Ph.D. Thesis

Variability in Remote Portlets

Sandy Pérez González

2015

eman ta zabal zazu

Universidad del País Vasco

Euskal Herriko Unibertsitatea

Web Engineering Research Group
Supervisor: Prof. Dr. Oscar Díaz

Variability in Remote Portlets

Dissertation

presented to

the Department of Computer Languages and Systems of

the University of the Basque Country

in Partial Fulfillment of

the Requirements

for the Degree of

Doctor of Philosophy

Sandy Pérez González

Supervisor: *Prof. Dr. Oscar Díaz García*

San Sebastián, Spain, 2015

This work was hosted by the *University of the Basque Country* (Faculty of Computer Sciences). The author enjoyed a doctoral grant from the Basque Government under the "*Researchers Training Program*" during the years 2006 to 2010. The work was co-supported by the *Spanish Ministry of Science and Education*, and the *European Social Fund* under contract MODELINE, TIN2008-06507-C02-01 and TIN2008-06507-C02-02. The views and conclusions contained herein are those of the author and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the University of the Basque Country, the Basque Government, the Spanish Ministry of Science and Education, or the European Social Fund.

To my parents, Caridad and José,
for their endless encouragement and patience,
¡Gracias!

Summary

Portlet standardization efforts, namely, the *Java™ Portlet Specification* (formerly JSR 286) and the *Web Services for Remote Portlets* (WSRP) promise to make portlets into universal and reusable plug-and-play components, making it possible to build a portal by plugging portlets from different vendors into a portal from any vendor. To make portlet reuse practical and effective, portlets need to offer some degree of variability to increase their capability to be tailored to the diversity of settings through which they might be delivered. So far, portlet standards account for variability by accessing and storing persistent configuration data and user profile parameters whose values are provided by the portal at runtime. These mechanisms imply that portlets are equipped with a number of variants, and are able, at runtime, to select between them. However, this might not be enough.

The portlet architecture distinguishes between the portlet provider and the portlet consumer. The provider builds the portlet, the consumer integrates the portlet. Providers and consumer might not coincide. This forces providers to foresee the different scenarios in which the portlet will be used. This is not easy. Currently available configuration mechanism falls short to account for the different needs that might arise when integrating portlets in third-party portals.

This thesis explores three scenarios where mechanisms provided by current portlet standards are not enough, namely:

1. the realization of Service-Oriented Architecture (SOA). Portals as integration platforms, offer an excellent conduit for realizing SOAs. The thesis propose the use of Software Product Lines techniques that account for portlets to become truly reusable services,

2. the provision of social tagging as a portal commodity. That is, tagging functionality is up to the portal but offered through portlets. To account of the portal specifics, portlet events and RDFa annotations are used to build a novel architecture that allows portlets to be seamlessly plugged into portal tagging infrastructure,

3. mashup-based personalization. Here, mashing is regarded as an additional personalization mechanism whereby portal users can supplement portal services with their own data needs. An architecture is presented where portlet providers facilitate mashups placeholders and XBL bindings are used to dynamically bound user mashups to them.

The thesis develops the theoretical underpinnings, and provides different implementations as a proof-of-concepts. All solutions are JSR-286 and WSRP compliant.

Contents

List of Figures

Chapter 1

Introduction

"A journey of a thousand miles begins with a single step."
– Lao Tzu.

1.1 Context

An **Enterprise Information Portal** (EIP) is a framework for integrating information, people and processes across organizational boundaries in a manner similar to the more general web portals (Wikipedia). One hallmark of enterprise portals (hereafter referred to as just "portal") is the de-centralized content contribution and content management, which keeps the information always updated. Another distinguishing characteristic is that they cater for customers, vendors and others beyond an organization's boundaries. This contrasts with a corporate portal which is structured for roles within an organization. Portals enable people to communicate and collaborate, providing a unified point of access to dynamic content from business applications, breaking down silos of content, and delivering information effectively through context-driven personalization. *A hallmark of portals is integration*. Rather

than providing its own services, a portal is also a conduit for external applications. So offered applications are technically known as portlets.

A **portlet** is a user interface Web component which is 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 delivery). Designed to achieve interoperability between portlets and portal platforms, the *Java™ Portlet Specification* [Jav03, Jav08] promises *write once, deploy anywhere* portlet development. This provides the infrastructure to make feasible a portlet market *à la COST*[1] 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*" [BKPS06]. Other similar initiatives include *Portlet Swap* (jboss.org) and *Liferay Marketplace* (liferay.com).

The *Web Services for Remote Portlets (WSRP)* specification [ftAoSIS03, ftAoSIS08] goes one step further and allows for the remote rendering of portlets. The WSRP mantra is *"deploy once, deliver anywhere"*. With WSRP, a portlet can be hosted ("produced") on physically and logically separate infrastructure from the portals surfacing ("consuming") the portlet. In so doing, WSRP becomes a feasible way to build federated portals, i.e., a network of interoperating portals, whereby resources hosted in one portal can be made available in many. From the above 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, differences stem from what is being reused (i.e., which includes the presentation layer) and where is the integration achieved (i.e., at the front end).

[1] COST stands for Commercial off-the-shelf.

1.2 General Problem

Software variability is the ability of a software system or artefact to be changed, customized or configured for use in a particular context. But, how much variability should be considered? Who is aware of the variability required? To answer these questions let us introduce an example.

Consider an airline company (e.g., IBERIA). It offers two services in terms of portlets: *searchFlight* and *bookFlight*. In this way, travel agencies (e.g., HALCON) can inject these portlets within their portals. Users (e.g., customers of HALCON) access the travel agency portal without being aware about *searchFlight* is being offered by a third party (i.e., IBERIA). Therefore, portlets are explicitly developed for and used in multiple portals, either locally deployed or remotely provided. The important point to note is that portlets might need to be integrated in diverse context and audiences. For instance, IBERIA offers its services not only to HALCON but also to other travel agencies (e.g., EROSKI). This means that *searchFlight* should also cater for the integration needs of EROSKI. But this is not enough. Even customers within a given portal might have different needs. For instance, Oscar is very apprehensive to weather conditions so that he looks at the weather forecast before setting the trip date. This just applies to Mr. Oscar, and it is not contemplated by *flightBooking*. Hence, Mr. Oscar is forced to move outside the portal realm to satisfy this data need (e.g., through a *weatherForecast* widget), and to bridge himself the passing of data from the portal to the widget.

The previous example serves to highlight the different contexts in which variability can be set and decided. Specifically, three "realms of decision" can be envisaged:

- the portlet provider (e.g., IBERIA). Here, the provider conducts a deep domain analysis to account for the variations among all the users and settings in which its portlets can be deployed.

- the portlet consumer (e.g., HALCON). Here, the consumer portal

conducts an analysis of its consumer base, and means are offered for the portal owner to configure the portlet appropriately.

- the end user (e.g., Mr. Oscar). No design can provide information for every situation, and no designer can include personalized information for every user. This is true for any web application, and portlets are not exception.

Worth mentioning, that these realms are not orthogonal but complementary. All, the portlet provider, the portlet consumer, and even, end users, should collaborate to obtain a better Web experience. Next sections delve into each of these scenarios.

1.3 Scenario 1: Provider-Based Variability

This scenario puts the variability burden on the provider's shoulders. The provider should conduct domain analysis to comprehend the different settings in which portlets are going to be deployed. This leads to the notion of the ***Consumer Profile***.

> *The Consumer Profile includes not only the consumer's platform (e.g., JBoss GateIn, Liferay, eXo Platform) but also presentation and functional requirements posed by the portal owner that needs to be catered for by the portlet producer.*

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. This makes even more

stringent to address portlet variability. To this end, we propose the use of software product line (SPL) development.

SPL development refers to software engineering methods, tools and techniques for creating a collection of similar software systems from a shared set of software assets using a common means of production (Wikipedia). 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, and hence realizing the so-called Service Oriented Architectures (SOAs)

A SOA is "an IT strategy that organizes the discrete functions contained in enterprise Service-oriented applications into interoperable, Architecture standards-based services that can be combined and reused quickly to meet business needs" - BEA Systems, Inc. SOA offers organizations greater agility, as they can quickly deploy new business processes or modified existing ones in response to marketplace changes. Portlet-based portals might provide a light-weight approach to SOA. Using portal terminology, the previous definition can be rephrased to define portlet-based portals as a SOA realization *that organizes the discrete functions contained in enterprise applications* [portlets] *into interoperable* [remote portlets can communicate and share data]*, standards-based services* [WSRP is built on Web Service standards] *that can be combined* [portals can aggregate information from multiple portlets] *and reused quickly* [WSRP does not require programming effort] *to meet business needs*. Although a growing number of organizations are moving to SOA many are having difficulties in identifying the first step in the process. *As organizations search for a way to leverage a service oriented architecture, portlet-based portals can provide a lightweight approach* [Phi05].

This SOA scenario not only requires portlet interoperability (through portlet standards) and portlet dissemination (through portlet repositories)

but also *portlet variability*. Portlets tend to be coarser grained than traditional Web services since they encapsulate the presentation layer as well as the functional layer. These coarse-grained components have fewer chances to be reused "*as-is*" [JGJ97]. This can jeopardize the vision of portlets as reusable services. This introduces the following research questions:

How SPL development can be applied to portlets?

Which would be the implications for the WSRP standard?

1.4 Scenario 2: Consumer-Based Variability

Portlet providers might not always foresee all needs, or some portlets might need some extensions that cannot be taken care for with just parameterization configuration. Social tagging is a case in point. Portals, to a bigger extent than other Web applications, have the notion of community deeply rooted inside its nature. Specifically, *Enterprise Information Portals* are borne to support the employees within an organization. Therefore, it is just natural to integrate social networking into these portals. Among social networking activities, we focus on tagging. Motivation for bringing tagging at the working place admits a two-fold perspective. From the company's viewpoint, tagging is an affordable approach to account for knowledge sharing and retention in the context of an organization, preventing leaking critical data outside the company [DMG+08]. From an employee's perspective, distinct studies [TSMM08, AN07, MYWF07] conclude that individual motivations for tagging at the working place, such as creating a personal repository or re-finding one's own resources, remained important.

In this scenario, taggers (i.e., the portal community) and tags are portal assets. However, and unlike self-sufficient tagging sites, portals could not hold the description of all tag-able resources. For instance, the description of the books or hotels offered through the portal could be

remotely kept by, e.g., Amazon and Expedia, respectively. This outsource of content description does not imply that the external resources are not worth tagging. This leads to distinguish between two actors: the *resource provider*, which keeps the content of the tag-able resources (e.g., book description in the case of Amazon), and the *resource consumer* (i.e., the portal), which holds the tagger community and the tags. In the same way, that portlets adapt their rendering to the aesthetic guidelines of the hosting portal, tagging through portlets should also cater for the peculiarities of the consumer portal. This introduces the following research questions:

> *How portlets can be made aware of portal specificities, specifically, portal-based tagging?*

1.5 Scenario 3: User-Based Variability

Personalization is the process of tailoring pages to individual users' characteristics or preferences that will be meaningful to their goals. It works on the theory that each user has unique interest and needs. Unfortunately, users typically have no influence on choosing which content can be personalized or how it can be manipulated. Furthermore, it is not always easy for the designer to foresee the distinct utilization contexts and goals from where the application is accessed. "*No design can provide information for every situation, and no designer can include personalized information for every user*" [Rho00].

Not so long ago, only developers could create applications. Users could only use what developers have created before. However, things have changed. With mashups, users can avoid the IT department and assemble their own applications in response to their own individual needs. A mashup is a lightweight Web application created by combining information and capabilities from more than one existing source to deliver new functions and insights. New powerful tools are currently available (e.g., *Software AG Presto*, *IBM Mashup Center*, *Kofax Kapow*) that require little or

no IT involvement, and allow savvy users to create the mashup they need, whenever they need it. Enabling users in this way can reduce the application backlog on the IT department, development time and costs and lower the cost of customizing information so individuals can adapt information easily into exactly the form they need it [CDG+08]. By taking advantages of mashups features, a portal could become a highly-dynamic yet personalized environment. This introduces the following research questions:

How mashups can be introduced in portlet-based portals?

1.6 Contributions

This thesis has been developed in the context of engineering, which pushed us to achieve not only an academic contribution but also to look at the broader applicability of these ideas. To this end, we provide a realization (implementation) of the ideas we describe here. In our opinion, this thesis makes the following contributions:

Provider-Based Variability

This work promotes the vision of portlets as reusable services by introducing the notion of *Consumer Profile* 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 WSRP-compliant architecture that let portlet consumers registry and handle "family portlets" in the same way that "traditional portlets".

Consumer-Based Variability

It advocates for means to better account for the portlet consumer specifics. Tagging is used as an example. It argues for tagging to be orthogonally supported as a crosscut on top of portlets, i.e., as a portal commodity.

The main challenge rests on consistency at both the back-end (i.e., use of a common structure for tagging data, e.g., a common set of tags), and the front-end (i.e., tagging interactions to be achieved seamlessly across the portal using similar rendering guidelines). Portlet events and *RDFa* annotations are used to meet this requirement.

User-Based Variability

The presented approach introduces mashups as an additional personalization mechanism whereby portal users can supplement portal services (i.e., portlets) with their own data needs. The approach is realized for *Liferay* as the portal engine, portlets as the realization of portal services, and *XBL* as the integration technology.

1.7 Design-Science as Research Approach

Design science is "the scientific study and creation of artefacts as they are developed and used by people with the goal of solving practical problems of general interest" [JP14]. In additional quote from P. Johannesson and E. Perjons, brilliantly summarise the essence of this approach:

> *The starting point for a design researcher is that something is not quite right with the world, and it has to be changed. A new artefact should be introduced into the world to make it different, to make it better. Design science researchers do not only think and theorise about the existing world. They model, make, and build in order to create new worlds. They produce both a novel artefact and knowledge about it and its effects on the environment. In particular, they need to formulate problem statements, determine stakeholder goals and requirements, and evaluate proposed artefacts. In other words, artefacts as well as knowledge about them are research outcomes for design science.*

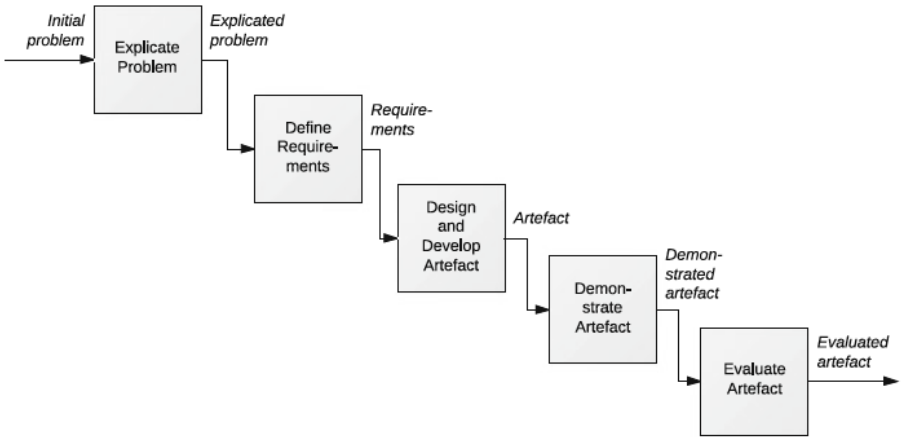


Figure 1.1: Overview of the method framework for design science (taken from [JP14])

Specifically, we follow the framework defined in [JP14]. For self-contained, next paragraphs are an excerpt of [JP14] where the different tasks of their Design Science methodology are described (see Figure 1.1):

- **Explicate Problem.** The Explicate Problem activity is about investigating and analysing a practical problem. The problem needs to be precisely formulated and justified by showing that it is significant for some practice. The problem should be of general interest, i.e., significant not only for one local practice but also for some global practice. Furthermore, underlying causes to the problem may be identified and analysed.

