NEW APPROACHES TO PORTLETIZATION OF WEB APPLICATIONS
Fernando Bellas (*), Iñaki Paz, Alberto Pan, and Óscar Díaz

Fernando Bellas (*)
Department of Information and Communications Technologies, University of A Coruña.
Facultad de Informática, Campus de Elviña, 15071, A Coruña, Spain
Phone: +34 981 16 70 00
Fax: +34 981 16 71 60
fbellas@udc.es

Iñaki Paz
ONEKIN Research Group, University of the Basque Country
Facultad de Informática, Paseo Manuel de Lardizabal 1, 20018, San Sebastián, Spain
Phone: +34 943 01 80 64
Fax: +34 943 01 55 90
inaki.paz@ehu.es

Alberto Pan
Department of Information and Communications Technologies, University of A Coruña.
Facultad de Informática, Campus de Elviña, 15071, A Coruña, Spain
Phone: +34 981 16 70 00
Fax: +34 981 16 71 60
apan@udc.es

Óscar Díaz
ONEKIN Research Group, University of the Basque Country
Facultad de Informática, Paseo Manuel de Lardizabal 1, 20018, San Sebastián, Spain
Phone: +34 943 01 80 64
Fax: +34 943 01 55 90
oscar.diaz@ehu.es
NEW APPROACHES TO PORTLETIZATION OF WEB APPLICATIONS

ABSTRACT

Portlets are interactive Web mini-applications that can be plugged into a portal. This chapter focuses on “portletizing” existing Web applications, that is, wrapping them as portlets, without requiring any modification. After providing some background on portlet technology, we discuss two kinds of approaches to portletization: automatic and annotation-based. Automatic approaches make use of heuristics to automatically choose the fragments of the Web application pages to be displayed into the space available in the portlet’s window. In turn, in annotation-based approaches, it is the portal administrator who annotates each page of the portletized Web application to specify which fragments should be displayed. Annotation-based approaches also allow to supplement the functionality of the original Web application. Each approach is explained by using a sample scenario based on the same Web application. We also pinpoint the advantages and shortcomings of each approach and outline future trends in portletization.

INTRODUCTION

Moving an application to the Web implies facing its multi-channel character. The very same functionality can be offered through a variety of “channels”, e.g. as a Web application to be accessed from a desktop or a small-screen device (e.g. PDA, smartphone, etc), or as a portlet available through a portal. Although the functionality remains the same, the characteristics of each channel determine the implementation. The pressure to have a Web presence, and the pace at which technology evolves, make most organizations support the desktop-oriented Web application first, and at latter stages, care about adapting it for small-screen devices and portals.

Portlets are interactive Web mini-applications, local or remote to the portal, that render markup fragments that the portal can integrate into a page (usually shared by other portlets). Integrating a Web application as a portlet improves the user experience, since the portal can automatically apply user profiles when accessing the Web application. For example, if the application requires authentication, the portal can store authentication parameters to automatically authenticate the user in the application when she or he logs in the portal. Support for smart bookmarks is another example. The portal can allow the user to store bookmarks to specific screens of the application. Each bookmark stores the navigation sequence to reach such a screen (Anupam et al, 2000).

In principle, to integrate a Web application, a specific portlet must be developed. This implies to develop a new presentation layer, which is always a time-consuming task. This chapter focuses on “portletizing” existing Web applications, that is, wrapping them as portlets. The wrapping approach allows to expose Web applications as portlets to the portal without requiring any modification to the applications, thus reducing very much the development time.

The rest of the chapter is organized as follows. First, we provide background on portlet technology and portletization approaches. Then, we exemplify the two main approaches...
proposed to date by summarizing the portletization techniques the authors have been working on. Next, we compare both approaches. And finally, we outline future trends in portletization and present conclusions.

BACKGROUND

Overview of portlet technology

The classic approach to integrate a Web application into a remote portal consists in defining a Web service (SOAP/REST) that exposes part of the application’s business logic. The problem with this approach is that any portal wishing to integrate such an application must re-implement the user interface. It would be easier if the service was a remote portlet, returning HTML markup fragments rather than plain data. The Web Services for Remote Portlets (WSRP) specification (OASIS Consortium, 2003) standardizes the interfaces of the Web services a portlet producer (typically a portal) must implement to allow another application (typically another portal) to consume its portlets, regardless of the technology the producer and consumer use (e.g. J2EE, .NET, etc). Portlet URLs embedded into fragments point to the portal. Whenever the user clicks on a link, the portal receives the HTTP request, invokes an operation on the portlet’s producer that returns the markup fragment corresponding to this interaction, and finally composes a page that includes the response of this portlet and those of the remaining portlets in the page.

