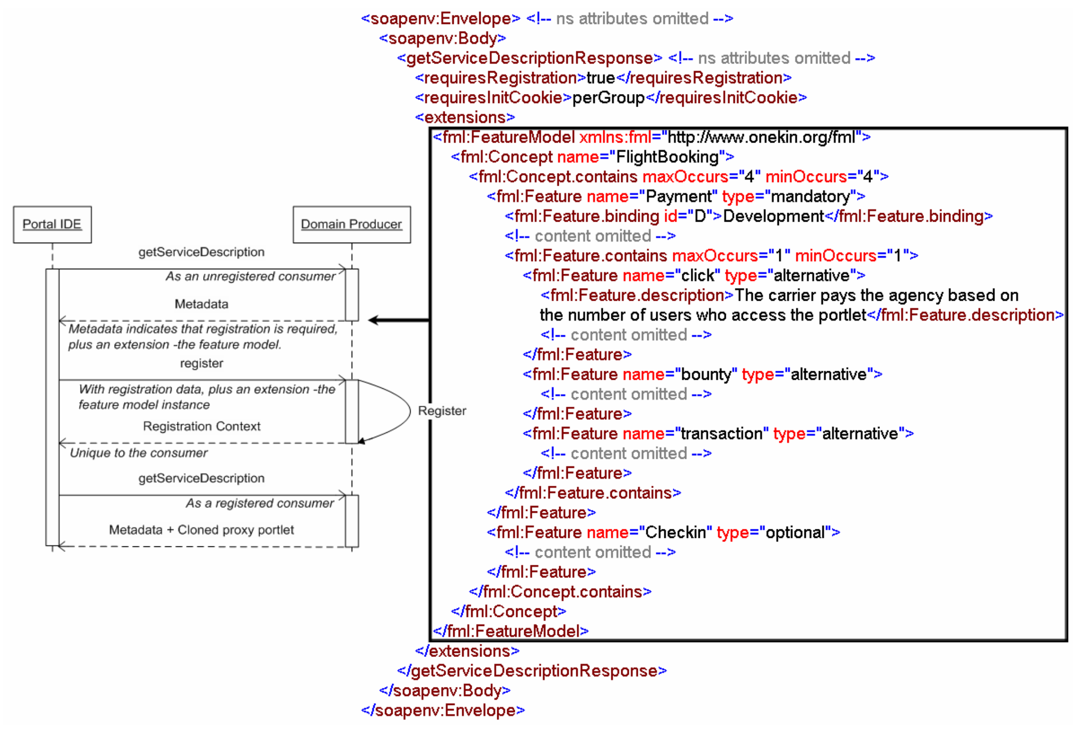
This maintenance penalty partly stems from the fact that features tend to impact more than one artifact, i.e. they cross cut distinct groups of artifacts, which makes variations more difficult to track and maintain. Since a product is defined by selecting a group of features, this implies that a carefully coordinated and complicated mixture of parts of different components are involved [17].

Feature	Artifact	controller	mVIEW model	mVIEW view	mEDIT model	mEDIT view	mCONFIG model	mCONFIG view	portlet.xml
Base		X	X	X	X	X	X	X	X
Checkin		X	X	X					
FlightType	domestic		X	X	X	X			
	internacional		X	X	X	X			
PortletPref	usrSetMeal	X	X	X	X	X			X
	usrSetArrival	X	X	X	X	X			X
	admSetArrival	X					X	X	X
	admSetDepart	X					X	X	X
Payment	click		X				X	X	
	bounty		X				X	X	
	transaction		X				X	X	

Figure 6: Feature scattering along distinct artifacts.