- **Define Requirements.** The Define Requirements activity outlines a solution to the explicated problem in the form of an artefact and elicits requirements, which can be seen as a transformation of the problem into demands on the proposed artefact.

- **Design and Develop Artefact.** The Design and Develop Artefact activity creates an artefact that addresses the explicated problem and fulfils the defined requirements. Designing an artefact includes

determining its functionality as well as its structure.

- **Demonstrate Artefact.** The Demonstrate Artefact activity uses the developed artefact in an illustrative or real-life case, sometimes called a "proof of concept", thereby proving the feasibility of the artefact. The demonstration will show that the artefact actually can solve an instance of the problem.

- **Evaluate Artefact.** The Evaluate Artefact activity determines how well the artefact fulfils the requirements and to what extent it can solve, or alleviate, the practical problem that motivated the research.

As indicated by P. Johannesson and E. Perjons, these tasks do not follow strictly in sequence. Rather, research is commonly iterative, moving back and forth between all the activities of problem explication, requirements definition, development, and evaluation. The arrows in Figure 1.1 should not be interpreted as temporal orderings but as input–output relationships. In other words, the activities should not be seen as temporally ordered but instead as logically related through input–output relationships.

1.8 Outline

This section provides a brief summary of each chapter of this dissertation. Figure 1.2 presents a chapter map to help to put each of them in context.

Chapter 2

This chapter introduces the background (*Enterprise Information Portals*, *Portlets* and *WSRP*) on top of which the remaining chapters are developed.

Chapter 3

This chapter introduces the notion of "portlet families" and discusses the factors that must be considered when introducing variability in a family of portlets.

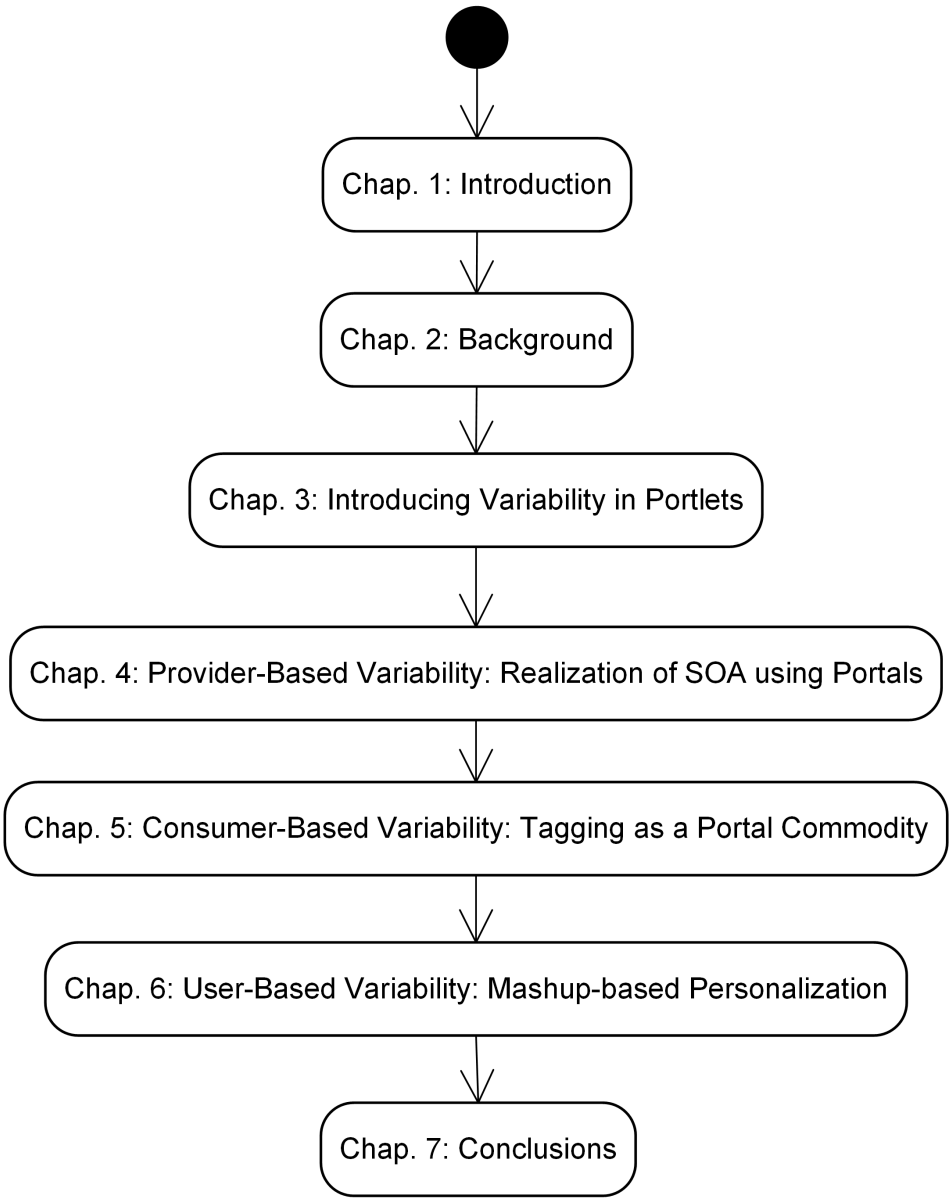


Figure 1.2: Chapter Map

Chapter 4

This chapter tackles scenario 1.3. It promotes the vision of portlets as reusable services by introducing the notion of *Consumer Profile* as a way to capture the distinct organization scenarios where a portlet can be deployed.

Chapter 5

This chapter tackles scenario 1.4. It advocates for means to better account for the portlet consumer specifics. Tagging is used as an example. It argues for tagging to be orthogonally supported as a crosscut on top of portlets, i.e., as a portal commodity.

Chapter 6

This chapter tackles scenario 1.5. It introduces mashups as an additional personalization mechanism whereby portal users can supplement portal services (i.e., portlets) with their own data needs.

Chapter 7

This chapter concludes the dissertation. It summarizes the obtained results, makes an assessment and also identifies future research topics that this work raised.

1.9 Conclusions

The intention of this chapter was to give an overview of the contents of this dissertation. The topic was introduced and what, in our opinion, are its contributions were listed. The next chapter starts with a review of the background.

Chapter 2

Background

2.1 Overview

The proliferation of Web applications has raised the need for a new kind of application to tie these disparate Web applications together into aggregated applications—the Web portals. As soon as the need was identified, there were many portal vendors claiming their products to be the solution to such a need. This first generation of portals had entirely proprietary APIs.

At the same time, J2EE was having a great success as a platform for server programming in the Java™programming language. Quickly, portal vendors began to release non-standard extensions to J2EE. However, these portal-specific extensions were against the portability of enterprise applications, which was considered the characteristic of the J2EE that had most contributed to its success.

IBM and Sun recognized this problem and, after having launching a proposal for standardization separately, they reached an agreement to combine both proposals into JSR 168 (a.k.a. the Java™Portlet Specification version 1.0) [Jav03]. JSR 168 was adopted by most Java™-based portal vendors.

However, the widespread use of the technology soon brought new requirements for new functionality that was not addressed in JSR 168. This situation leads, once again to the detriment of portability, to the emergence of new portal-specific extensions to JSR 168. JSR 286 (a.k.a. the Java™Portlet Specification version 2.0) [Jav08] tries to solve this problem by adding that functionality most requested by developers.

Alongside the JSR 168, a related standard was developed—the *Web Services for Remote Portlets Specification* (WSRP) [ftAoSIS03]. The WSRP specification is motivated by the need of portals to aggregate not only local content, but also content provided by external content hosts. Traditionally provided by data-oriented Web services, this content requires aggregating applications (e.g., a portal) to provide specific presentation logic. This approach is not well suited to dynamic integration of content as a plug-and-play solution. To solve this problem, the WSRP specification introduces a presentation-oriented Web service interface that provides both business logic and presentation logic. Being related specifications, the release of the JSR 286 implied the release of a new version of the WSRP specification [ftAoSIS08].

This chapter provides a *quick glance* at Web portals, portlets and WSRP. We invite the reader to skip this chapter if familiar with those concepts.

2.2 Enterprise Information Portals

It is quite difficult to define accurately what a "Web portal" is. This term has been overused and it takes on a somewhat different meaning depending on the viewpoint of the stakeholder. But, in general terms a portal is just a gateway, and a Web portal can be seen as a gateway to the information and services on the Web. Tatnall et al. [Tat05] defines a Web portal as a special Internet (or intranet) site designed to act as a gateway to give access to other sites. It should be seen as providing a gateway not just to sites on the Web, but to all network-accessible resources, whether involving intranets,

extranets, or the Internet. In other words, a Web portal provides users a single access point to all the types of information.

The term "Web portal" probably has its origins in search engine sites such as *Yahoo!*, *Lycos*, and so on, which can now be classified as the first generation Web portals. Apart from the search engine feature, these first generation Web portals quickly evolved into sites offering other services such as e-mail, news, stock quotes, community building, and so on. Web portals may be horizontal or vertical in nature [Str02]. *Horizontal portals* are intended to be accessed by a broad base of users, whereas *vertical portals* focus on a particular audience.

A kind of vertical portals that have grown to be widely popular over the last few years is the so-called *Enterprise Information Portals* (a.k.a. *Corporate Portals*). The term Enterprise Information Portals is applied to the gateway to the corporate intranets that are used to manage the knowledge within an organization. Primarily designed to automate business-to-employee (B2E) processes, they can be seen as a customized home page or desktop that offers employees the means to access and share data and information within the enterprise. In this dissertation, when speaking of portal it means Enterprise Information Portal.

Features of a Portal

The following list presents those features commonly offered by portals [RAV+04].

- **Aggregation of content**. It refers to the capability of portals to aggregate content from different sources and to present it into one consistent and interoperating view.

- **Customized views**. Different users accessing the same URL may get different views depending on the role of the person in the organization.

- **Personalized content**. Personalization takes customization one

step further and allows individual users to tailored their views to individual users' characteristics or preferences that will be meaningful to their goals.

- **Unified security model**. Users have only one account for all different systems aggregated by the portal. This provides not only *single sign-on*[1], but also an enterprise-wide security policy based on role.

- **Collaboration features**. Collaboration is mainly concerned with building communities of interest, whereby portal's users can share common knowledge and insight on a particular set of data.

- **Localization**. Localization implies to adapt the content to the locale in which it is being presented, including the language of the user interface and features such as date format, currency exchange, and so on.

- **Internationalization**. Internationalization is the process of designing an application or portal in such a way that it can be localized without engineering changes.

- **Web services access**. Web services are a widespread approach to provide services to third-parties and to consume them. This feature refers to the ability of a portal to consume and provide Web services.

- **Workflow**. A workflow consists of a sequence of connected steps. In the case of portals, it allows users to seamlessly move through a set of tasks across multiple data sources and applications.

- **Self-service**. Self-service is the practice of serving oneself. For example, in many gas stations, customers pump their own gas rather than have an attendant do it. In the case of portals, self-service means

[1] *Single sign-on* allows portals to pull all of your different systems together and make them available by logging in just once.

that the user should conduct transactions easily with minimal or no support from other people.

- **Client agnostic processing**. Portals should be browser-, device-, and platform-independent.

The above list does not include all the features we could found in a portal. Neither the minimum set of features that any portal must offer. Its main goal is to give an overview of the types of features we can expect to find in a portal.

2.3 Portlets

With the trend towards more portal solutions also came a growing list of portal-specific APIs that developers had to learn. The Java™Portlet Specification [Jav08] defines a common API for developing portal applications that can work on any portal. JSR 286-compliant portal applications are known as portlets.

A portlet is an application that provides a specific piece of content (information or service) to be included as part of a portal page. The content generated by a portlet is also called a *fragment*. Being user-facing, portlets generate markup fragments (e.g., HTML, XHTML, WML) rather than data-oriented XML. The content of a portlet is normally aggregated with the content of other portlets to form the portal page. Figure 2.1 shows a sample portal page containing four portlets.

Portlets are multi-step applications, i.e., they encapsulate a chain of steps rather than a one-shot delivering. The life cycle of a portlet is managed by a portlet container, which is responsible for initializing and destroying the portlet. After a portlet is properly initialized, the portlet container may invoke the portlet to handle client requests. The *Portlet* interface defines two methods for handling requests, the *processAction* method and the *render* method. Additionally, the *EventPortlet* and *ResourceServingPortlet* interfaces define the additional life cycle methods

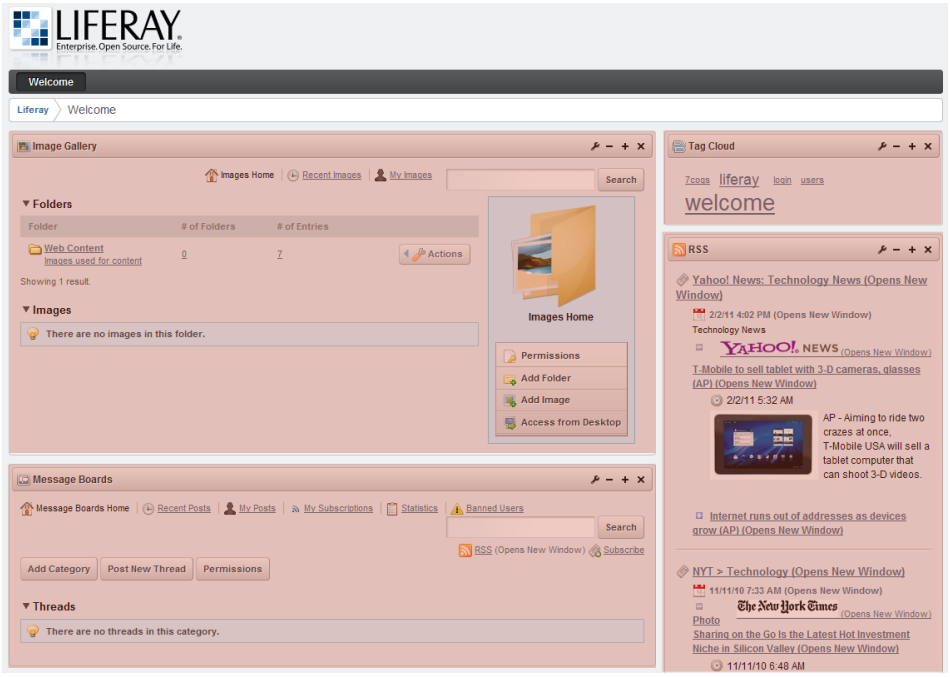


Figure 2.1: Sample portal page with portlets.

processEvent and *serveResource*. Figure 2.2 presents the life cycle of an action request.

Normally, users interact with content produced by a portlet, for example by following links or submitting forms, resulting in a portlet action being received by the portal (1). Then, the portal requests the portlet container to invoke the portlet to process the action (2), which ends with the portlet container executing the request on the hosted portlet (3). As a result of an action, the portlet may publish events (4), which result in one or more invocations of the *processEvent* method of this or another portlet (5). After the event processing is finished, portal invokes the *render* method for all portlets in the portal page (6), each of them returning a fragment. Finally, the portal builds a "quilt page" out of these fragments, and renders it back to the user (7).