The Java Portlet Specification (Java Community Process, 2003) standardizes a Java API (JSR 168) for implementing local, WSRP-compatible portlets. Java portlets run in a portlet container, a portal component that provides portlets with a runtime environment. Apart from the portlet container, the typical architecture of a Java portal server (Bellas, 2004) includes other components:

• The WSRP producer component provides an implementation of the WSRP interfaces, so other consumers can access local portlets.
• The WSRP consumer is implemented as a local portlet that acts as a generic proxy for any WSRP producer.

The WSRP standard defines portlet modes and window states. Portlet modes refer to the types of functionality a portlet can perform. In view mode, the portlet renders fragments that support its functional purpose (e.g. displaying weather forecasts). In edit mode, the portlet lets the user customize the behavior of the portlet (e.g. specifying the default temperature unit). The help mode provides help information. Finally, the preview mode allows to view the portlet before adding it to a portal page.

Window states act as an indication of the space available in the portlet’s window. The normal state indicates the portlet is likely sharing the page with other portlets. When the state is minimized, the portlet should not render visible markup. In the maximized state, the portlet has more space compared to other portlets in the page. Finally, the solo state indicates the portlet is the only portlet in the page.

The portal decorates the fragments returned by a portlet with buttons to let the user select the portlet mode and window state.
Approaches to portletization

During the last years, several approaches have been proposed to address the portletization of Web applications. A first way of classifying them is according to whether they provide “shallow” or “deep” portletization. These two types of portletization are described below:

- **Shallow portletization** (a.k.a. “Web clipping”). These techniques only allow to portletize a single page (typically the home page) of the Web application. If the user navigates to a new page (for instance, by clicking on a link in the portletized page), then the rest of the interaction between the user and the application will be outside the scope of the portal system.

- **Deep portletization**. This implies portletizing the whole bulk of pages that comprise the Web application. In this case, the whole interaction between the user and the Web application occurs inside the portal realm. (Díaz et al, 2005b) describes in detail the issues involved in deep portletization.

The approaches to portletization proposed to date are based on providing a special portlet (hereafter, the “bridge portlet”) that navigates automatically to the target Web page (Anupam et al, 2000; Freire et al, 2001; Pan et al, 2002), and extracts the desired regions. The techniques proposed to build the bridge portlet can be roughly classified into two different types: annotation-based approaches and automatic approaches.

In the annotation-based approaches, the portal server administrator configures the bridge portlet for a specific Web application by manually identifying in the target pages the set of regions she or he is interested in. Usually, the process of specifying the desired regions can be entirely performed through a graphical interface that automatically generates a file, sometimes called an annotation file (Hori et al, 2004). This file specifies the location of those regions (e.g. by using XPath expressions).

Current industrial portal servers (e.g. BEA WebLogic Portal, IBM WebSphere Portal, Oracle Portal, etc) provide this kind of annotation-based support, but they only allow for shallow portletization. Furthermore, changes in the Web pages (e.g. adding or removing an HTML tag) may easily invalidate the addressing expressions in the generated annotations, which leads to a maintenance nightmare.

In turn, works like (Díaz et al, 2005b; Paz et al, 2007) allow for annotation-based, deep portletization. They also consider the problem of how to generate annotations with resilient addressing expressions, which may keep on working correctly in the presence of certain layout changes in the target pages.

Automatic approaches require minimum human intervention to configure and maintain the bridge portlet for each target Web application. In fact, the portletization of Web applications can be carried out by the end-user. Web pages are displayed according to the space available in the portlet’s window (window state), and the system is responsible for automatically (heuristically) obtaining the most “relevant” regions of the target page that fit into the available space. Automatic approaches allow to naturally realize deep portletization.
For instance, in the automatic approach proposed in (Bellas et al, 2006), when the portlet is in the solo state, all the content of the page is displayed. In the maximized state, only the body block (all regions except header, footer, and sidebars) is shown. In the normal state, the body block is segmented. The bridge portlet is in charge of automatically detecting the main blocks of the page, extracting the body block, and segmenting it.