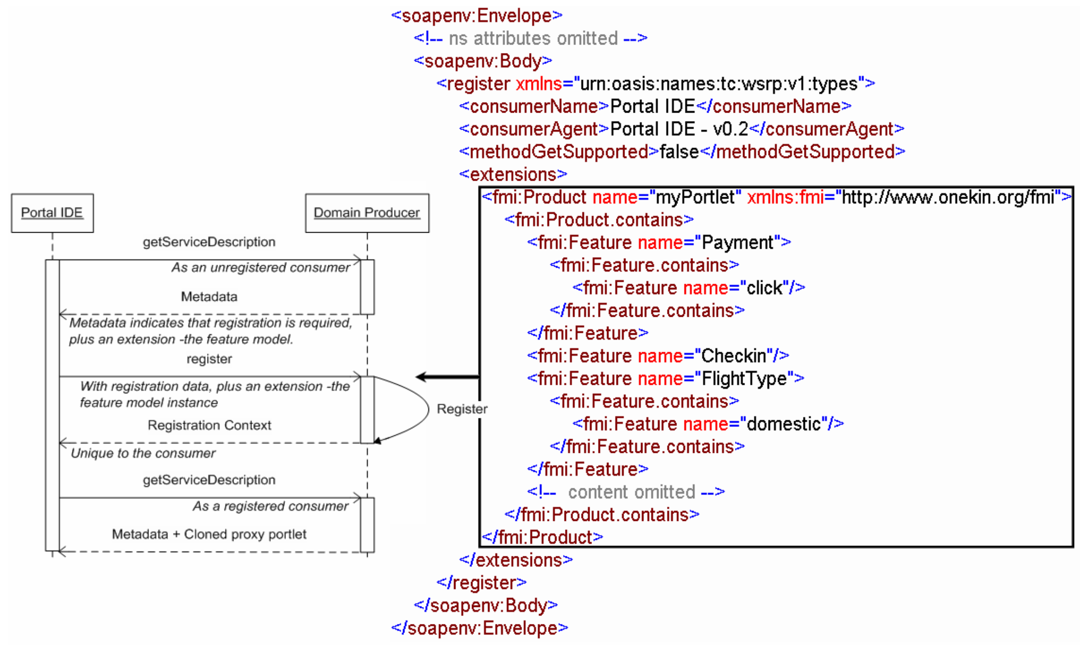
Figure 6 shows the *"feature x artifact"* matrix that highlights the distinct artifacts that are affected by the inclusion of a given feature. For our sample case, as for the artifact axis, portlet realization follows a MVC pattern with a single controller that governs the distinct portlet modes (e.g. *view, edit, config*) where each mode includes a model, a view and the deployment descriptor file where portlet preferences are set (i.e. the *portlet.xml*). On the other hand, the *feature* axis enumerates the distinct characteristics that realize the Consumer Model. The "*base*" stands for the common behavior. Adding feature *Checkin* to this base implies to add/modify some JSP pages for interacting with the user, enlarging the Java classes to access the database, and including this additional step in the application flow. This is reflected in figure 6 by marking the cells for the *controller*, the *mView model* and the *mView view*.

Other example is enhancing this portlet with *usrSetMeal*. This feature allows for the user to be prompted about meal preferences, and requires a new entry form as well as storing this information in the database. Moreover, "meal" is made a portlet preference. This implies changes in *"portlet.xml"* as well as enhancing the views that support the *"edit"* mode which now should permit the user


Portal IDE
Domain Producer
getServiceDescription
As an unregistered consumer
Metadata
Metadata indicates that registration is required, plus an extension -the feature model.
register
With registration data, plus an extension -the feature model instance
Registration Context
Unique to the consumer
getServiceDescription
As a registered consumer
Metadata + Cloned proxy portlet
Register

```
<soapenv:Envelope> <!-- ns attributes omitted -->
  <soapenv:Body>
    <getServiceDescriptionResponse> <!-- ns attributes omitted -->
      <requiresRegistration>true</requiresRegistration>
      <requiresInitCookie>perGroup</requiresInitCookie>
      <extensions>
        <fml:FeatureModel xmlns:fml="http://www.onekin.org/fml">
          <fml:Concept name="FlightBooking">
            <fml:Concept.contains maxOccurs="4" minOccurs="4">
              <fml:Feature name="Payment" type="mandatory">
                <fml:Feature.binding id="D">Development</fml:Feature.binding>
                <!-- content omitted -->
                <fml:Feature.contains maxOccurs="1" minOccurs="1">
                  <fml:Feature name="click" type="alternative">
                    <fml:Feature.description>The carrier pays the agency based on
                    the number of users who access the portlet</fml:Feature.description>
                    <!-- content omitted -->
                  </fml:Feature>
                  <fml:Feature name="bounty" type="alternative">
                    <!-- content omitted -->
                  </fml:Feature>
                  <fml:Feature name="transaction" type="alternative">
                    <!-- content omitted -->
                  </fml:Feature>
                </fml:Feature.contains>
              </fml:Feature>
              <fml:Feature name="Checkin" type="optional">
                <!-- content omitted -->
              </fml:Feature>
            </fml:Concept.contains>
          </fml:Concept>
        </fml:FeatureModel>
      </extensions>
    </getServiceDescriptionResponse>
  </soapenv:Body>
</soapenv:Envelope>
```