At this point, it is important to notice two things. Firstly, portlet can react to actions or state changes not directly related to an interaction of the

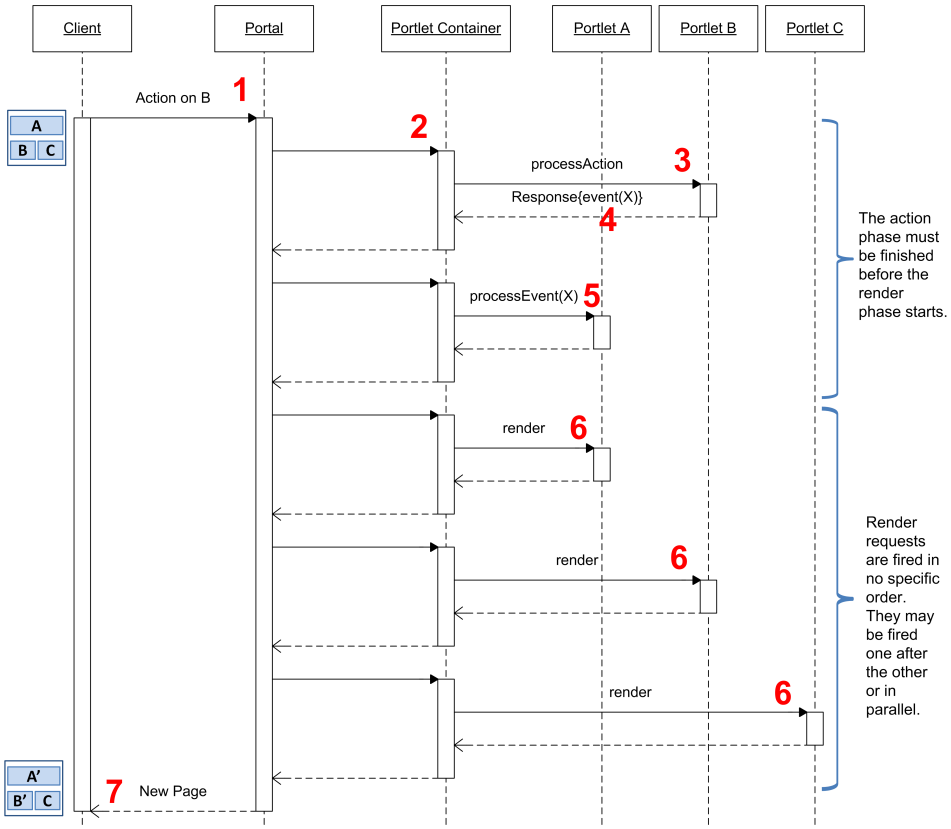


Figure 2.2: Request handling sequence.

user with the portlet. This is the case of portlet A in figure 2.2. Secondly, the life cycle of an action request is split into two parts: the action processing phase, which includes the event processing phase, and the rendering phase. This provides a clean separation of the action semantics from the rendering of the content.

Additionally, in order to serve resources or render content fragments via the portlet, the portlet can implement the *ResourceServingPortlet* interface and create resource URLs that will trigger the *serveResource* method on this interface. The portlet container does not render any output in addition to the content returned by the *serveResource* call. The *serveResource* call can be used to implement Ajax use cases.

Adaptive/Adaptable Portlets

Normally, portlets perform different tasks and create different content depending on the function they are currently performing. Such a function is indicated by the **portlet mode**. A *portlet mode* advises the portlet what task it should perform and what content it should generate. For instance, when in the *VIEW* mode, the portlet renders fragments which support its functional purpose. 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 and the *HELP* mode, where a portlet may provide help screens that explain the portlet purpose, and its expected usage.

The mode example illustrates how portlets can adapt their behavior to the function they are currently performing. Moreover, portlets should also decide how much information they should render depending on the amount of portal page space they have assigned. The **window state** indicates the portlet the available space. The *NORMAL* window state indicates that a portlet may be sharing the page with other portlets. The *MAXIMIZED* window state is an indication that a portlet may be the only portlet being rendered in the portal page. And, when a portlet is in *MINIMIZED* window state, the portlet should only render minimal output or no output at all.

Both *portlet mode* and *window state* can be change programmatically by the portlet or manually by the portal user. To this end, content generated by portlets is usually enclosed in a decorator provided by the portal. This decorator is the object responsible for providing the controls that allow a portal user to manually change these values (see figure 2.3).

Portlet not only adapt their behaviour to the *portlet mode* and the *window state*, but they can also be adapted to the user profile[2], initialization parameters, and additional portlet-specific data collected as portlet preferences.

[2]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.

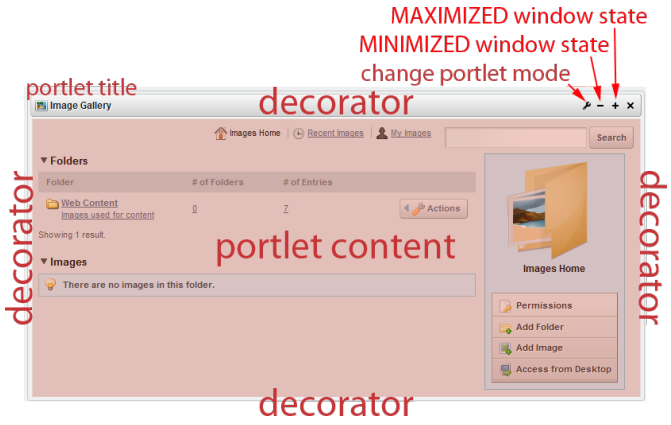


Figure 2.3: A sample portlet.

These **portlet preferences** provide a parameterization-based mechanism to adapt the portlet. They are commonly configured to provide a customized view or behavior for different users. For example, a weather portlet that displays the weather of a specific location could offer the location as a portlet preference in order to allow individual portal users to select the location he or she wants. 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).

2.4 The WSRP Specification

The WSRP specification [ftAoSIS08] standardizes the Web service interface a portlet producer must implement to allow another application (typically a portal) to consume its portlets. Such an interface defines a protocol that decouples portlet producers from portlet consumers. Therefore, the main actors involved here are **Producers** (web services conforming to this specification) and **Consumers** (applications consuming Producers in a manner conforming to WSRP specification).

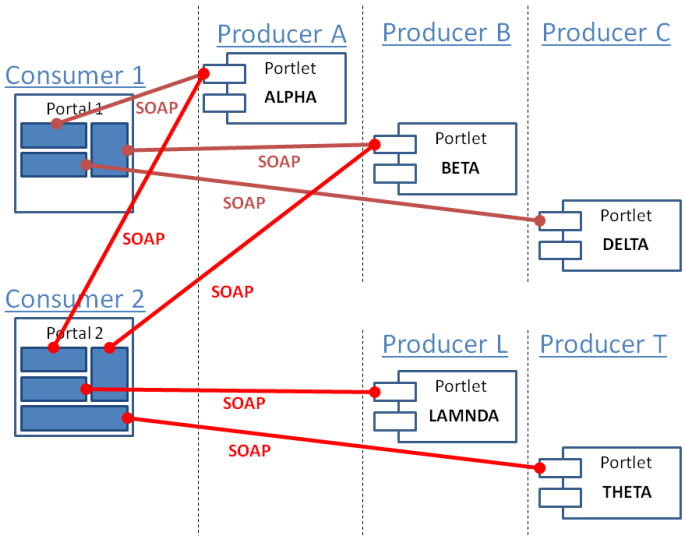


Figure 2.4: Portlet producers and consumers.

The WSRP specification defines four interfaces:

- *Service Description*. The *getServiceDescription* operation provides a discovery mechanism-agnostic means for a *Consumer* to ascertain a *Producer*'s or portlet's capabilities.
- *Markup*. This interface offers operations to request the generation of markup and the processing of interactions with that markup.
- *Registration*. A registration describes a relationship between a *Consumer* and a *Producer*. This interface allows *Consumers* to establish these relationships.
- *Portlet Management*. This interface provides the operations that allow *Consumers* to clone and customize the portlets the *Producer* offers.

It provides the infrastructure to make feasible a portlet market *à la COST*[3] so that portals can deliver portlets being provided by third parties (see figure 2.4).

[3]COST stands for *Commercial off-the-shelf*.

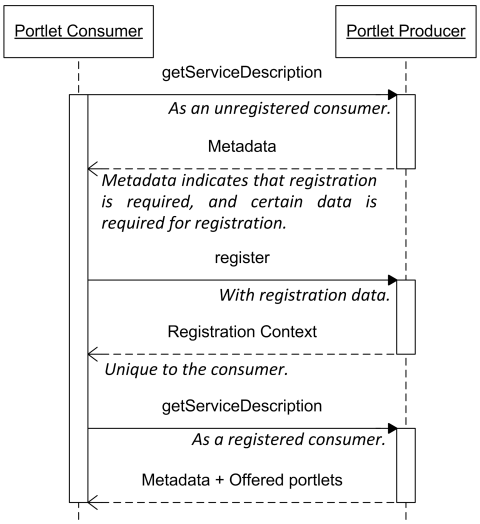


Figure 2.5: WSRP protocol: portlet Consumer registration.

The interaction among these actors goes as follows. First, portal registration is achieved by the portal administrator normally through a portal IDE (e.g., Liferay, eXo Platform, etc.), and ends up with a portal being registered to a given portlet producer. Figure 2.5 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.

Once registered, the portal is ready to engage the portlet in conversation to deliver its service. Continuing with the example introduce in figure 2.2, figure 2.6 presents the life cycle of an action request, but now the portlet *B* is remotely provided using WSRP. 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. But no markup is returned to the

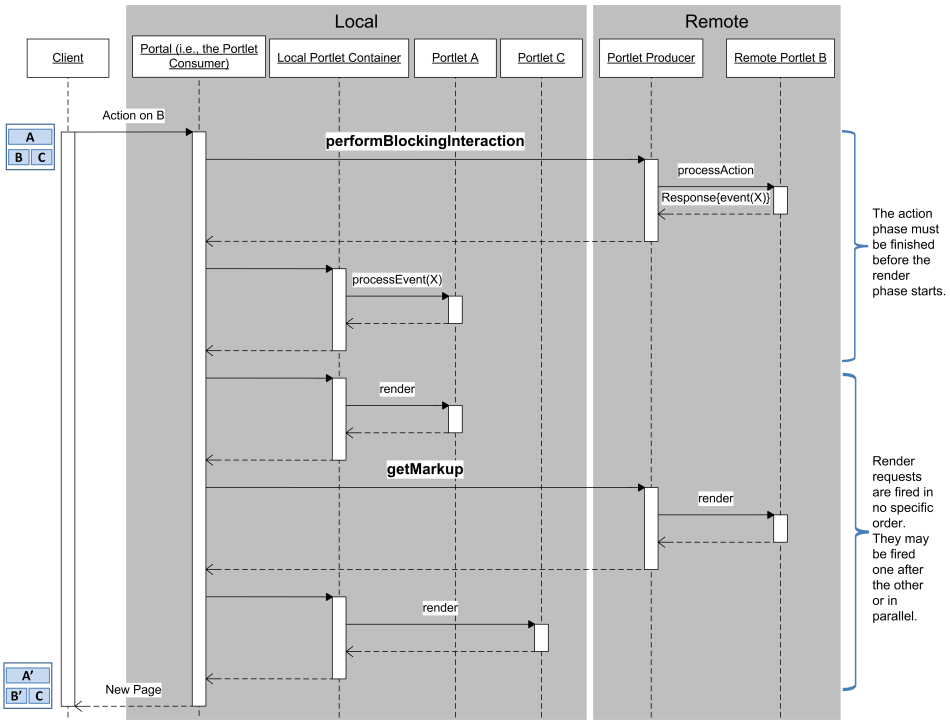


Figure 2.6: WSRP protocol: request handling sequence.

consumer. This requires the consumer to issue a *getMarkup()* to recover the eventually new markup associated with this new state. Thus, the separation of the action semantics from the rendering of the content, established by the portlet specification (see section 2.3), is kept untouched.

2.5 Conclusions

The purpose of this chapter was to provide a brief introduction to the background on top on which this work is built on, namely:

- Web portals
- Portlets
- WSRP

The interested reader can find further details of the covered topics in the references.

Chapter 3

Introducing Variability in Portlets[1]

3.1 Overview

A portlet is a reusable software artefact that could be simultaneously used in different shapes by different portals. *Portlet reusability* is defined as the capability of the portlet to be used in different portals [MCnP+05]. However, portlets tend to be coarse-grained software artefacts since they encapsulate the presentation layer as well as the functional layer. These coarse-grained components have fewer chances to be reused "as-is" and this can jeopardize the vision of portlets as reusable software artefacts. Of course, portlets can be used "*as-is*" when they match the problem. However, slightly different requirements could make such a portlet unsuitable for the system at hand. An important lesson learned by the software reuse community is that any reusable software artefact must provide the 'right' abstractions and the 'right' level of *variability*

[1]This chapter is not part of the contribution of this dissertation; it is based on [SvGB05] and its goal is to contextualise next chapters.

[JGJ97, Bos00].

Handling variability implies engineering core artefacts 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. 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. Therefore, there is a shift from developing individual portlets to create a portfolio of closely related portlets with controlled variations.

Development of portlet families relies heavily on the use of variability to manage the different between individual portlets. A key issue is how this variability is realised (a.k.a. *variability realization techniques*). Many variability realization techniques have been proposed over the years [JGJ97, SvGB05] and choosing an appropriated technique is not a trivial task as research shows [JRLR00, GBS01, CN01, BFG+02, JGJ97, JB02]. The portlet engineer needs to choose the techniques that provide support for the different patterns of variation that are encountered during the portlet development life cycle. This chapter discusses the factors that must be considered when introducing variability in a family of portlets in order to choose an appropriate variability realization technique.

The rest of the chapter is structured as follows. Section 3.2 discusses the factors they need to consider in order to choose an appropriate variability realization technique. Finally, some conclusions end this chapter.

3.2 Introducing Variability in a Family of Portlets

Firstly, substantial portlet reuse requires that commonality and variability be identified. This variability needs then to be constrained to that which is actually required to support the current and future, planed needs of the portlet at hand. Once variability has been identified and constrained, it must be implemented. The last step is to manage the variability, for example, to extend functionality by adding new variants to cover future user requirements, pruning those variants that are no longer used and so on. Therefore, to introduce variability in portlets, it is necessary to perform a minimal number of steps, namely, variability identification, variability constriction, variability implementation and variability management [SvGB05] (see figure 3.1).

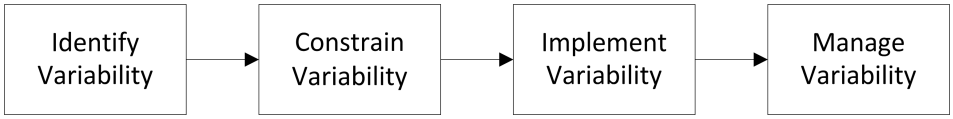


Figure 3.1: Steps for introducing variability in portlets.

Variability is concerned with all types of software development artefacts such as requirements model, architecture design, software components, test plans, etc. This dissertation is focused on variability in software components, more specifically, in portlets.

3.2.1 Identification of Variability

The identification of variability is a research field in its own right [CN01, Bos00] and it is outside the scope of this dissertation to study it in depth. However, when discussing variability, it is convenient to introduce the term *feature*. Bosch et al. [Bos00] defines a feature as "*a logical unit of behaviour that is specified by a set of functional and quality requirements*". Features can be mandatory, alternative, optional

or external; let me illustrate it using an example. Consider a family of portlets to book flights. Every portlet in such a family might share common characteristics. For example, the flight search engine might be the same for all portlets (i.e., a *mandatory feature*); every booked flight must have a travel insurance added; however, insurance company can vary from one portlet to another (i.e., an *alternative feature*); some portlets might accept payments from PayPal and others might not (i.e., an *optional feature*); finally, portlets from this family could be accessible and usable to people with disabilities if browser rendering them supports assistive technologies such as WAI-ARIA[2] (i.e., an *external feature*). External features are those "*features offered by the target platform of the system*" [GBS01]. They are not part of the portlet but they are important because they are used by the portlet which depends on them. Travel agencies that want to buy a portlet of this family have no choice of flight search engine; however, they can choice the insurance company they would like to work with and decide whether to accept payments from PayPal or not. On the other hand, all portlets from this family will or will not be more accessible to people with disabilities depending on the browser rendering them.