Annotation-based approaches account for a fine-grained, accurate adaptation of the Web application to the portlet setting, since particular regions of the Web pages can be extracted. However, they require to manually annotate each page in order to specify the regions of interest and how they will be displayed depending on the window state. In consequence, they can only be used by a developer or a portal administrator. Annotations must also be maintained to accommodate the layout changes in the target sources.

In turn, automatic approaches require minimum human intervention, enable end-user portletization, and are more resilient to changes in the target pages. However, since they rely on heuristic-based techniques, they may be less accurate. In addition, they do not allow for fine-grained portletization, that is, when the user is only interested in specific parts of the pages.

The next three sections describe and compare in more detail these two approaches using a specific case study to further illustrate how they work.

**AUTOMATIC PORTLETIZATION**

**Sample scenario**

This section uses the Yahoo!™ Weather service (http://weather.yahoo.com) as a sample scenario to illustrate a possible model of how a Web application can be automatically displayed as a portlet in view mode. According to the expected behavior in the standard window states, the bridge portlet automatically adapts Web pages to the space available in the portlet’s window. It also tries to keep the original look-and-feel. Fig. 1 shows the responses generated by the bridge portlet.

Fig. 1(a) shows the first page of the application when the user has selected the solo window state. Since the portlet is the only one in the portal page, the bridge portlet decides to show the original content. As the fragment returned by the portlet must be inserted into the portal page, the portlet extracts the markup contained inside the \(<\text{body}>\) tag of the original page, and rewrites the URLs that must be navigable inside the portlet’s window.

Fig. 1(b) shows the same page when the user selects the maximized window state. In this state, the portlet is supposed to have a lot of space in the page, but less than in the solo window state. The content of Web pages is often structured in five high-level blocks: header, footer, left side-bar, right side-bar, and body (the content region of the page). From the point of view of the actual content, the body block is the most important one, since the rest of blocks are usually dedicated to navigational issues and miscellaneous information. In consequence, to save space, the bridge portlet detects the
high-level blocks, extracts the body block, and rewrites the URLs that must be navigable inside the portlet’s window.

Fig. 1(c) shows the same page when the user selects the *normal* window state. Since the portlet is supposed to share the page with other portlets, the bridge portlet must reduce even more the amount of markup it generates. To do so, it applies the same transformations as in the *maximized* window state and returns a fragment that contains the first part of the body block.

Fig. 1(d) illustrates the situation when the user enters *La Coruna* in the form and clicks on the *Go* button. The portlet had rewritten the form’s URL previously to point to the portlet itself. In consequence, this interaction causes a request directed to it. The portlet fetches the corresponding page and proceeds as in the previous case.

To keep the application navigable, the bridge portlet includes five *navigability buttons* (enabled or disabled as appropriate) at the bottom of all the generated fragments. Such buttons are links that point to the bridge portlet. The *previous/next fragment* buttons allow the user to navigate through the fragments of a page when the window state is *normal*. The *previous/next page* buttons allow the user to navigate through the pages already visited. For convenience, a *home page* button is also included to let the user go directly to the initial page.

**A case study**

This section provides an overview of the bridge portlet described in (Bellas et al, 2006), which allows to automatically display a Web application as a portlet according to the model illustrated previously.
To understand the design of the bridge portlet, it is important to realize that the fragments shown in Fig. 1 can be generated by using a chain of transformations that depends on the current window state. For example, when the window state is maximized, as in Fig. 1(b), the fragment can be generated by (1) detecting the high-level blocks of the original page and extracting the body block, and (2) including the navigability buttons. When the window state is normal, as in Fig. 1(c), an additional transformation, that splits the body and extracts the appropriate fragment, can be applied between the two transformations used in the maximized window state.

The above observation gives rise to the framework depicted in Fig. 2. TransformerChainManager allows to obtain a particular chain (TransfomerChain) of transformation strategies to be applied to a page in a given window state. All transformation strategies implement the Transformer interface. The transform method in TransformerChain allows to execute the chain of transformations on a given page. By default, the framework provides three transformers:

• **BodyExtractor.** It returns the body block by detecting the high-level blocks of the original page and discarding the header, the footer, and the side bars.

• **GeneralSegmenter.** It divides the page into rectangular sections and returns the requested section. The area (in square pixels) of the rectangular sections is specified by a configuration parameter, and represents the approximate size of the portlet’s window the user wishes in the normal window state.

• **PageNavigabilityTransformer.** It inserts the navigability buttons.