Figure 7: The *DomainProducer* communicates to the *PortalIDE* the Consumer Model.


Portal IDE
Domain Producer
getServiceDescription
As an unregistered consumer
Metadata
Metadata indicates that registration is required, plus an extension -the feature model.
register
With registration data, plus an extension -the feature model instance
Registration Context
Unique to the consumer
getServiceDescription
As a registered consumer
Metadata + Cloned proxy portlet
Register

```
<soapenv:Envelope>
  <!-- ns attributes omitted -->
  <soapenv:Body>
    <register xmlns="urn:oasis:names:tc:wsrp:v1:types">
      <consumerName>Portal IDE</consumerName>
      <consumerAgent>Portal IDE - v0.2</consumerAgent>
      <methodGetSupported>false</methodGetSupported>
      <extensions>
        <fmi:Product name="myPortlet" xmlns:fmi="http://www.onekin.org/fmi">
          <fmi:Product.contains>
            <fmi:Feature name="Payment">
              <fmi:Feature.contains>
                <fmi:Feature name="click"/>
              </fmi:Feature.contains>
            </fmi:Feature>
            <fmi:Feature name="Checkin"/>
            <fmi:Feature name="FlightType">
              <fmi:Feature.contains>
                <fmi:Feature name="domestic"/>
              </fmi:Feature.contains>
            </fmi:Feature>
            <!--  content omitted -->
          </fmi:Product.contains>
        </fmi:Product>
      </extensions>
    </register>
  </soapenv:Body>
</soapenv:Envelope>
```

Figure 8: The *PortaIDE* communicates to the *DomainProducer* its Consumer Profile.

to provide a default for this parameter. More to the point, this *usrSetMeal* feature requires the portlet being tuned for international flights (domestic flights do not have meals), hence the effect of a feature can ripple even to artifacts realizing other features!

Therefore, handling variability implies engineering core artifacts for reuse in a planned way. Approaches to reuse can be opportunistic or systematic. The former does not represent an organization-wide strategy but rather, an opportunity exploited on a project-by-project basis. Common "clone&own" practices are a case in point. In this way, the *flightBookingWithCheckin* portlet would be constructed by copying the *flightBooking* basic portlet, and extending it with the *Checkin* additions.

By contrast, systematic reuse takes an organizational perspective rather than a project view. The assumption is that projects in the same business area tend to build systems that satisfy similar needs, so that these systems can be regarded as instances of a family or a product from a product line. Therefore, there is a shift from developing individual portlets to create a portfolio of closely related portlets with controlled variations. That is, developing a product line of portlets.

A **Software Product Line (SPL)** is *"a set of software-intensive systems, sharing a common, managed set of features that satisfy the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way"* [6]. This "particular market segment" corresponds to a business area also known as a *domain*. For our sample case, the domain would be "flight booking". Both the mission of an organization and the changing needs of its customers determine the objectives of that business-area organization.