Features are organized into the so-called *feature diagrams*. A feature diagram represents a hierarchical decomposition of features and their character, that is, whether they are mandatory, alternative, optional or external (see, e.g., FODA [KCH+90], FORM [KKL+98], and FeatuRSEB [GFdA98]). Figure 3.2 shows the feature diagram corresponding to the flight booking portlet family described above using a notation derived from FeatuRSEB which is introduced in [GBS01].

[2]WAI-ARIA, the Accessible Rich Internet Applications Suite, defines a way to make Web content and Web applications more accessible to people with disabilities.

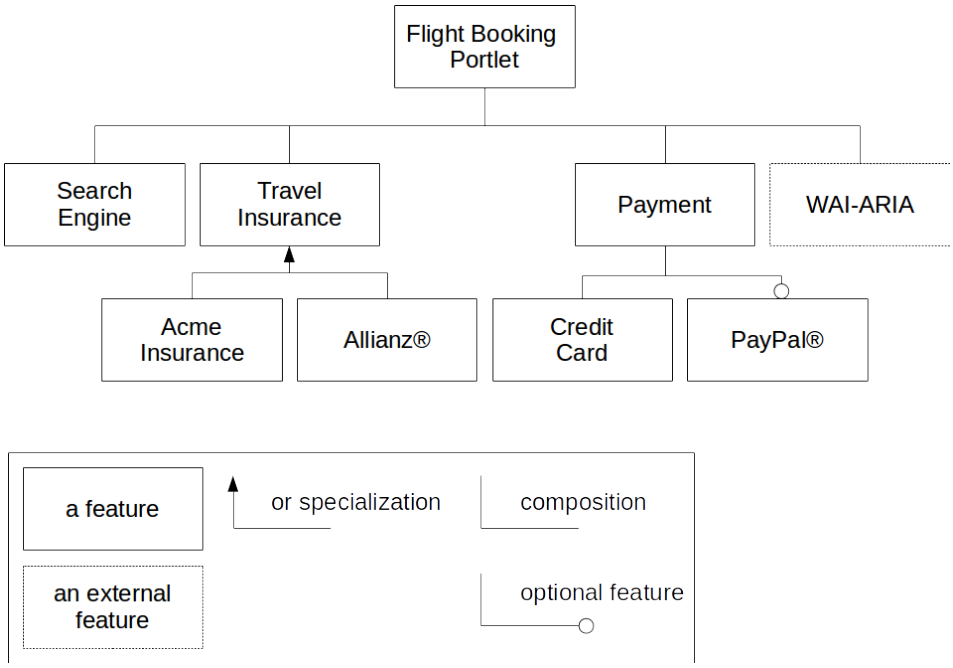


Figure 3.2: Feature diagram of the flight booking portlet family.

Notice that by modelling a family of portlets using features, variations between members of the family is expressed in terms of alternative and optional features. Thus, the process of identifying variability consists of listing those varying features.

3.2.2 Constraining Variability

After having identified a variable feature (i.e., an alternative or optional feature), the next step is to constraint it. If our family of portlets is too large and family members vary too widely, family could collapse into the old-style one-at-a-time portlet development effort and then benefits of developing portlets as a family will be lost [CN01]. By constraining the variability we enable an informed decision on how to implement the variable feature in the family of portlets [SvGB05].

When implementing a variable feature, portlet developers need to consider a number of factors in order to select an appropriate variability

realization technique. These factors can be identified by considering the life cycle of the variable feature. As figure 3.3 shows, a variable feature goes through different states until there is a decision on which variant to use.

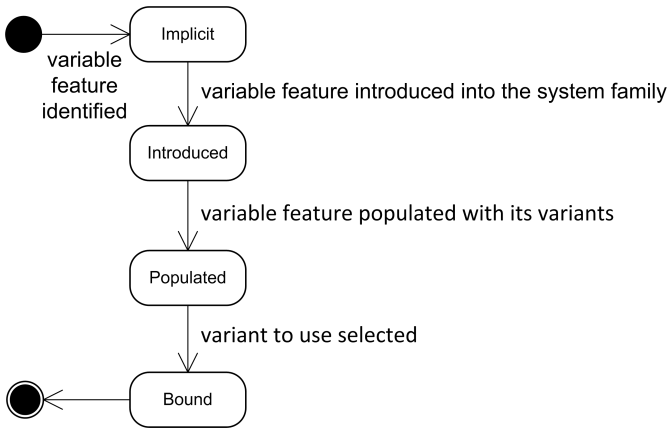


Figure 3.3: Variable feature life cycle.

Let's take a closer look at each of these states.

- *Implicit*. As stated in section 3.2.1, the identification of variability is the first step to introduce variability in a family of portlets. Variable features identified in 3.2.1 it is said to be implicit. Implicit variable features only exist as concepts; they are part of the outcome of 3.2.1 but are not yet implemented.

- *Introduced*. Variable features exploit at variation points. A variation point "*identifies one or more locations at which the variation will occur*" [JGJ97]. Each variable feature may be implemented by one or more variation points. A variable feature is introduced in the family of portlets when it has already been decided how to implement it, i.e., variation points to realize the variable feature are introduced in the family. Notice that variants may not be present at this time.

- *Populated*. Once a variable feature is introduced the next step is to populate it with its variants. This means that, for each variation point, variants are implemented.

- *Bound*. To create a particular member of a family of portlets, we have to make the decision of which variant is used for each variation point. That is, the family of portlets is bound to one of the variants for a particular variable feature.

Only the last three states are of importance for constraining a variable feature. These states are detailed in the following sections.

Throughout this life cycle, decisions need to be made in each state change. Svahnberg et al. [SvGB05] identifies three groups of stakeholders responsible for making such decisions.

- *Domain engineers*. People that are responsible for designing and implementing the family of portlets. Domain engineers are those who identify mandatory, alternative and optional features, introduce variation points and populate them with its variants. In a WSRP scenario, this role is played by the *portlet provider* that conducts a deep domain analysis and provides *portlet consumers* with a fixed set of variants to choose from.

- *Application engineers*. This role is played by the *portlet consumer*, which is responsible for deriving members from the family of portlets developed by the *portlet providers*. That is, in a WSRP scenario, *portlet consumers* are in charge of binding variable features to one of its variants.

- *End users*. The clients of the *portlet consumers*, people that use the portlets created by *portlet consumers*.

Introducing a Variable Feature

To introduce a variable feature into a family of portlets means to create the set of variation points necessary to implement all of the variants for

that feature. The decision on when to introduce a variable feature is governed by a number of things, such as: size of software entities, number of variation points and cost of maintaining a variable feature [SvGB05].

Components, class packages, single classes, lines of code or even a combination of all of them are valid ways to implement variants of a variable feature. When speaking of software entities it refers to these different implementation artefacts. *Size of software entities* has not a direct connection to how to implement the variation points; however, it determines the overall strategy of how to implement the variable feature. During the portlet development process, portlets goes through different development phases (see figure 3.4). Different software entities are mostly likely to be considered during different development phases. For example, components are in focus during architecture design whereas lines of code are considered during implementation and compilation phases.

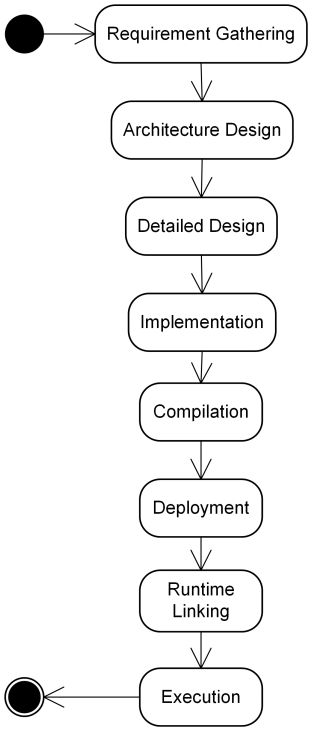


Figure 3.4: Portlet development activities.

On the other hand, a single variable feature may be implemented by one or more variations point. Similarly, each variation point can be populated with one or more variants. A thousand variation points may imply several thousands of variants. So, it is recommendable to maintain the *number of variation points* corresponding to each single variable feature as small as possible in order to increase understandability of the source code, facilitate maintenance and reduce the risk of introducing bugs.

Additionally, the *cost of maintenance* is a factor to take into account. Variable features should be added no too early, but no later than is needed either. It is typically more expensive (computationally) to have a variable feature that work during runtime. However, the earlier we add a software entity, the higher is the cost of maintaining it. If we introduce a software entity during the architecture design, we need to consider it during architecture design, detailed design, implementation, compilation, deployment and linking (see figure 3.4).

Populating a Variable Feature with Variants

Once the variation points have been created the next step is to populate them with their variants. When populating a variable feature we need to consider three factors, namely, *when*, *how* and *who* [SvGB05].

When it is possible to populate. We can add new variants to a variation point only during some phases of their life cycle. For these phases it is said the variation point is *open*, and *closed* during all other. For example, variation points implemented using lines of code are usually open during detailed design and implementation but close during all the others. When a variation point is open or closed is determined from both a technical perspective (e.g., lines of codes are limited to detailed design and implementation phases) and as a design decision (e.g., we desire a variation point to be open during runtime).

How to populate. Population can be done *implicitly* or *explicitly*. Implicit population means that the variation point has no knowledge of

the available variants, whereas with explicit population there is an explicit list of all variants available. Being aware of all of the possible variants (i.e., explicit population), the portlet could discern between them and by itself select a suitable variant during runtime. However, implicit population requires the intervention of a third-party (e.g., the *portlet consumer*) to specify which variant to use.

W*ho populates the variable features with variants*. In order to select the appropriated mechanism, it is crucial to take into account who is allowed to populate a variable feature. Commonly, *portlet providers* provide the *portlet consumers* with a fixed set of available variants to choose from. However, sometimes *portlet providers* allow *portlet consumers* to create and add their own product-specific variants. Furthermore, there is an increasing tendency to provide variability to end users (e.g., plug-in mechanisms). Depending on this factor some techniques are more suitable than others.

Binding to a Variant

At this point of the life cycle of a variable feature, variation points have been introduced and populated with their variants. The last step is to select a variant for a variable feature (a.k.a. *binding*). The decisions to make here are *when* to bind and *how* to bind [SvGB05].

When to bind. Binding can be done at several stages during the portlet development and also at runtime.

- *Product architecture derivation*. The architecture of a particular portlet is derived from the architecture of the portlet family. The design of an architecture of a family of portlets usually includes many unbound variation points in order to support variability. A portlet architecture is obtained by binding these variation points to a particular variant.

- *Compilation*. During compilation the source code is transformed into Java^{TM} bytecode. This may include extending the code to

add new behaviour using code superimposition techniques such as aspect-, feature-, and subject-oriented programming.

- *Deployment*. Portlet are no standalone applications, every portlet is deployed inside a portlet container that controls the life cycle of the portlet and provides it with necessary resources and information about its environment. A portlet container is responsible for initializing and destroying portlets and also for passing user requests to it and collecting responses. The behaviour of portlets can be customised during the initialization process by using initialization portlet parameters.

- *Linking*. The Java™Virtual Machine dynamically loads, links and initializes classes and interfaces. Linking is the process of taking a class or interface and combining it into the runtime state of the Java™Virtual Machine so that it can be executed. The dynamic nature of the Java™linking phase make possible technologies such as OSGi [The14] that supports the hot replacement of *jar* libraries. These technologies can be used to implement variability using component-driven development techniques.

- *Runtime*. This phase refers to the period during which a portlet is running by the portlet container. Variations points can be open for population at runtime (e.g., to provide variability to end-users). Variants added during runtime are commonly referred as plug-ins and are usually developed by a third party.

At this point, it should be note that for a variable feature that exploits at many variation points, variation points need to be bound either at the same time, or the binding of several variation points is synchronized so that a variation point that is bound during compilation binds to the same variant that related variation points have already bound to during product architecture derivation.

When considering when to bind, generally, the later the binding is done the more costly (e.g., in term of performance) it is. For example, to enable binding at runtime implies the portlet have to include extra functionality to support it, which, in turn, involves a cost in term of performance to conduct the binding.

How to bind. Binding can be done *internally* or *externally*. Internal binding means that the portlet itself contains the functionality to bind to a particular variant, whereas external binding implies a third-party is responsible for performing the actual binding. If combined with implicit and explicit population, we obtain four different options.

- *Implicit population and internal binding*. The portlet has no knowledge of the available variants, but it contains the functionality to perform the binding by itself. The *XMLHttp* requests are an example of this. *XMLHttp* requests allow JavaScript developers to initiate HTTP request from anywhere in an application. However, Microsoft's implementation prior to Internet Explorer 7 is an ActiveX control, whereas the other browsers use a native JavaScript object—the *XMLHttpRequest* object. If the portlet at hand has to make use of such functionality, despite it does not need to explicitly manage the set of variants, it should be able to select one of the variants by itself each time it needs to initiate an HTTP request. This type of variation point is commonly bound during runtime.

- *Implicit population and external binding*. The portlet has no knowledge of the available variants and it is a third-party who is responsible for performing the actual binding. For example, a portal that can be bundled with different application servers (e.g., Apache Geronimo[3] and GlassFish[4]). After being created by a build tool (i.e., the third-party in charge of performing the binding), the portal itself does not need to know it could be bundled with another application

[3] http://geronimo.apache.org/
[4] https://glassfish.dev.java.net/

server as well. This combination is the most commonly used to bind variations point in the development phases, i.e., all phases except runtime and dynamic linking.

- *Explicit population and internal binding*. There is an explicit list of all variants available and the portlet contains the functionality to perform the binding by itself. For example, an RSS feed reader portlet that allows an end-user to specify which RSS feed should be used by default, and where the set of available RSS feeds is not fixed, but it can be extended at runtime. This variation points are typically both populated and bound at runtime.

- *Explicit population and external binding*. There is an explicit list of all variants available, but it is a third-party who is responsible for performing the actual binding. This combination is not very common or even likely.

Decide between internal and external binding depends on many factors, for example, the binding should be performed by the portlet developer or by the end-user. An important aspect to take into account here is that internal binding implies that the portlet must include the functionality to perform the binding by itself, which involves an increment in the complexity of the portlet.

3.2.3 Implementing Variability

The implementation of a variable feature requires a selection of a variability realization technique. Such a selection must be done basing on constraints established in the previous section 3.2.2. We must select the technique that involves a better balance between constraints.

Precisely, variability implementation is the main topic of the subsequent chapters where part of the contribution of this dissertation resides. The next three chapters study three different scenarios where

it is needed to introduce variability in portlets and where current portlet standards are not enough to provide the 'right' level of variability.

3.2.4 Managing the Variability

Inevitable business changes will make re-planning and directional shifts unavoidable. New actions, adjusted goals and metrics, and changes in process and organization will be needed to keep focused and maintain momentum [JGJ97]. Changes may involve adding new variants to variable features or removing obsolete variants, but they may also imply adding new variable features or removing them altogether. Management phase deals with these changes. As with identification of variability, management is outside the scope of this dissertation.

3.3 Conclusions

This chapter introduces the notion of "portlet families". It provides an overview of variability techniques available to cater for the differences between the distinct "members" of these families. Choosing an appropriated technique is not trivial. This chapter revised different approaches and distinguishing criteria. These criteria will be used in the rest of this thesis to account for the three scenarios outlined in the introduction, namely, provider-based variability (Chapter 4), consumer-based variability (Chapter 5) and user-based variability (Chapter 6).

Chapter 4

Provider-Based Variability: Realization of SOA using Portals[1]

4.1 Overview

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,

[1]Parts of this chapter have been previously presented [DTP07]

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, let's say, *Liferay* (liferay.com), can be deployed at an *eXo Platform* (exoplatform.com) portal, and vice versa. The *Web Services for Remote Portlets* (WSRP) specification brings this interoperability by providing a protocol that decouples portlet providers from portlet consumers (see chapter 2). This provides the infrastructure to make feasible a portlet market *à la COST*[2] 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*" [BKPS06]. Other similar initiatives include *JBoss Portlet Swap* (portletswap.jboss.org) and *Liferay Marketplace*.