The default implementation of the TransformerChainManager returns the empty chain when the window state is solo, the \{BodyExtractor \rightarrow PageNavigabilityTransformer\} chain when the window state is maximized, and finally, the \{BodyExtractor \rightarrow GeneralSegmenter \rightarrow PageNavigabilityTransformer\} chain for the normal window state.

Except in the minimized window state, when the bridge portlet receives a request for a page, it calls: (1) **getPage** on PageManager to retrieve the page, (2) **getTransformerChain** on TransformerChainManager to obtain the particular chain of transformers to be applied in the current window state, (3) **transform** on TransformerChain to generate the fragment by applying the chain of transformations, and finally (4) **postProcessPage** on PagePostProcessor. When the window state is minimized, the bridge portlet returns an empty fragment (this is the default behavior inherited from javax.portlet.GenericPortlet).

The **Page** object returned by PageManager provides “getter” methods to access certain information of the page, such as JavaScript code, CSS styles, and the fragment that contains all the tags inside the \(<body>\) tag. Transformers work with this fragment, since the rest of tags cannot be included in the portal page. Internally, this fragment is represented as a DOM tree. All transformers work with the standard DOM API. Since some transformers (BodyExtractor and GeneralSegmenter) need to know the visual information of each node, we use a mechanism of DOM Level 3 that allows to attach information to nodes. In particular, each node has an attached object providing visual
information of the rectangular region it defines. This information includes the X and Y coordinates of the left upper corner of the rectangle, and its width and height.

![Diagram](image)

**Fig. 2.** Bridge portlet’s framework.

The implementation of *PagePostProcessor* has to rewrite the URLs that must be navigable inside the portlet’s window, include CSS styles and JavaScript code defined inside the `<head>` tag of the original page, and rename all JavaScript functions/variables and CSS styles to avoid collisions with the rest of portlets included in the same page and with the portal itself.

The most critic transformer is *BodyExtractor*, since this is currently the only heuristic transformer. Internally, *BodyExtractor* uses four “block removers”: *HeaderRemover*, *FooterRemover*, *LeftSideBarRemover*, and *RightSideBarRemover*. The implementation of all block removers is orthogonal. Each remover tries to find a group of nodes that make up a region that visually can be considered as the block it is trying to remove. A region can be classified as a particular block in function of its shape (width/height ratio) and its position. For example, the header block has a flat shape (high width/height ratio) and is placed at the top of the page. Each remover uses configuration parameters that define the maximum height (for the header and the footer) or width (for the left and right sidebars) of the region, and the minimum (for the header and the footer) or maximum (for the left and right sidebars) width/height ratio the region must fulfill.

To improve their accuracy, block removers take also into account the logical nesting of nodes. In particular, nodes that by its position and shape could be classified into one of the outer regions (header, footer, and sidebars) are not removed if they are contained in a node that probably defines the region corresponding to the block body.

To portletize a Web application, the portal user (and administrator, or even, an end-user) creates an instance of the bridge portlet, adds it to a portal page, and selects the *edit* mode. In the *edit* mode, the bridge portlet displays a configuration wizard that allows to specify:

- The URL of the first page of the application (e.g. `http://weather.yahoo.com` in the sample scenario). This is the only field that must be explicitly filled.
• A set of URL patterns for the URLs that must be navigable inside the portlet’s window. The bridge portlet rewrites the URLs matching these patterns to point to itself. In the sample scenario, if we wish to keep the search form and the “browse by location” links navigable, we could specify /search/weather2 for the search form, and /regional/* and /forecast/* to browse by location. URL patterns can be specified in “absolute URL” or “absolute path” (as in the example) notation.

• The configuration of BodyExtractor and GeneralSegmenter. The configuration wizard provides reasonable default values, which have been obtained empirically.

The above information is stored as the portlet’s preferences (Java Community Process, 2003), and in consequence, each portlet instance has its own configuration.

Experiments have confirmed the viability of this automatic approach. To this end, 50 Internet Web applications were portletized. 15 of them were used to empirically choose good default values for BodyExtractor’s configuration parameters, and the other 35 were used to evaluate the accuracy of BodyExtractor. More than 70% of the Web applications were displayed perfectly with the default configuration. For the rest of applications, more than half were displayed perfectly by adjusting usually only one configuration parameter. The remaining applications were displayed with minor problems.

ANNOTATION-BASED PORTLETIZATION

Sample scenario

This section presents a sample scenario, based also on the Yahoo! ™ Weather service, to illustrate an example of the annotation-based approach to portletization. The intention is to build a weather portlet based on the Yahoo! ™ Weather service. Its main functional purpose is to render the weather forecast for a user-provided place.