This implies a shift of focus from a specific application to a domain. This, in turn, leads to distinguish between two processes, namely, the domain engineering process, and the application engineering process. Using a "*design-for-reuse*" approach, **domain engineering** is in charge of determining the commonality and the variability among product family members (through a feature model as described in the previous section). The commonality constitutes **the software platform** i.e. *"the set of software subsystems and interfaces that form a common structure from which a set of derived products can be efficiently developed and produced"* [19]. This includes the architecture, software components, design models and, in general, any artifact that is liable to be reused. On the other hand, and using a "*design-with-reuse*" approach, **application engineering** is responsible for deriving a specific product from the SPL platform.

Distinguishing between these processes permits to separate construction of the software platform from production of the custom application. Domain engineering is responsible for providing the right amount of variability for the custom application to be produced. Application engineering focuses on reusing the software platform, and binding the variability as required for the different applications [21]. Details about using product-line techniques in a Web setting can be found at [1, 5, 8, 12, 22, 24]. These previous works introduce SPL as a means to reduce the time and costs of production and to increase the software quality by reusing elements which have already been tested and secured. Our work however, looks at SPLs also as a cost-effective way to enhance variability and hence, improving the "serviceness" of portlets. Next section introduces a SPL architecture to portlet families. Implementation issues are not addressed here.

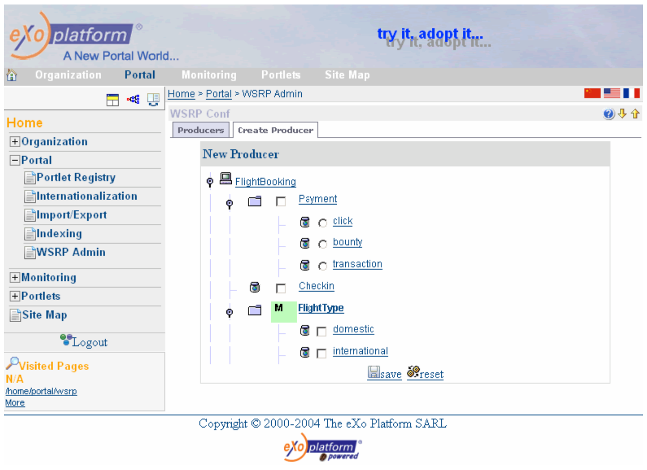


Figure 9: Conforming the *Consumer Profile* through the portal IDE.

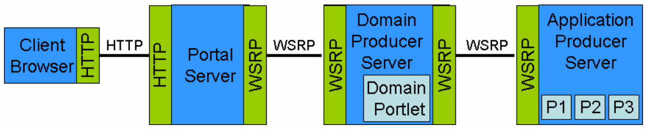


Figure 10: The architecture.

6. A PRODUCT-LINE ARCHITECTURE TO PORTLET FAMILIES

SPLs achieve systematic reuse for a set of applications sharing a "family flavor". What are the specificities brought to SPLs when the product to be built is a portlet? Differences mainly stem from:

1. domain engineering. Besides the user profile, browser agent and other context features, portlets have an additional source of variation: the Consumer Profile. Unlike, standalone software thought to be run on its own, services in general, and portlets in particular, are born to be "consumed" to conform higher functional units. Customization to the consumer then becomes a main ability to achieve seamless, tight higher functional units.

2. application engineering. Current practices assume portlets to be already deployed at the provider. An approach is to create a portlet clone where some configuration parameters can be singularized for the consumer. But variations are always considered at runtime. As argued in previous sections, this can lead to convoluted portlet implementations due to the cross-cutting nature of features. This issue is addressed through "hot deployment" i.e. generating the portlet on demand using generative techniques.

The rest of the section presents how to accommodate these demands in WSRP. The proposal has been validated with WSRP4Java [10].