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 "*as-is*" [JGJ97] 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, de- vice or user profiles (see [KPRS03] for an overview). Portlets are Web applications, so these aspects are applicable here. Additionally, and unlike "traditional" Web

[2]COST stands for Commercial off-the-shelf.

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., Liferay, eXo Platform, 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" behaviour.

Portlet development standards (e.g., JSR 286) 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 Platform as the portal IDE (Integrated Development Environment), and WSRP4Java[3] as the portlet provider.

The rest of the chapter is structured as follows. Section 4.2 and 4.3 motivate the issue by addressing the subject of variations and the time of variations with the help of an example. Section 4.4 outlines how to handle those variations using product-line techniques. The main contribution of the chapter rests on Section 4.5 that introduces a "portlet-line architecture" using WSRP. Finally, some conclusions end the chapter.

4.2 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 [DR05]. 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 advises to focus on some specific reuse contexts. An artefact is not universally variable, and making it variable on A can prevent the artefact from being variable on B. Since, it is most important to identify the distinct situations in which the portlet is most

[3] http://portals.apache.org/wsrp4j/

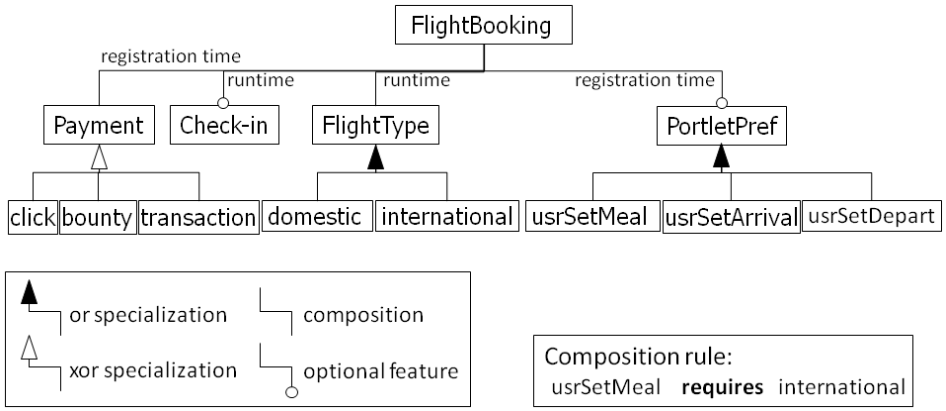


Figure 4.1: The Consumer Model.

likely to be reused. 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 van Gurp et al. in [GBS01].

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 4.1):

- *Payment,* which indicates how travel agencies are compensated by their cooperation. Alternatives include (i) *click fees*, where the carrier will pay the agency based on the number of users who access the portlet; (ii) *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 (iii) *transaction fees*, where the incomes of the ticket sales are split between the carrier and the

agency. These variants are alternatives.

- *Check-in*, which provides the namesake functionality. This variant is optional. Some travel agencies might offer online check-in and others might not.

- *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. *PortletPref* permits to tune which parameters are going to be set as portlet preferences (i.e., liable to be provided by end users). 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 favourite departure airport through the *EDIT* portlet mode. Other option is *usrSetArrival* that permits to provide a default for the arrival airport to end users.

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 [Bat05].

This feature model conforms **the Consumer Model**. This model acts as a catalogue 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.

4.3 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), see chapter 3. The following options are considered for the portlet case:

- *compilation time*, where the decision is taken when the portlet is being compile, adding the components required to supply the selected variant,
- *registration time*, in a WSRP scenario, when the relationship between portlet consumers and providers is established.
- *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* portlet mode). The terms "adaptive" and "adaptable" are used throughout the chapter to refer to these 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 [SvGB05] "*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 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 advises *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 4.1 is extended with annotations to reflect the binding strategy. In this way,

- *Payment* is set at registration time,
- *Check-in*, is resolved at runtime, e.g., by prompting the portal administrator of the travel agency through the *CONFIG*[4] portlet mode,
- *FlightTypes*, are also decided at runtime execution but, unlike *Check-in*, they are automatically resolved basing on the user profile (e.g., only users with the CEO profile can book for international flights), and
- *PortletPref* is decided at registration time to allow portal administrators to establish which portlet preferences are available to end users.

4.4 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 one hundred.

This advises to have distinct "versions" of the portlet at least for those features whose decisions can be resolved at compilation time (e.g.,

[4] A popular custom mode which is mainly used to read and modify the administrator level of preferences.

Payment and *PortletPref* in our sample case). Nevertheless, the number of combinations still goes up to twenty four 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 [JS00]. 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 [GEM04]. Therefore, the Web setting cannot 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 artefact, i.e., they cross cut distinct groups of artefacts, 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 [KLM+97].

Table 4.1 shows the "*feature x artefact*" matrix that highlights the distinct artefacts that are affected by the inclusion of a given feature. For our sample case, as for the artefact 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 behaviour. Adding feature *Check-in* 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. Moreover, "check-in" is made a read-only portlet preference, i.e., the administrator level of preference. This implies changes in *"portlet.xml"* as well as enhancing the views that support the *"CONFIG"* mode which now should permit the travel agency to enable

Feature	Artefact	controller	mVIEW		mEDIT		mCONFIG		portlet.xml
			model	view	model	view	model	view	
Base		X	X	X	X	X	X	X	X
Check-in		X	X	X			X	X	X
FlightType	domestic	X	X	X					
	international	X	X	X					
PortletPref	usrSetMeal	X	X	X	X	X			X
	usrSetArrival	X	X	X	X	X			X
	usrSetDepart	X	X	X	X	X			X
Payment	click		X						
	bounty		X						
	transaction		X						

Table 4.1: Feature scattering along distinct artefacts.

and disable the online check-in functionality. This is reflected in table 4.1 by marking the cells for the *controller*, the *mVIEW model*, the *mVIEW view*, the *mCONFIG model*, the *mCONFIG view* and the *portlet.xml*.

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. The "meal" portlet preference should be added to the *portlet.xml* and views that support the *"EDIT"* mode enhanced to enable the end users 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 artefacts realizing other features!

Therefore, handling variability implies engineering core artefacts 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 *Check-in* 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"* [CN01]. 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"* [ML97]. This includes the architecture, software components, design models and, in general, any artefact 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 [PBvdL05]. Details about using product-line techniques in a Web setting can be found at [BRPA05, CD03, DTA05, JBZZ03, RJ05, TBD07]. 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 an SPL architecture to portlet families. Implementation issues are not addressed here.

4.5 A Product-Line Architecture to Portlet Families

SPLs achieve systematic reuse for a set of applications sharing a "family flavour". 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 crosscutting 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.

4.5.1 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 chapter 2). These operations need now to account for the Consumer 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 4.2 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 [BTT05] for the description of feature models in XML. Basically, the snippet serializes in XML the model of figure 4.1. 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 4.3).

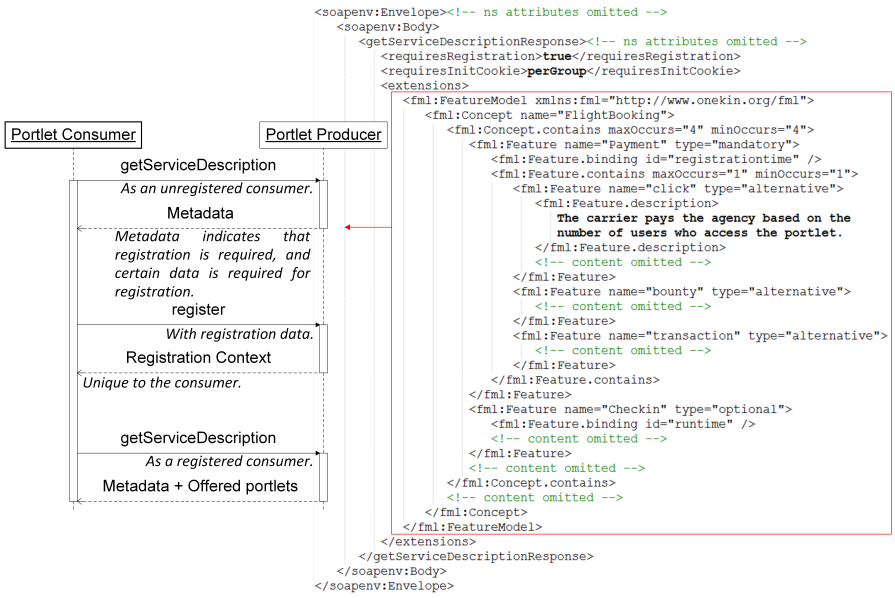


Figure 4.2: The *DomainProducer* communicates to the *PortalIDE* the Consumer Model.

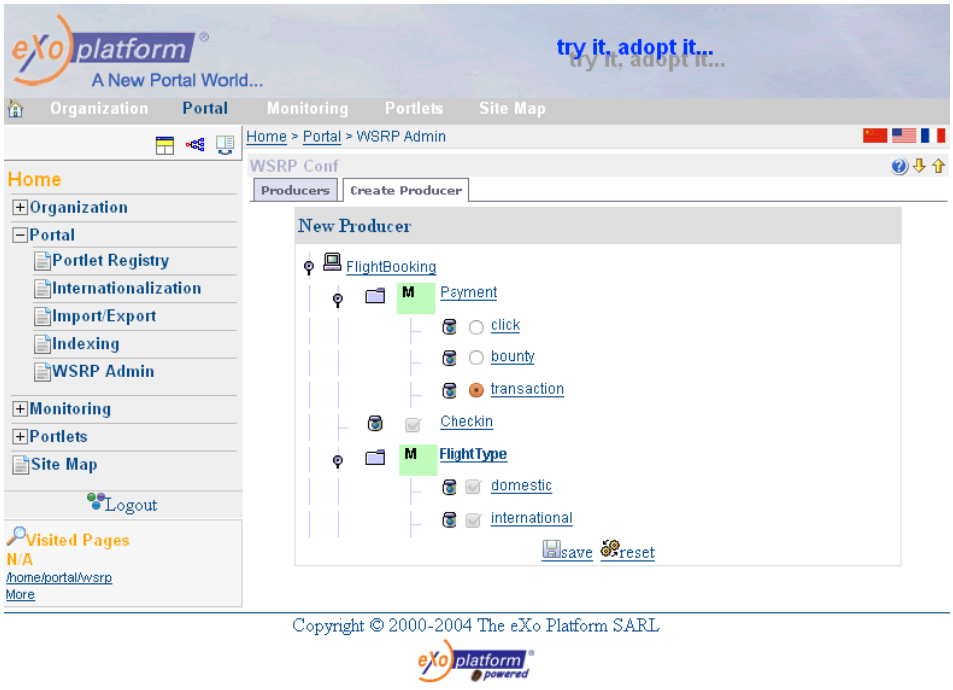


Figure 4.3: Conforming the *Consumer Profile* through the portal IDE.

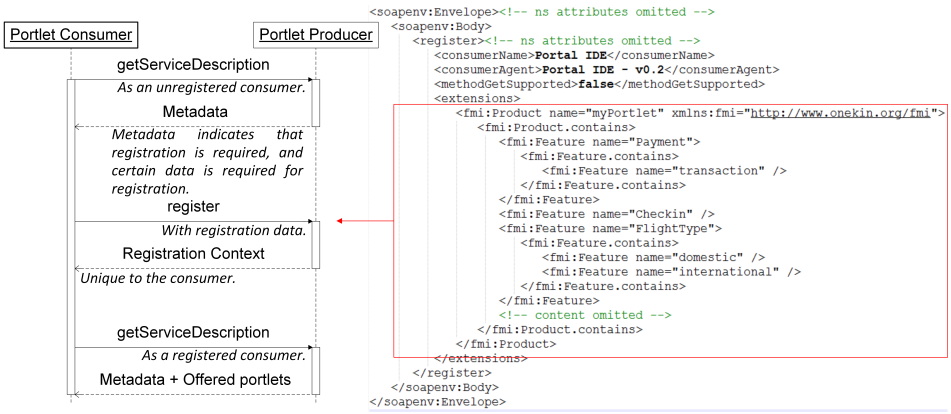


Figure 4.4: The *PortaIDE* communicates to the *DomainProducer* its Consumer Profile.

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 4.4 illustrates this situation for our sample case where the profile includes *transaction* as *Payment*, *domestic* and *international* will be available as *FlightType* and availability of *Check-in*.

4.5.2 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

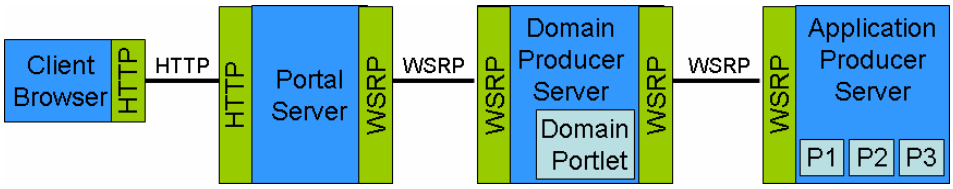


Figure 4.5: The architecture.

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 [Kru06]. 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 4.5). 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 chapter 2).

Portlet registration. A "family portlet" registration is achieved through the *domainProducer* (see figure 4.6). 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.

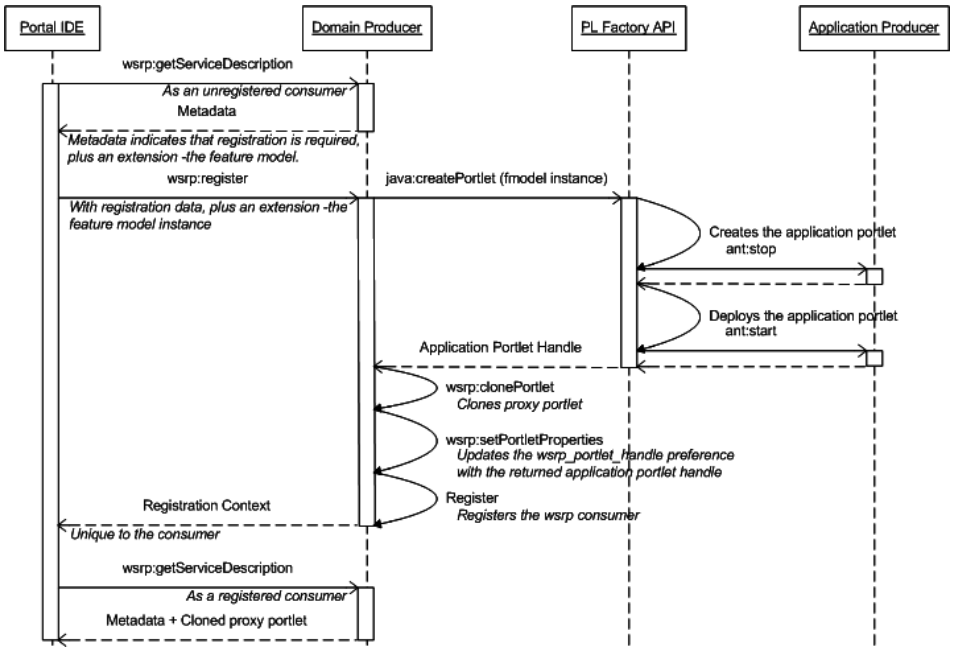


Figure 4.6: Registration time: sequence diagram.

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 4.6). 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 4.6), 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 raise 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.

4.6 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.

Parts of the work described in this chapter have been previously presented:

- Oscar Díaz, Salvador Trujillo and Sandy Pérez. Turning Portlets into Services: The Consumer Profile. In proceedings of the 16th International World Wide Web Conference (WWW2007), Banff, Alberta, Canada, 2007.