In this sense, Fig. 3 (1) details the decomposition of the Yahoo! ™ Weather service home page depending on the window state.

The Yahoo! ™ Weather home page (see Fig. 3(a)) is the first page of the Web application and thus the origin of the fragments. Notice that it has been deliberately decided not to render weather news, which could be used to produce a new portlet. The first fragment viewed in the portlet (see Fig. 3(b)) in the normal window state shows the form to search for a location. From this fragment, through the use of the bottom links of the fragment (“Locations” and “Maps” links) we can move to a browsed search of a location (see Fig. 3(c)) or to a browsed search of a weather map (see Fig. 3(d)). From any of them, by maximizing the portlet window, a portlet fragment (see Fig. 3(e)) containing all those small fragments is rendered.

Then, searching a location on the form or browsing the locations would end on the presentation of a fragment that contains the forecast for the selected location (See Fig. 4).

How the portlet behaves with the use of window states has been described. However, portlets also have modes. Fig. 3 (2) shows how the bridge portlet reacts to the use of
modes. A form which allows to ask for help on weather terms has been used for the help mode, while the edit mode allows the user to edit her or his preferences. In this case, the edit mode allows to set a default value for the search form or to select a default behavior (one of the three fragments extracted from the home page) for the first time the user accesses the portlet (in normal window state).

Fig. 3. Yahoo!™ Weather service decomposition in: (1) Window State fragments for the home page and (2) Portlet Modes in normal window state.

A case study

This section introduces the annotation-based bridge portlet framework, described in (Paz et al, 2007), where the sample previously presented is used to drive the implementation concepts. Annotation refers to the process of increasing the knowledge on a Web application through the use of “notes” that on a later stage can be re-used for new means. In this sense, a Web application can be annotated to add semantic meaning to its contents (Paz et al, 2006b), or as used in this case, to wrap a Web application as a full-featured portlet. Annotations on a Web application can be performed directly on the source pages (HTML), or maintained on a remote repository to be accessed by whoever wants to use them, granting that the Web application is left untouched. Annotations are normally specified as XML documents to ease its reuse.

Among other portletization concepts introduced in previous sections (e.g. URL rewriting, etc), it is important to notice that to portletize a Web application, first we should know which parts of the Web application are to be reused and how, and the annotations produced to be consumed by our bridge portlet framework reflects this fact. These annotations will be stored locally to our framework.

Information extraction patterns can be used to locate a portion on a Web application page. Patterns are “tools” used to locate data on a page. These “tools” range from regular expressions (Atzeni et al, 1997) to complex expression languages (Baumgartner
et al, 2001). (Laender et al, 2002) presents a good survey on them. However, and unlike previous approaches, our aim is not to extract data buried into HTML pages, but chunks of XHTML markup that can become portlet fragments. To characterize these chunks, we resort to a language able to express patterns based on the page structure: XPath. **XPath** is a standard language to address elements in XML documents. Since HTML documents (Web pages) can be converted to XML through the use of tools, such as JTidy (The JTidy Volunteer Team, 2006), XPath expressions may extract those parts of the pages that make up a portion.\(^1\)

Thus, given a page, it can be annotated through the use of XPath expressions. Since Web application pages will normally be different, and different pages normally imply distinct content with different structure, an XPath selecting a portion on a page may not select nothing at all, or even worse, select unwanted content on another. Therefore, pages need to be classified.\(^2\) In this sense, a Web application is seen as a state machine where the states represent the distinct pages and the arrows the user interactions among them (see Fig. 4).

![Fig. 4. Yahoo!™ Weather application state machine.](image)

The model shown in Fig. 5 summarizes these concepts. Basically, a Web application can be seen as a set of pages, while a portlet can be seen as a set of fragments, where both pages and fragments are composed of portions. The model is captured in XML annotation files that are consumed by the bridge portlet.

First of all, notice that a bridge portlet, as a portlet, has portlet modes. However, in a Web application environment, there are no different modes of operation, everything may be intertwined on a page, different pages, or even do not exist at all. Thus, in contrast to the automatic approach, a portlet may have one or more entry points (one per supported mode). For instance, on the sample portlet, the *help* mode fragment comes directly from the Yahoo!™ Weather help page.