6.1 WSRP Parameter Extensions

Before a consumer obtains the service (portlet instance), a relationship needs to be established with the producer, determine its capabilities, and set the preferences. This is achieved through the WSRP *getServiceDescription()* and *register()* operations (see section 2). These operations need now to account for the Consumer

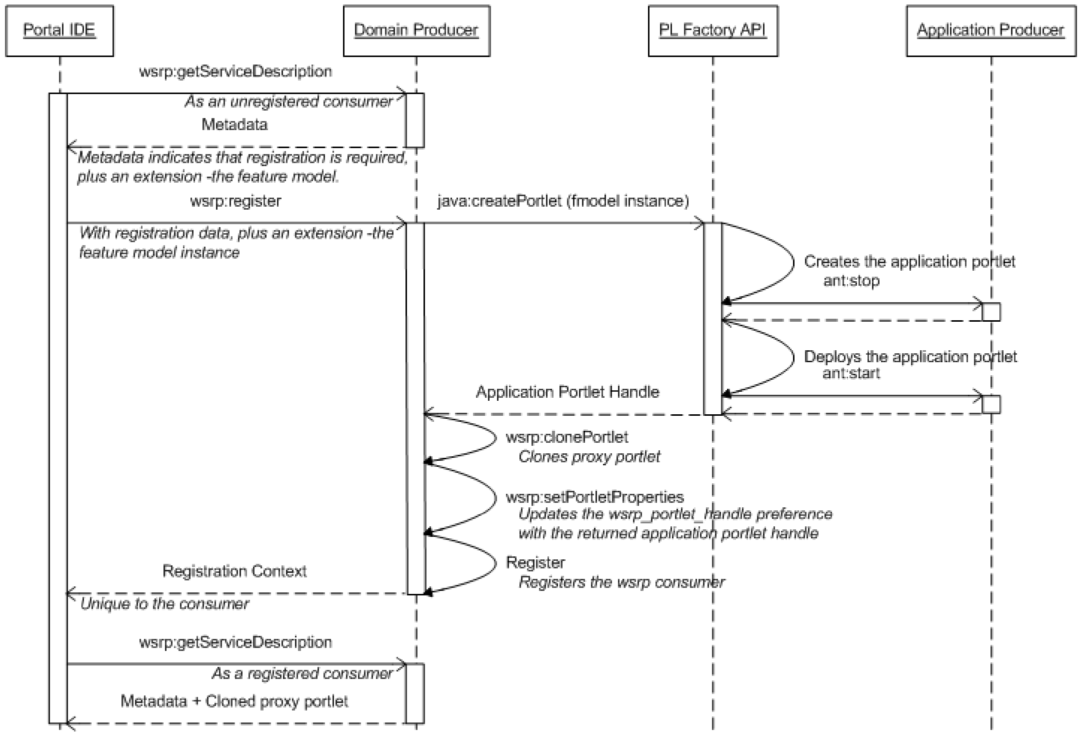

Portal IDE
Domain Producer
PL Factory API
Application Producer
wsrp:getServiceDescription
As an unregistered consumer
Metadata
Metadata indicates that registration is required, plus an extension -the feature model.
wsrp:register
With registration data, plus an extension -the feature model instance
java:createPortlet (fmodel instance)
Creates the application portlet
ant:stop
Deploys the application portlet
ant:start
Application Portlet Handle
wsrp:clonePortlet
Clones proxy portlet
wsrp:setPortletProperties
Updates the wsrp_portlet_handle preference with the returned application portlet handle
Register
Registers the wsrp consumer
Registration Context
Unique to the consumer
wsrp:getServiceDescription
As a registered consumer
Metadata + Cloned proxy portlet

Figure 11: Registration time: sequence diagram.

Model. Specifically, the service description is extended with the Consumer Model, whereas service registration serves to communicate the Consumer Profile of the current consumer.

Once registered, *getServiceDescription* returns the producer's metadata and the list of the "Producer-offered-Portlets". Figure 7 shows a snippet of the returned *ServiceDescription* structure. Using the extensional facilities of WSRP, a new parameter is introduced to describe the Consumer Model using the XML notation proposed in [3] for the description of feature models in XML. Basically, the snippet serializes in XML the model of figure 5. The *portalIDE* takes this model as input and produce a GUI for the portal administrator to input the Consumer Profile that better fits his preferences (see figure 9).

Next, the portal administrator selects the feature variants that better fit its organization, and conforms the Consumer Profile. This profile is returned back to the *domainProducer* through the *register()* operation. This requires to extend the parameters of *register()* to convey the new profile. Figure 8 illustrates this situation for our sample case where the profile includes *click* as *Payment*, *domestic* as *FlightType* and availability of *Checkin*.