Chapter 5

Consumer-Based Variability: Tagging as a Portal Commodity[1]

5.1 Overview

Enterprise Information Portals play a three-fold role. As a means by which to manage and access content, portals play the role of content managers. As the mechanism to integrate third party applications using portlets or gadgets, portals can be regarded as front-end integrators. Finally, portals also offer a conduit for on-line communities. It is in this third role where the importance of incorporating social networking facilities in current portal engines emerges. Among social networking activities, this chapter focuses on social tagging.

Traditional tagging sites such as *Delicious* (delicious.com), *Tumblr* (tumblr.com) or *Flickr* (flickr.com) can be characterized as being *self-sufficient* and *self-centred*. The former implies that all it is need for tagging (i.e., the description of the resource, the tag and the user) is kept within

[1]Parts of this chapter have been previously presented [DPA09]

the tagging site. *Delicious* keeps the bookmark URL, the tags and the user community as assets of the site. On the other hand, self-centeredness indicates that all *Delicious* care about is its own resources, tags and users. No links exists with other tagging sites, even if they tag the same resources, e.g., *CiteULike* (citeulike.org).

This situation changes when moving to a portal setting. A hallmark of portals is integration. Rather than providing its own services, a portal is also a conduit for external applications. So offered applications are technically known as portlets. Portlets can be locally deployed or be provided remotely through third-party providers. For instance, a portal can offer the possibility of blogging, purchasing a book, or arranging a trip, all without leaving the portal. Some of these portlets can be built in house whereas others can be externally provided by third parties, e.g., *Amazon* (amazon.com) or *Expedia* (expedia.com). The portal mission is to offer a common gateway that hides such distinct origins from the user. This has important implications on the way tagging can be incorporated into portals, namely:

- portals are not self-sufficient. Taggers (i.e., the portal community) and tags are portal assets. However, and unlike self-sufficient tagging sites, portals could not hold the description of all tag-able resources. For instance, the description of the books or hotels offered through the portal could be remotely kept by, e.g., *Amazon* and *Expedia*, respectively. This outsource of content description does not imply that the external resources are not worth tagging. This leads to distinguish between two actors: *the resource provider*, which keeps the content of the tag-able resources (e.g., book description in the case of *Amazon*), and *the resource consumer* (i.e., the portal), which holds the tagger community and the tags,

- portals are not self-centred. Traditional tagging sites are "tagging islands": each site keeps its own tagging data. Providing an integrated view of these heterogeneous tagging silos is at the user

> expenses. By contrast, portals strive to glue together heterogeneous applications. This effort implies offering a consistent set of tags no matter neither the resource nor the portlet through which the tagging is achieved. That is, our expectation is that employees would use a similar set of tags no matter the portlet that holds the tagged resource.

Based on these observations, consistency is identified as a main requirement, i.e., tagging should be seamlessly achieved across the portal, regardless of the type (messages, books, hotels, etc.), or origin (i.e., *Amazon*, *Expedia*, etc.) of the resource. This consistency is two-fold. *"Back-end consistency*" implies the use of a common structure for tagging data, e.g., a common set of tags. On the other hand, *"front-end consistency*" entails tagging interactions to be achieved seamlessly and cohesively across the portal using similar rendering recourses and aesthetic guidelines.

To support back-end and front-end consistency, currently, most portal vendors offer tagging *as a portal functionality.* The portal is regarded as a content manager. The portal owns the resources, and provides functionality for tagging. Tagging is restricted to those resources within the realm of the portal. Being content managers, portals can keep their own resources. Additionally, portals are also integration platforms, making external resources available through portlets. The remote origin of these resources does not imply that they are not worth tagging.

As integration platforms, portals offer commodities for easing the integration of heterogeneous applications (e.g., the *Single Sign-On*[2] commodity is a popular example). Likewise, we advocate for tagging services to be offered *as a portal commodity*. This is, tagging services are up to the portal but offered through the companion portlets. This implies that portlets should be engineered to be plugged into this commodity

[2]This commodity enables a user to log in once at the portal, and gain access to the available applications being offered through the portal without being prompted to log in again.

rather than building their own tagging functionality. In the same way, that portlets adapt their rendering to the aesthetic guidelines of the hosting portal, tagging through portlets should also cater for the peculiarities of the consumer portal.

To this end, this work presents a novel architecture to orthogonally support tagging as a crosscut on top of portlets, i.e., as a portal commodity. *RDFa* annotations are used as a mean for the *resource provider* to communicate the *resource consumer* the existence of tag-able resources. On the other hand, the standard portlet event mechanism allows the *resource consumer* to broadcast tag-based queries to *resource providers*. This has been implemented for the *Liferay* (liferay.com) portal engine using *TASTY*[3] as the tagging engine. This implementation evidences the feasibility of the approach, which make up the contribution of this chapter.

The rest of the chapter is structured as follows. Section 5.2 and 5.3 addresses the subject of variations and the time of variations with the help of an example. Section 5.4 outlines how to handle those variations can be handle using markup replacement techniques inspired by linkers in the C programing language. The main contribution of the chapter rests on Section 5.5 that introduces how actual consistency is realized. Finally, some conclusions end the chapter.

[3] *https://code.google.com/p/microapps/wiki/Tasty*

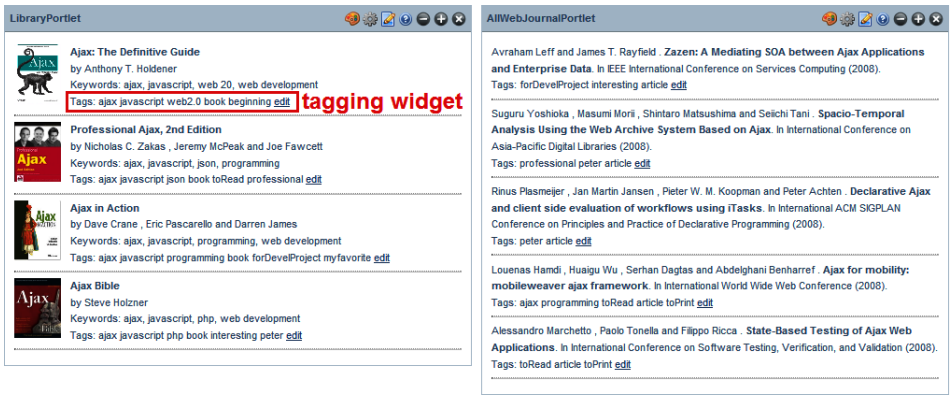


Figure 5.1: A portal page offering two portlets (i.e., *LibraryPortlet* and *AllWebJournalPortlet*).

5.2 What Can Vary

Portlet development standards account for front-end consistency by defining a set of Cascading Stylesheets (CSS) styles that portlets should use in rendering their markup in order to achieve a common and pluggable look and feel. However, this only establishes how elements must be rendered on screen (e.g., fonts, colours, spacing), but not how to interact with them. For example, one portal can support tag addition via in-place editable lists whereas a different one may opt for prompting the user using pop-up forms, and both of the them would be portlet's standards-compliant. From the portlet perspective, front-end consistency means that the same portlet must use in-place editable tag lists for tag addition when being delivered through the former portal, and pop-up forms when being delivered through the latter one. Thus, to achieve front-end consistency, the portlet's presentation layer must have the capability of being varied to accommodate the different tagging interaction patterns found in portals.

As for back-end consistency, figure 5.1 provides a snapshot of a portal page offering two portlets: *LibraryPortlet* and *AllWebJournalPortlet* that render content on books and publications, respectively. Both books and publications are kept outside the portal realm.

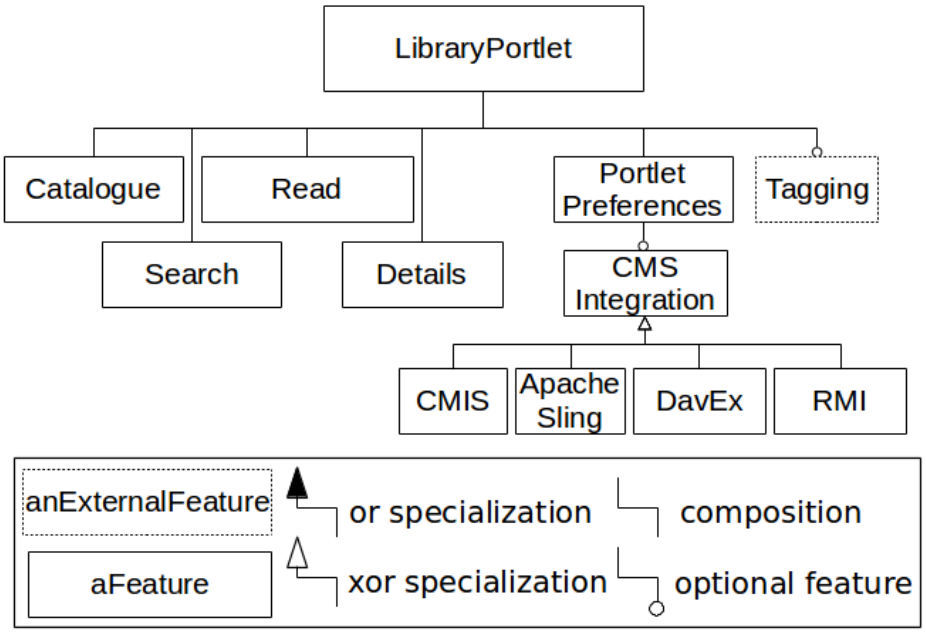


Figure 5.2: A feature diagram where tagging is represented as an external feuture.

Notice that both portlets, regardless of their different providers, offer the same set of tags. That is, tagging adapts to the hosting portal. This also implies that if the very same portlet is offered through a different portal then, the rendered tag set will be distinct since tags reflect the projects, roles and ways of working of the organization at hand.

Therefore, both front-end and back-end consistency requires variations in the presentation layer of the portlet. On the other hand, since tagging functionality is up to portal, no variation is required neither in the functional layer of the portlet nor in the data layer.

Tagging as a portal commodity is a clear example of an *external feature* introduced in section 3. It is a functionality offered by the portal but used by the portlets that depend on it. As a feature, it can be included as part of a *Consumer Model* (see section 4). External features are motivated by the need to map requirements that can be met by using functionality external to the system to features [ZJ97]. Figure 5.2 shows a possible Consumer

Model for the *LibraryPortlet* presented in figure 5.1.

However, depending on certain external feature limits the amount of deployment platform. To avoid this, in figure 5.2 tagging is modelled as an optional feature. Organizing external features under variant/optional features may help improve platform independence.

5.3 When Can It Vary

The binding options available for the portlet case are:

- *compilation time*, where the decision is taken when the portlet is being compile, adding the components required to supply the selected variant,
- *registration time*, in a WSRP scenario, when the relationship between portlet consumers and providers is established.
- *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* portlet mode). The terms "adaptive" and "adaptable" are used to refer to these two kinds of runtime binding.

Binding at compilation time only works for locally deployed portlets. Remotely provided portlets are already running on a remote container and communicate with the portal through WSRP (see chapter 2 for an overview).

On the other hand, binding at registration time, like in chapter 4, means that we are generating an organization-specific portlet for every organization registered against our portlet provider. In the previous chapter it makes sense because we were dealing with portlet provided functionality. However, tagging as a portal commodity is an external functionality provided by the portal surfacing the portlet. That is, variations could depend on the used portal platform but never on the organization at hand.

Therefore, the binding option that better fit tagging supported as a portal commodity is at runtime.

5.4 How Is It Supported

At this point, two different approaches are feasible. One option is that portlets are equipped with binding functionality and are able, at runtime, to select the right tagging variant for the portal at hand. Although this is a feasible option, the solution would not be very scalable. New portal platforms may involve changes in the binding functionality to account for the new variants.

A different solution requires binding functionality to be residing at the portal side. This solution is inspired by linkers in C programing language. In C, the compilers translate pre-processed code into assembly code. The assembly code generated by the compilation step is then passed to the assembler which translates it into machine code; the resulting file is called an object file. Since an object file will be linked with other object files and libraries to produce a program, the assembler cannot assign absolute memory locations to all the instructions and data in a file. Rather, it writes some *notes* in the object file about how it assumed things were layed out. It is the job of the linker to use these notes to assign absolute memory locations to everything and resolve any unresolved references. In the portlet case, portlet developers play the role of "assemblers" and introduce *notes* into the portlet markup in order to communicate the presence of tag-able resources. The role of linker is played by the portal, which use these notes to seamlessly weave the markup corresponding to the tagging functionality into the portlet markup.

Notice that this solution does not jeopardize portlet interoperability, in the worst case, we won't be able to add tags to the resources being rendered through the portlet but the portlet functionality will work as expected. Next section delves into the details.

5.5 Social Tagging as a Portal Commodity

This scenario introduces three actors, namely: **portlets**, which provide the tag-able resources; **the portal**, which embodies the portal users as potential taggers; and **the tagging commodity**, i.e., a portal component that provides tagging utilities. This chapter looks at two such tagging functionalities: tag assignment and tag-based querying.

However, the existence of three different actors should not jeopardize one of the portal hallmarks: interaction consistency across the portal. Tagging should be homogenously achieved throughout the portal, no matter where the resource resides (i.e., which portlet renders it). Additionally, and on top of the portal mandate, the desire for tag consistency emerged as a major request among portal users, (e.g., "how will others find my content if I don't use the same tags over and over?") as drawn from a recent study [TSMM08].

This consistency is two-fold. "Back-end consistency" implies the use of a common structure for tagging data, e.g., a common set of tags. On the other hand, "front-end consistency" entails tagging interactions to be achieved seamlessly and cohesively across the portal using similar rendering guidelines. Next sections delve into the details.

5.5.1 *Back-end* Consistency

Back-end consistency implies tagging data to be a portal asset rather than being disseminated across different silos. Tagging data basically refers to tags, taggers and tag-able resources. Both, taggers and tags, are naturally held by the portal. However, tag-able resources can be outside the portal realm. Although tagging could become a portal duty, some tag-able resources would still be provided by third-party portlets. Therefore, a mechanism is needed for portlets to make the portal aware of their tag-able resources.

The main means for portlet-to-portal communication is the markup

fragment that the portlet delivers to the portal. Here, the portal is a mere conduit for portlet markups. Portals limit themselves to provide a common skin and appropriate decorators to portlet fragments, being unaware of what this markup conveys. We propose to annotate this markup with tagging concerns using *RDFa* [W3C08a].

RDFa is a W3C standard that provides syntax for communicating structured data through annotating the XHTML content. In our case, *RDFa* offers a means for the portlet provider to communicate the portlet consumer the existence of tag-able resources. The idea is to capitalize on the fragment layout to annotate tag-able resources.

Firstly, an ontology is defined which is later used to annotate the fragment. This ontology should serve to indicate both *what* to tag and *where* to tag (see later). This is the aim of *PartOnt* (*Part*icipatory *Ont*ology), an ontology that aims at capturing manners in which users engage in the participatory web. One of these ways is of course, tagging. Rather than defining its own concepts, *PartOnt* capitalizes on existing ontologies that already formalize some of these notions. Specifically, *PartOnt* benefits from *TagOnt*, an ontology that captures tagging by describing resources, tags, taggers, tagging time, and so on [Kne]. Figure 5.3 shows these ontologies, both the RDF code and the *Protégé* rendering counterpart.

These ontologies are then used to annotate the portlet markup. An example is shown in Figure 5.4. The JSP script outputs a *LibraryPortlet* fragment. Book data (i.e., title, authors, etc.) are rendered as table rows (*TR*), where book keywords are iteratively enclosed within SPAN elements. All of the table cells are wrapped within a table (<*table*>) which in turns is wrapped in another table together with the book-cover image.