Basically, the process executed by the bridge portlet is as follows:

1. The first page for the current mode of operation is retrieved.
2. The retrieved page is classified among the defined classes.
3. The portions are extracted from the retrieved page. Portions denote those semantic units of content on pages (i.e. banners, forms, sections, etc).

---

\(^1\) XSLT is used to extract the portions from the page by solely copying on the output those elements selected by the XPath expressions.

\(^2\) XPath may also be used to characterise a page based on the existence of a concrete element on the document. Text patterns are used also over URLs to capture navigational characteristics of the corresponding pages.
4. A fragment (in the portlet’s output) is built using the extracted portions depending on the mode of operation and the window state.
5. The fragment is post-processed to handle URL redirection, CSS, etc.
6. Finally the fragment is submitted to the portal to be rendered, where the user interacts with it and the cycle starts again.

![Fig. 5. Bridge portlet model.](image)

Some examples follow that define the extraction rules of a portion and the classification rules of a page from Yahoo!™ Weather:

```xml
<page class="home">
  <classifier>
    <xpath select="//div[@class='ulmForm']"/>
  </classifier>
</page>

<portion name="form">
  <extractionRules pageClassRef="home">
    <xpath mode="select">
      //div[@class='ulmForm']
    </xpath>
  </extractionRules>
</portion>
```

The left column specifies what characterizes the *home* page. Given a page, we know that it is the *home* page because it contains one *div* whose class is *ulmForm*. The right column describes that the *form* portion on the *home* page is gathered by selecting that *div*.

Annotation technologies to wrap a Web application normally hide the definition of the extraction patterns on a rich user interface. This is also our case where a tool enables the user to build the portlet by visually selecting the elements of the page that make up a portion. In addition, our tool also optimizes the gathered extraction patterns to improve the robustness of the portlet to changes on the Web application through the use of simulated annealing (Paz et al, 2007). For instance, the XPath expression `//div[@class='ulmForm']` is an optimized version of the absolute XPath expression gathered from the tool that selects that *div*. Notice that the XPath expression is based only on an attribute (*class*) that somehow describes its contents.

So far, the focus has been on wrapping the Web application as a portlet so that a Web application can be included as part of the portal. However, the fact of being enacted from a portal has deeper implications. Portals provide diverse commodities to enhance both user experience and seamless application integration.
Some of the implications include integrating the application with the Single Sign-On feature of the portal, perfect CSS harmonization, maintaining a form log with last values typed by the user, personalizing navigation on the Web application through usage mining, etc.

Due to the difficulties to foresee the specific enhancements a Web application needs when portletized, the bridge portlet can be adjusted through the use of "extensionModules". Extension modules are plugin-like packages that extend the portlet with a given functionality using an aspect-like approach. An aspect is a unit of modularity that encapsulates crosscutting concerns. As stated in [Elrad et al. 2001], “separating the expressions of multiple concerns … promises simpler system evolution, more comprehensible systems, adaptability, customizability, and easier reuse”.

Extension modules extend the bridge portlet in 2 ways:

- Defining new rendering portions and fragments to be presented.
- Defining aspects that supplement the bridge portlet’s life cycle with the new required functionality.

For instance, enabling the portlet to let the user select which of the fragments is used as a home fragment whenever the portlet is enacted (See Fig. 3(III)) implies:

- A new fragment for the edit mode that reuses the form.
- A new title portion.
- A new portlet portion that selects the default behavior.
- An aspect that captures the form submission and stores the user’s preferences.
- An aspect, which depending on the user’s preferences, modifies the default behavior and selects the user’s preferred fragment to be rendered.

```xml
<fragment name="homeEdit" layout="edit.jsp">
    <portion name="newTitle"/>
    <portion name="form"/>
    <portion name="behavior"/>
    <portion name="other"/>
</fragment>

<factory class="org.onekin.TitleFactory">
    <portions>
        <portion name="newTitle"/>
    </portions>
</factory>
<advice class="org.onekin.PrefStoreAdv">
    <pointcut moment="before" method="ReadParams"/>
</advice>
```

In this case, the left column specifies a new fragment that will be used for the edit mode, and which contains the form to set a default value, a newTitle for the fragment and several radio buttons to select the default behavior. The newly specified portions (newTitle and behavior) have to be defined. The factory element specifies the implementation class that builds the specified portions. The column on the right specifies that the TitleFactory class creates the newTitle portion. Finally, an advice, the code an aspect supplements, needs to be defined. The PrefStoreAdv advice will be executed before the portlet reads the parameters (readParams) from the portlet request in order to capture and set the user’s preferences whenever the form has been enacted in the edit mode.