6.2 Portlet Registration Extensions

To avoid the cluttering code that crosscutting features can cause, this work argues for the use of SPL techniques. Broadly speaking, the registration of a singularized portlet goes along a three-step process: (1) instantiation of the Consumer Model which outputs a Consumer Profile; (2) synthesis of the singularized portlet as an output of the SPL along the lines of the Consumer Profile, and (3) registration of the singularized portlet with the Consumer.

Current practices assume portlets to be already deployed at the provider. This implies the previous process to be split as follows. First, steps (1) and (2) where the singularized portlet is obtained, and deployed at the provider. And second, step (3) that goes along the traditional registration process. However, this split makes the consumer organization (i.e. the travel agencies for our sample case) aware of the use of SPLs.

By contrast, we strive to make the portlet-generation process transparent. Regardless of whether an SPL approach or a single-product approach is used, portlet consumers go always along the same protocol. To this end, we are forced to use a generative approach to portlet product lines [18]. The architecture of this approach is presented in the following paragraphs.

According to the SPL paradigm, we distinguish between the platform (i.e. the core assets) and the application (see figure 10). The platform is realized as a portlet producer (the *domainProducer*) that holds the scope of the family (i.e. the feature model), and the common platform from where the application portlet is generated. As for the application, it includes a “traditional” producer (the *applicationProducer*) that holds organization-aware portlets (the *applicationPortlet*). The *applicationProducer* is just a container for the portlets generated by the *domainProducer*.

The challenge is how to make this architecture transparent to the consumer. Along with the WSRP protocol, we distinguish between portlet registration and portlet enactment (see section 2).

Portlet registration. A “family portlet” registration is achieved through the *domainProducer* (see figure 10). The only difference with “traditional” registration is that now the response of *getServiceDescription()* is extended to include an XML specification of the feature model of the domain at hand (e.g. booking of flights) as described in the previous subsection.

On reception, the *portalIDE* renders the feature model to the portal administrator who selects the feature variants that better fit its Consumer Profile, and the Consumer Profile is returned back to the *domainProducer* through the *register()* operation.

Next, the *domainProducer* commands the *PLFactory* to generate an *applicationPortlet* along the lines of the Consumer Profile (see figure 11). This *applicationPortlet* is generated and deployed on the *applicationProducer* container. As a result, an *applicationPortlet* handle is returned. On reception, the *domainProducer* clones the *domainPortlet* (see figure 10), which is a proxy portlet, and updates one of its preferences with the returned *applicationPortlet* handle. The outcome of creating this proxy portlet is in turn a *proxyPortletHandle* that is delivered to the *portalIDE* the next time *getServiceDescription()* is invoked.

Portlet enactment. At this time, each organization (e.g. each travel agency) has registered its own portlet which has been customized to fit its Consumer Profile. The travel agency portal (i.e. the portlet consumer) interacts with the *applicationPortlet* through the *domainPortlet*. Such *domainPortlet* is a proxy portlet that just forwards all the requests to the customized *applicationPortlet*. From then on, *applicationPortlets* do not differentiated from "traditional" portlets.

The indirection that this solution implies can rise some concerns about efficiency at enactment time. Notice however, that both the *domainProducer* and the *applicationProducer* are kept on the same machine. Hence, this additional request is local and can be neglected in comparison with the remote call made by the portlet consumer.

7. CONCLUSIONS

This work promotes a SOA approach to portal construction that relies upon portlets as truly reusable services. However, reusability can be jeopardized by the coarse-grained nature of portlets. To overcome this drawback, the notion of Consumer Profile is introduced as a way to capture the distinct organization scenarios where a portlet can be deployed. This in turn leads to the use of an SPL approach to portlet development, and the introduction of an architecture that permits to handle SPL portlets in the same way that traditional portlets. The solution has been supported in WSRP4Java, and the additions on the protocol are WSRP compliant.