This markup is then annotated along the previous ontologies. Specifically, the following structural HTML elements are annotated[4]:

[4]As far as this work is concerned, we ignore the resource content (i.e., we do not annotate e.g., titles or authors of book resources). All the portal needs to know is that a tag-able resource is being rendered. The details about the rendering itself are left to the portlet.

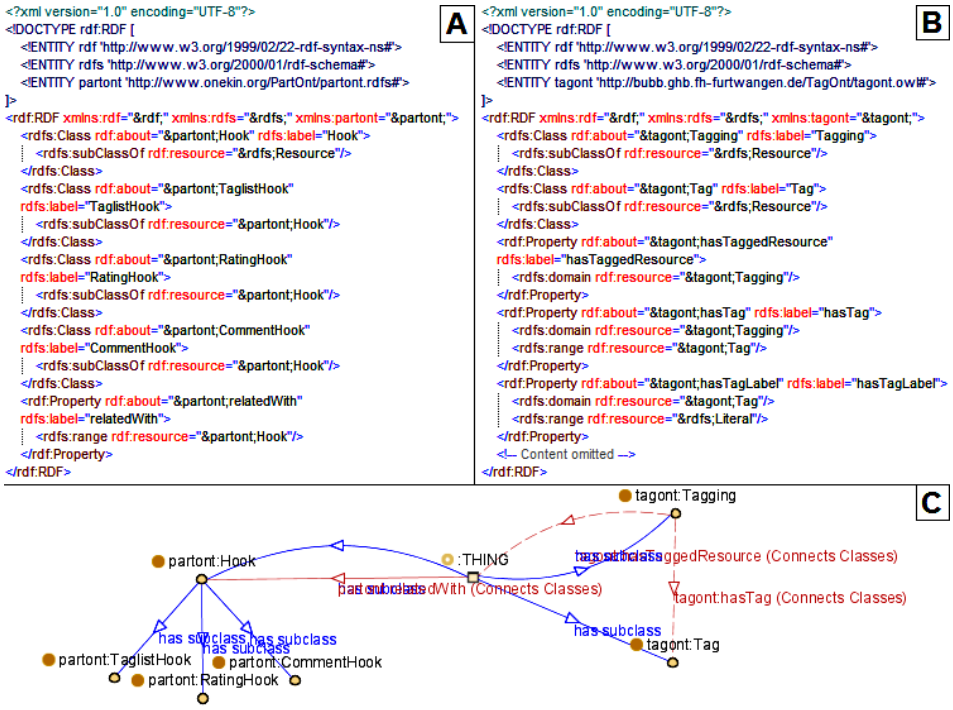


Figure 5.3: The *PartOnt* (a) and the *TagOnt* (b) ontologies together with their *Protégé* rendering counterparts (c).

- HTML element that embodies a tag-able resource. In our example, this corresponds to the outer <*table*> element. This element now includes an *"about"* attribute which denotes the existence of a tag-able resource. The identifiers of tag-able resources are supported as *Uniform Resource Identifiers* (URIs). Following Semantic Web practices, these URIs are created by concatenating a namespace with a resource's key,

- HTML element that conveys potential tags. In this case, we identify keywords as playing such role. This implies to annotate the <*span*> element with the *"tagont:Tag*" annotation. These tags are provided for the portal's convenience, and they are expected to describe the resource content. It is up to the portal to incorporate these tags as suggestions during tagging. These portlet-provided tags should not

```
<jsp:useBean id="library" scope="request" class="java.util.ArrayList" />
<div   xmlns:books="http://www.onekin.org/library/"
       xmlns:partont="http://www.onekin.org/PartOnt/partont.rdfs#"
       xmlns:tagont="http://bubb.ghb.fh-furtwangen.de/TagOnt/tagont.owl#">

       <c:forEach var="book" items="${library}">
              <table about="[books:${book.id}]"><tr>
                     <td><!-- BOOK'S PAPERBACK IMAGE --></td>
                     <td>
                            <table>
                                   <tr><td><!-- BOOK'S TITLE --></td></tr>
                                   <tr><td><!-- BOOK'S AUTHORS --></td></tr>
                                   <tr><td typeof="tagont:Tagging">
                                          <div rel="tagont:hasTaggedResource" resource="[books:${book.id}]"/>
                                          Keywords:
                                          <span rel="tagont:hasTag">
                                                 <c:forEach var="keyword" items="${book.keywords}">
                                                        <span typeof="tagont:Tag" property="tagont:hasTagLabel">
                                                               <c:out value="${keyword}"/>
                                                        </span>
                                                 </c:forEach>
                                          </span>
                                   </td></tr>
                                   <tr><td>
                                          <div style="display:none;" rel="partont:relatedWith"
                                                                     typeof="partont:TaglistHook" />
                                   </td></tr>
                            </table>
                     </td>
              </tr></table>
       </c:forEach>
</div>
```

Figure 5.4: JSP that delivers a fragment markup with annotations along the *TagOnt* and *PartOnt* ontologies.

be mistaken with those provided by the portal users.

These annotations permit the portlet consumer (i.e., the portal) to become aware of resources and tags coming from external sources. This external data is incorporated into the portal not when it is rendered but when it is tagged. When the first tag is added, the portal check if the resource ID is already in the tagging repository (see later Figure 5.5).

However, resource IDs and tags are not introduced in the tagging repository right away. Rather, the tagging commodity should include a *"cleaning module*" to ascertain whether two tags/resources really stand for the same notion. For instance, the same resource can be offered as a book in *LibraryPortlet* and as a publication in *AllWebJournalPortlet*. Likewise, this resource can be tagged as *"ServiceOrientedArchitecture*" in one place and *"SOA*" in the other. This cleaning module will provide some heuristics to ascertain the equality of resources and tags being offered in different forms by different resource providers. This effort is akin to the

view of the portal as an *integration* platform, and an argument in favour of tagging being conducted through the portal rather than as a disperse activity performed at each resource provider.

5.5.2 *Front-end* Consistency

Portals are a front-end technology. Much of their added value (and investment) rests on how content is rendered and navigated. In this context, presentation consistency is a must to hide the diverse sources that feed the portal. Tagging wise, consistency mandates tagging interactions to be seamlessly and coherently achieved across the portal. This would not be much of a problem if tagging were only up to the portal. But, this chapter highlights that portal tagging is a joint endeavour among the portal and the companion portlets. Rendering wise, this coupling can be achieved at the portlet place (through markup portions referred to as *widgets*) or at the portal place (using a publish/subscribe approach with local portlets). Next subsections address each of these approaches.

Front-end Consistency through Widgets

Seamlessness calls for tagging to be conducted at the place tag-able resources are rendered (side-by-side rendering). This place is the portlet fragment. But portlets should not deliver their own tagging functionality since a premise of this work is that such functionality should be provided by the portal. But, portals are traditionally mere proxies for the portlet markup. Tagging however, requires portals to take a more active role. Besides skins and decorators, portals now become the purveyors of tagging widgets to be injected into the portlet markup.

The question is how can the portal know where to inject these widgets? Annotations are again used for this purpose. Specifically, the *PartOnt* ontology includes a *Hook* class, with a subclass *TaglistHook* that denotes an extension point for adding markup to update the tag list. This class annotates the HTML element that plays the “hook” role. Figure 5.4 shows

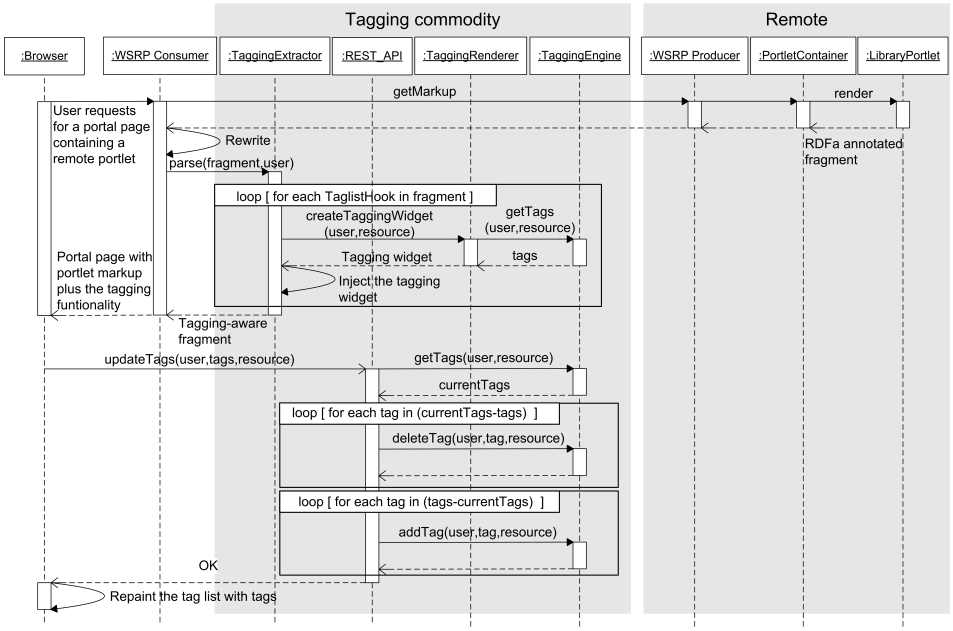


Figure 5.5: Interaction diagram: *base requests* vs. *tagging requests*.

our sample fragment where this role is played by a <*div*> element. At execution time, the portal locates the "hooks" and injects the tagging widget (see later).

Markup coming from the portlet should be seamlessly mixed together with markup coming from the portal so that the user is unaware of the different origins. After all, this is the rationale behind letting the portlet specify the tagging hooks: injecting the extra markup in those places already foreseen by the portlet designer so that the final rendering looks harmonious. However, the distinct markup origins become apparent to the portal which needs to propagate the user interactions to the appropriate target. Specifically, *base requests* (i.e., those with the portlet markup) are propagated to the portlet provider, while *tagging requests* (i.e., those with the tagging widget) are processed by the tagging commodity.

Figure 5.5 provides an overview of the whole process where these two types of interactions are distinguished:

1. *base request*. According with the WSRP standard, user interactions

with portlet markup are propagated till reaching the appropriate portlet. In return, the portlet delivers a markup, now annotated with tagging metadata,

2. content annotation processing. At the portal place, the tagging commodity (specifically an *RDFa* parser) extracts both tag-able resources and tags conveyed by the actual markup. This data is kept at the tagging repository.

3. *hook* annotation processing. If the markup also holds *"TaglistHook"* annotations, the tagging commodity (specifically, a markup renderer) outputs the appropriate widget to be injected at the hook place. The markup renderer can need to access the tagging repository, e.g., to recover the tags currently associated with a given resource.

4. markup rendering. The original markup has now become a tagging-aware fragment, i.e., a fragment through which tagging can be conducted,

5. *tagging request*. Now, the user interacts with the tagging markup (e.g., requesting the update of the tag set). This petition is directed to the tagging commodity which checks the additions and removals being made to the tag set kept in the repository. In return, the tagging commodity repaints the tagging markup.

As the previous example illustrates, the co-existence of markups from different origins within the same portlet decorator brings an Ajax-like style to markup production. In Figure 5.5, lines with solid triangular arrowheads denote synchronous communication whereas open arrowheads stand for asynchronous communication. Specifically, the tagging request is asynchronously processed.

Front-end Consistency through Local Portlets

Previous subsection illustrates the case of a tagging functionality (e.g., tag update) to be achieved at the portlet place. However, other services can be directly provided by the portal but in cooperation with the companion portlets. Tag-based querying is a case in point.

Comprehensive querying implies the query to expand across resources, no matter their origin. A query for resources being tagged as *"forDevelProject"* should deliver books (hence, provided by the *LibraryPortlet* portlet), publications (hence, supplied by the *AllWebJournalPortlet* portlet), post blogs (locally provided), etc. being tagged as used in this project. Such a query can be directly answered through the tagging repository that will return the set of resource identifiers meeting the query condition.

However, portals are a front-end technology. Providing a list of identifiers is not a sensible option when an end user is the addressee. Rather, it is the content of resource what the user wants to see. We need then to *de-reference* these identifiers. Unfortunately, the tagging repository cannot "de-reference" those identifiers. The portal owns the tagging data. But it is outside the portal realm to know the resource content as well as how this content is to be rendered. This is the duty of the resource providers, i.e., the portlets. Therefore, the portal cannot accomplish the whole query processing on its own since this also involves content rendering.

Figure 5.6 illustrates this situation. First, a mean is needed for the user to express the query. For the sake of this work, a simple portlet has been built: *TagBarPortlet*. This portlet consults the tagging repository, renders the tags available, and permits the users to select one of these tags. The selection has two consequences. First, the selected tag is highlighted. Second, and more important, the companion portlets synchronize their views with this selection, rendering those resources that were tagged with the selected tag *at this portal*. This last point is important. The very

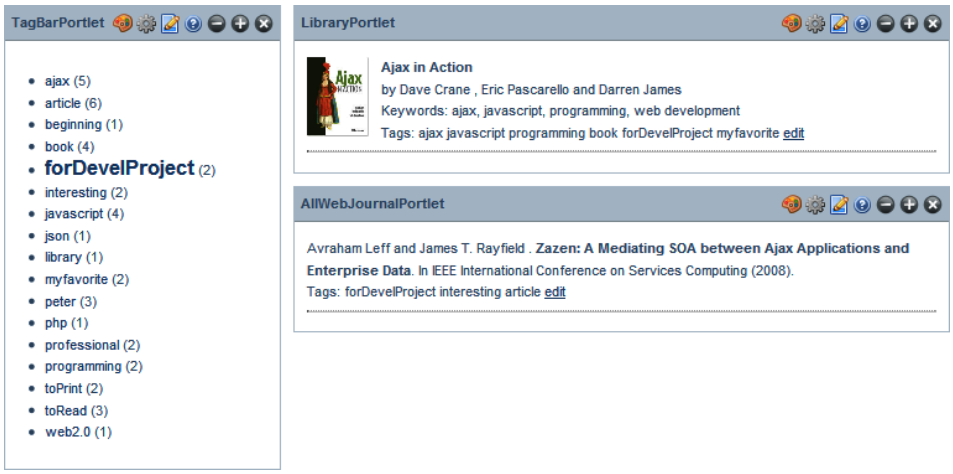
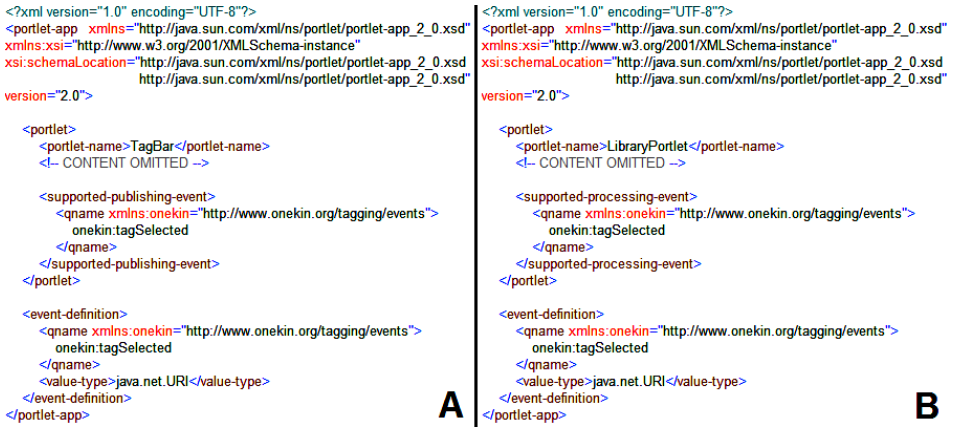


Figure 5.6: Split query processing. Query specification goes through *TagBarPortlet*: the tag selected by the user is highlighted. Query outcome is delegated to the portlets holding the resource content, i.e., *LibraryPortlet* and *AllWebJournalPortlet*.

same portlet can be offered through different portals. Hence, the same resource (e.g., a book) can be tagged at different places (i.e., through distinct portals). When synchronized with the *TagBarPortlet* of portal P1, the portlet just delivers those resources being tagged through portal P1.