---

3 This fragment has been assigned to the edit mode for the home page, which is the entry point to the Web application. Thus the form portion has been reused here.
This tiny example introduces how a bridge portlet can be extended. However, more complex features can be built through the use of extension modules. (Paz et al., 2006b) presents how to annotate portlets to be semantically integrated in a portal where data from one portlet feeds other portlets.

**DISCUSSION**

The sample scenarios included in the two previous sections allow to illustrate the advantages and shortcomings of automatic and annotation-based approaches. One advantage of automatic approaches is that they require minimum human intervention to configure and maintain the bridge Portlet, enabling end-user portletization. In the sample scenario illustrated in the automatic portletization section, the user easily configures the bridge portlet by using the *edit* mode to specify the URL of the first page of the application (http://weather.yahoo.com) and a set of URL patterns for the URLs that must be navigable inside the portlet’s window (/search/weather2 for the search form, and /regional/* and /forecast/* to browse by location). For some Web applications, it could be necessary to adjust some configuration parameter in either the BodyExtractor or GeneralSegmenter configuration. However, experiments have shown that a high percentage of applications can be displayed perfectly with the default configuration or adjusting only one configuration parameter.

Another related advantage of automatic approaches is that they are more resilient to changes in the pages of the source Web application, since they do not make use of addressing expressions to extract regions. On the contrary, they use heuristic methods to obtain the most “relevant” page regions depending on the page space available to the portlet.

Finally, another advantage of automatic approaches is that they naturally allow for deep portletization, since it is not necessary to configure or annotate each page of the portletized application.

On the other hand, annotation-based approaches allow for a more precise kind of tuning. For example, in the scenario provided in the annotation-based portletization section, the portal administrator has configured the bridge portlet to display the search form and the two kinds of browsed search in the *maximized* window state (see Fig. 3(e)) when the user accesses the first page of the application, and only one of the three components in the *normal* window state (see Figs. 3(b), 3(c), and 3(d)). This contrasts with the markup generated by the approach described in the automatic portletization section (see Figs. 1(b) and 1(c)), which includes more content in both window states.

However, the level of tuning exhibited by annotation-based approaches requires the use of addressing expressions to extract specific regions, which may break if the markup of the original Web pages change. Maintenance problems may arise if the number of applications and/or pages to be portletized is high. To alleviate this problem, annotation-based approaches may provide tools to: (1) build the portlet using a rich user interface and (2) generate change-resilient XPath expressions.

Unlike automatic approaches, annotation-based approaches allow to provide additional modes other than the *view* mode. For instance, in the sample scenario presented in the
annotation-based portletization section, the portal administrator has configured the bridge portlet to show the content of the Yahoo! ™ Weather help page when the user selects the help mode (see Fig. 3(II)). The portlet has also been configured to provide an edit mode (see Fig. 3(III)) that allows to set a default value for the search form or to select a default behavior the first time the user accesses the portlet. Note that the edit mode provided by the bridge portlet in the automatic approach is not specific to the portletized application nor it supplements its functionality. On the contrary, it provides a mechanism to configure the bridge portlet. Annotation-based approaches also allow the use of extension modules to provide additional functionality not provided by the original application.

In conclusion, we can see that the advantages of one approach are the drawbacks of the other, and vice versa. In fact, the two approaches are complementary. When a precise kind of tuning is required, the annotation-based approach is mandatory. Otherwise, the automatic approach is much more maintainable. In consequence, we believe that industrial solutions should provide support for both approaches.

FUTURE TRENDS

As it has been established in previous sections, both annotation-based and automatic approaches to portletization have their own advantages in specific scenarios. Therefore, a complete state-of-the-art package for portletization should support both. For instance, the bridge portlet described in the automatic portletization section could include a new transformer to allow annotation-based portletization, as an alternative to using the BodyExtractor transformer. Along these lines, industrial portal servers should provide similar integrated solutions in the near future if they want to allow portletization in more complex, realistic situations than the ones they currently support.

Another interesting direction that will probably generate further research is that of creating new techniques for identifying and extracting the “relevant” regions in the automatic approach. This can benefit from work in other areas. For instance, a closely related field is that of automatically adapting content for small-screen mobile devices. One example along these lines is the case study presented in the automatic portletization section. The techniques developed during the last years for automatically or semi-automatically extracting structured data from Web pages (Laender et al, 2002; Arasu et al, 2003; Pan et al, 2002) could also be successfully adapted to this new problem. In addition, learning techniques could be employed to exploit the user past navigation experience on the target Web application to improve the system guesses about what regions are “relevant”.