8. ACKNOWLEDGMENTS

We thank Iñaki Paz for his helpful comments on earlier drafts of this paper. This work was co-supported by the Spanish Ministry of Science & Education, and the European Social Fund under contract TIC2005-05610. Trujillo has a doctoral grant from the Spanish Ministry of Science & Education. Perez enjoys a doctoral grant from the Basque Government under the "Researchers Training Program".

9. REFERENCES

[1] L. Balzerani, D. di Ruscio, A. Pierantonio, and G. de Angelis. A Product Line Architecture for Web Applications. In *ACM Symposium on Applied Computing (SAC)*, 2005.

[2] D. Batory. Feature Models, Grammars, and Propositional Formulas. In *Sofware Product Line Conference (SPLC)*, 2005.

[3] D. Benavides, S. Trujillo, and P. Trinidad. On the Modularization of Feature Models. In *European Workshop on Model Transformation*, 2005.

[4] J. Blattman, N. Krishnan, D. Polla, and M. Sum. Open-Source Portal Initiative at Sun, Part 2: Portlet Repository, 2006.

[5] R. Capilla and J. C. Dueñas. Light-weight product-lines for evolution and maintenance of Web sites. In *European Conference on Software Maintenance and Reengineering (CSMR)*, 2003.

[6] P. Clements and L. Northrop. *Software Product Lines: Practices and Patterns*. Addison Wesley, 2001.

[7] O. Díaz and J.J. Rodríguez. Portlet Syndication: Raising variability concerns. *ACM Transactions On Internet Technology (TOIT)*, 5(4):627–659, 2005.

[8] O. Díaz, S. Trujillo, and F. I. Anfurrutia. Supporting Production Strategies as Refinements of the Production Process. In *Software Product Lines Conference (SPLC)*, 2005.

[9] eXo Platform. eXo Portal. http://www.exoplatform.com.

[10] Apache Software Foundation. WSRP4Java. http://portals.apache.org/wsrp4j/.

[11] P. Grünbacher, A. Egyed, and N. Medvidovic. Reconciling software requirements and architectures with intermediate models. *Software and System Modeling (SoSyM)*, 3(3):235–253, 2004.

[12] S. Jarzabek, P. Bassett, H. Zhang, and W. Zhang. XVCL: XML-based variant configuration language. In *International Conference on Software Engineering (ICSE)*, 2003.

[13] S. Jarzabek and R. Seviora. Engineering components for ease of customisation and evolution. *IEE Proceedings-Software*, 147(6):237–248, 2000.

[14] Java Community Process (JCP). JSR 168: Portlet Specification Version 1.0, 2003. http://www.jcp.org/en/jsr/detail?id=168.

[15] K. Kang, S. Cohen, J. Hess, W. Novak, and A. Peterson. Feature-Oriented Domain Analysis (FODA) Feasability Study. Technical Report CMU/SEI-90-TR-021, Software Engineering Institute, November 1990.

[16] G. Kappel, B. Pröll, W. Retschitzegger, and W. Schwinger. Customisation for Ubiquitous Web Applications: A Comparison of Approaches. *International Journal of Web Engineering and Technology*, 1(1):79–111, 2003.

[17] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. Loingtier, and J. Irwin. Aspect-oriented programming. In *European Conference on Object-Oriented Programming (ECOOP)*, 1997.

[18] C. W. Krueger. New Methods in Software Product Line Development. In *Software Product Line Conference (SPLC)*, 2006.

[19] M. H. Meyer and A. P. Lehnerd. *The Power of Product Platforms*. The Free Press, 1997.

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

[21] K. Pohl, G. Böckle, and F. van der Linden. *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer, 2006.

[22] D. C. Rajapakse and S. Jarzabek. An Investigation of Cloning in Web Applications. In *International Conference on Web Engineering (ICWE)*, 2005.

[23] M. Svahnberg, J. van Gurp, and J. Bosch. A taxonomy of variability realization techniques. *Software-Practice & Experience*, 35(8):705–754, 2005.

[24] S. Trujillo, D. Batory, and O. Díaz. Feature Oriented Model Driven Development: A Case Study for Portlets. In *International Conference on Software Engineering (ICSE)*, 2007.