In the case of annotation-based approaches, automatically detecting when addressing expressions break is another area where research in Web data extraction could be leveraged. Works such as (Raposo et al, 2006; Lerman et al, 2003) addressed the related problem of automatically maintaining Web wrappers.

We predict that portlet composition and interoperation will be another hot topic. This refers to composing several independently developed portlets to achieve a common goal. The upcoming versions of current portal standards (OASIS Consortium 2003; Java Community Process, 2003) will soon provide the developer with support to communicate different portlets. Nevertheless, this does not solve the problem of
communicating portletized Web applications. To make this possible, it is also necessary to correctly interpret the markup generated by the portletized applications. Semantic Web techniques can help overcome these limitations by providing a method to easily specify the structure and semantics of the markup generated by different portlets, as well as for establishing semantic mappings between them. (Díaz et al, 2005a) describes a pioneer work in this direction.

CONCLUSIONS

In this chapter, we have studied the problem of “portletizing” existing Web Applications. Portlets are interactive Web mini-applications which run inside a portal server. Each page of the portal is usually shared by several portlets.

Creating a new portlet from an existing Web Application involves issues such as adjusting its content to fit into the available space for the portlet or rewriting the URLs in the navigation links through the pages of the application to be used from inside the portal. Integrating a Web application as a portlet improves the user experience, since they get one-stop access to multiple services and the portal can automatically apply user profiles when accessing them.

We have studied two kinds of approaches for “portletization”: annotation-based and automatic. We have exemplified both approaches with the detailed description of two systems from our previous research.

We have also outlined their advantages and shortcomings. We conclude that they serve related but different goals and, therefore, they should be considered complementary to each other. Therefore, it is our opinion that forthcoming industrial solutions should provide integrated support for both approaches in order to allow realistic, successful portletization of existing Web Applications.

We have also argued that more research work is needed in the areas of portlet composition and interoperation and in the use of more sophisticated content extraction techniques for automatic portletization.

ACKNOWLEDGEMENTS

This work was partially supported by the Spanish Science and Technology Ministry and European Social Funds (FEDER) under projects TSI2005-07730 and TIC2002-01442. Alberto Pan’s work was partially supported by the “Ramon y Cajal” program of the Spanish Education and Science Ministry.

REFERENCES

KEY TERMS

Annotation approach (for portletization): The approach to portletization where the portal administrator configures the bridge portlet for a specific Web Application by manually identifying in the target pages the set of regions she or he is interested in. Usually, this process generates a file called an annotation file.

Automatic approach (for portletization): The approach to portletization where the configuration of the bridge portlet requires none or minimal human intervention, allowing end-users to carry out portletization tasks themselves. These systems use diverse heuristics to automatically obtain the most “relevant” regions of the target pages that fit into the space available in the portlet’s window.

Bridge Portlet: A special portlet used to wrap an existing Web Application into a portlet. When the user interacts with the portletized application, the bridge portlet navigates automatically to the original page in the existing application to extract the desired regions.

Deep Portletization: The kind of portletization where the whole bulk of pages comprising the existing Web Application are portletized. In “deep portletization”, the whole interaction between the user and the Web application may occur inside the portal.

JSR 168: The Java Portlet Specification (JSR 168) standardizes a Java API for implementing local portlets, which can be packaged and deployed in any JSR 168 compliant Java portal. If the Java portal provides WSRP producer functionality, JSR 168 portlets can be remotely accessed by WSRP consumers.

Portal: A portal is a Web application providing their users with integrated Web access to a set of underlying applications providing different services to the user.

Portlet: Portlets are interactive Web mini-applications, local or remote to the portal, that render markup fragments that the portal can integrate into a page (usually shared by other portlets).

Portletization: The term “portletization” refers to wrapping an existing Web application into a portlet, thus allowing its integration in a portal.

Shallow Portletization: The kind of portletization which only allows to portletize a single page (typically the home page) of the existing Web application.

WSRP: The Web Services for Remote Portlets (WSRP) specification standardizes the interfaces of the Web services a portlet producer (typically a portal) must implement to allow another application (typically another portal) to consume its portlets, regardless of the technology the producer and consumer use (e.g. J2EE, .NET, etc).