This scenario again requires a means for portal-to-portlet communication. Previous section relies on the rendering markup as the means of communication. This was possible because the data flew from the portlet to the portal. However, now identifiers/tags go the other way around: from the portal to the portlets. To this end, we follow a publish/subscribe approach where data flows from the publisher (i.e., the portal, better said, the portal representative, i.e., *TagBarPortlet*) to the subscriber (e.g., *LibraryPortlet* and *AllWebJournalPortlet*). The availability of an event mechanism in the Java™Portlet Specification [Jav08] comes to our advantage.

Portlet events are intended to allow portlets to react to actions or state changes not directly related to an interaction of the user with the portlet. Portlets can be both event producers and event consumers. Back to our




```
<?xml version="1.0" encoding="UTF-8"?>
<portlet-app  xmlns="http://java.sun.com/xml/ns/portlet/portlet-app_2_0.xsd"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://java.sun.com/xml/ns/portlet/portlet-app_2_0.xsd
                    http://java.sun.com/xml/ns/portlet/portlet-app_2_0.xsd"
version="2.0">

   <portlet>
      <portlet-name>TagBar</portlet-name>
      <!-- CONTENT OMITTED -->

      <supported-publishing-event>
         <qname xmlns:onekin="http://www.onekin.org/tagging/events">
            onekin:tagSelected
         </qname>
      </supported-publishing-event>
   </portlet>

   <event-definition>
      <qname xmlns:onekin="http://www.onekin.org/tagging/events">
         onekin:tagSelected
      </qname>
      <value-type>java.net.URI</value-type>
   </event-definition>
</portlet-app>
```

```
<?xml version="1.0" encoding="UTF-8"?>
<portlet-app  xmlns="http://java.sun.com/xml/ns/portlet/portlet-app_2_0.xsd"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://java.sun.com/xml/ns/portlet/portlet-app_2_0.xsd
                    http://java.sun.com/xml/ns/portlet/portlet-app_2_0.xsd"
version="2.0">

   <portlet>
      <portlet-name>LibraryPortlet</portlet-name>
      <!-- CONTENT OMITTED -->

      <supported-processing-event>
         <qname xmlns:onekin="http://www.onekin.org/tagging/events">
            onekin:tagSelected
         </qname>
      </supported-processing-event>
   </portlet>

   <event-definition>
      <qname xmlns:onekin="http://www.onekin.org/tagging/events">
         onekin:tagSelected
      </qname>
      <value-type>java.net.URI</value-type>
   </event-definition>
</portlet-app>
```

Figure 5.7: *portlet.xml* configuration files for *TagBarPortlet* and *LibraryPortlet*. Both portlets know about the *tagSelected* event.

sample case, the query-specification portlet, i.e., *TagBarPortlet*, fires the appropriate event that is broadcasted by the portal to the resource-provider portlets to make them aware of the tag being selected. Publications and subscriptions are parts of the portlet definition and hence, expressed in the configuration file *portlet.xml*. Figure 5.7 shows those files for *TagBarPortlet* and *LibraryPortlet*. The former defines a published event, *tagSelected*, whereas *LibraryPortlet* acknowledges the capacity to process *tagSelected* events.

Processing *tagSelected* occurrences imply rendering the content of the so-tagged resources at the portlet place. For instance, *LibraryPortlet* should produce markup for those books being tagged with the tag provided in the event payload. However, *LibraryPortlet* holds the resource content but ignores how they have been tagged. This tagging data is kept at the portal. Therefore, the portlet needs to get such data from the tagging commodity. As a result, the tagging-commodity URL is included as part of the event payload, so that the portlet can construct a REST petition asking which of *its* resources are so-tagged at *this* portal. Therefore, the very same portlet can process *tagSelected* occurrences coming from different portals

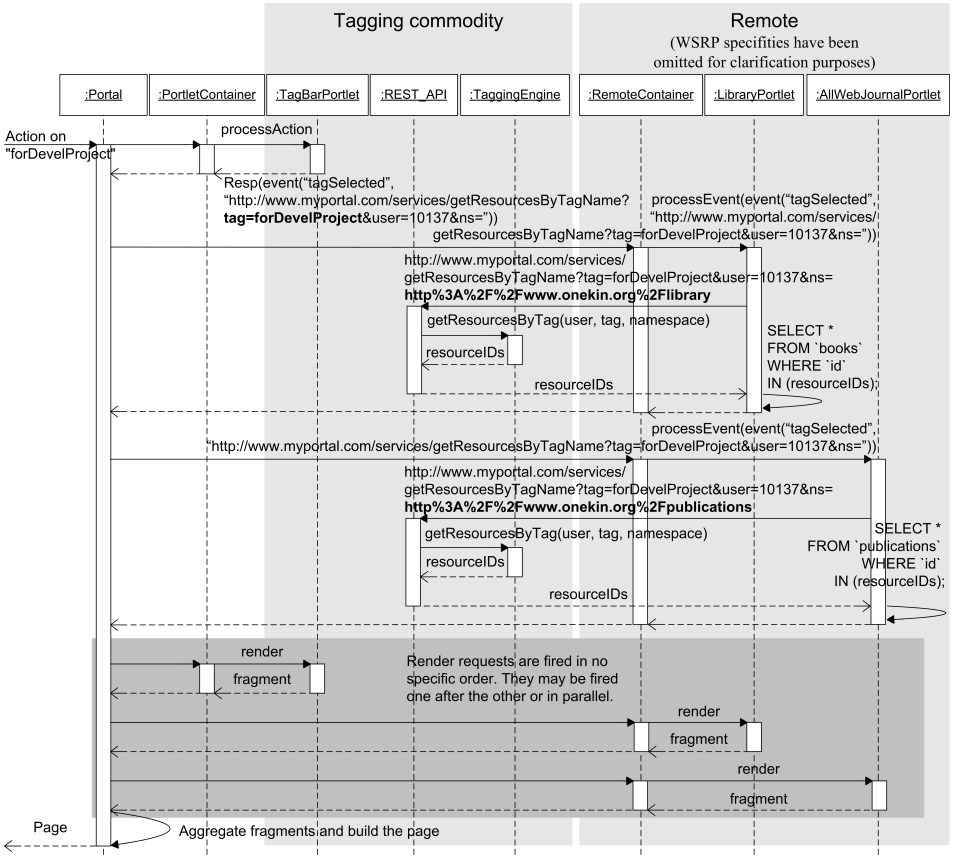


Figure 5.8: Handling a *tagSelected* occurrence.

and hence, whose payloads refer to different URLs[5]. In this way, portlet interoperability is preserved.

Figure 5.8 provides the global view. First, the user selects *"forDevelProject"* as the tag to be used as the filtering criteria. This request is handled by *TagBarPortlet* that signals a *tagSelected* occurrence. The portal forwards this occurrence to their subscribers: *LibraryPortlet* and *AllWebJournalPortlet*. Processing *tagSelected* involves first, to query

[5]An alternative design would have been for *TagBarPortlet* to recover itself all resource identifiers that exhibit the selected tag, and include the whole set of identifiers as part of the event payload. On reception, the portlet filters out its own resources. However, this solution does not scale up for large resource sets. Additionally, the option of restricting the payload to just those resources of the addressee portlet forces to have a dedicated event for each portlet.

the *TaggingEngine* about the so-tagged resources. To this end, the *REST_API* provides the *getResourceByTag* method. This method outputs a list of resource identifiers for which the *LibraryPortlet* should locally retrieve the content and produce the markup. This process is accomplished for all the resource-provider portlets. This ends the state changing logic phase of the portlet life cycle.

The rendering phase builds up the portal page out of the portlet fragments. This implies sending the *render()* request to each of the portlets of the page, assembling the distinct markups obtained in return, and render the result back to the user. For our sample case, the outcome is depicted in Figure 5.6.

5.6 Conclusions

This work argues for portal tagging to be a joint endeavour between resource providers (i.e., portlets) and resource consumers (i.e., portals). Additionally, portlets are reckoned to be interoperable, i.e., deliverable through different portals. These observations advocate for tagging to be orthogonally supported as a crosscut on top of portlets, i.e., a portal commodity.

A tagging commodity has been implemented for the *Liferay* portal engine using TASTY as the tagging engine[6]. This implementation evidences the feasibility of the approach. The benefits include:

- portal ownership of tagging data,
- increases consistency in the set of tags used to annotate resources, regardless of the resource owner,
- facilitates consistency among tagging activities, no matter the portal application through which tagging is achieved,

[6]http://microapps.sourceforge.net/tasty/

- permits tagging to be customized based on the user profile kept by the portal. For instance, the suggested set of tags can be based on the user profile, the projects he participates in, etc.

As in other situations where applications need to cooperate, the main challenge rests on agreeing in common terms and protocols. In our case, this mainly implies the standardization of the tagging ontology, and the REST API.

Parts of the work described in this chapter have been previously presented:

- Oscar Díaz, Sandy Pérez and Cristóbal Arellano. Tagging-Aware Portlets. In proceedings of the 9th International Conference on Web Engineering (ICWE2009), San Sebastian, Spain, 2009.

Chapter 6

User-Based Variability: Mashup-based Personalization [1]

6.1 Overview

A mashup is a *lightweight* web application created by *opportunistically* combining information or capabilities from more than one existing source, normally in a *do-it-yourself (DIY)* manner. On the other hand, Enterprise Information Portals (hereafter just portals) are *heavyweight*, carefully *pre-planned* applications that offer *corporations* a means by which to manage and access both content and applications from disparate sources across the firm. Integrating, adapting and sharing are all hallmarks of both portals and mashups. Differences partially stem from the complexity of the integration, the criticality of the services, the sensitivity of the data, and non-functional requirements (e.g., availability, efficiency, etc.) that impose a centralized and professionalized portal administration. From this perspective, both portals and mashups tackle data/service integration but provide different answers to the balance between easiness and reliability. This chapter addresses mashups on portals.

Traditional mashing-up distinguishes two scenarios w.r.t. source

[1] Parts of this chapter have been previously presented [PD10, DPnP07]

applications. In the first scenario, the mashup is a separate application from source applications (e.g., *Yahoo! Pipes*). In the second scenario, the mashup is an enhancement on the source application (e.g., *MashMaker* [EG07], *MARGMASH* [DPnP07]). This normally requires users to install a browser plug-in for the mashup to be woven with the application markup at the client. In both cases, source applications ignore they are being subject to mashup. Portals are Web applications. Hence, previous scenarios can be applied to portals. However, this prevents mashing-up from capitalizing on portal utilities (e.g., single-sign on, access control, customization, etc.). Unlike other Web applications, portals reckon to provide an integration space for corporate services. By mashing-up at the back of the portal, you miss the opportunity to benefit from this integration space. Therefore, we tackle "mashup-aware portals".

This departs from traditional scenarios. First, and unlike the *Yahoo! Pipes* approach, the mashup is offered without leaving the portal. Second, and unlike *MashMaker*-like approaches, now the portal takes an active role on facilitating mashups on portal services. This implies that (1) no additional plug-in is necessary since mashup weaving is already engineered into the portal, (2) the portal "guides" users throughout the mashup process, and (3), the portal provides the context for mashups to be seamlessly integrated into portal services. From the portal perspective, mashing-up becomes an additional approach to customize portal offerings.

Customization helps portal services (e.g., booking flight tickets) to be adapted to the users' roles. Both content and services can be adapted to the current user (e.g., *flightBooking* is only available for senior engineers). Personalization goes one step further by permitting users themselves to set some configuration options (e.g., the *destinationAirport* parameter is set to *"New York"* by *John Douglas*). This work introduces mashups as an additional personalization mechanism. For instance, *John Douglas* is very apprehensive to weather conditions so that he looks at the weather forecast before setting the trip date. This just applies to *Mr. Douglas*, and it is not contemplated by *flightBooking*. Hence, *Mr. Douglas* is forced

to move outside the portal realm to satisfy this data need (e.g., through a *weatherForecast* widget), and to bridge himself the passing of data from the portal to the widget. By contrast, mashup-aware portals would assist *Mr. Douglas* in weaving the *weatherForecast* widget to the *flightBooking* service. Specifically, we focus on portlets [DR04] as the realization of portal services.

The rest of the chapter is structured as follows. Section 6.2 and 6.3 addresses the subject of variations and the time of variations with the help of an example. Section 6.4 outlines how to handle those variations using variability realization techniques. The main contribution of the chapter rests on section 6.5 that addresses the challenges posed for both portlet providers and portlet consumers. Finally, sections 6.6 and section 6.7 introduce the related work and conclusions, respectively.

6.2 What Can Vary

This work regards portal mashups as enhancements provided by users but accomplished through the portal. This introduces a distinction among the tasks to be reached through the portal: main tasks and mashup tasks (e.g., *weatherForecast*). *Main tasks* are set by the portal administrator. They support the functional backbone of the portal. Being complex tasks, they tend to be realized as **portlets**. 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. 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) [DR04]. The latter is worth noticing: service accomplishment normally requires a sequence of markups that are progressively rendered to the user. For instance, the *flightBooking* service can be realized as a portlet that provides a set of markups for airport/date setting, site booking, entering billing data, and so on.

On the other hand, *mashup tasks* are subordinated to main tasks.

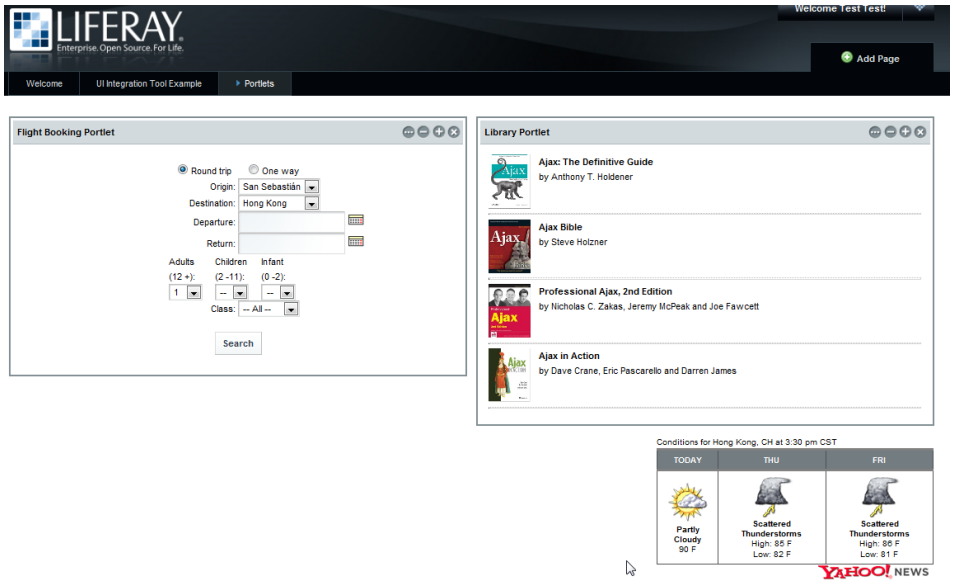


Figure 6.1: Side-by-side composition.

Normally, they consult and provide additional data rather than updating the service state. Unlike main tasks, mashup tasks can be set by portal users. We choose **widgets** as the realization technology for ancillary tasks. Widgets are full-fledged client-side applications that are authored using Web standards and packaged for distribution [W3C08b].

Traditionally, portal services are readily presented as you enter the portal page. Figure 6.1 provides a sample case. Two portlets (i.e., *flightBooking* and *librarySearch*), and one gadget[2] (i.e., *weatherForecast*) are rendered together. The figure illustrates traditional composition (a.k.a. "side-by-side composition"). As soon as you enter a portal page, portlets/gadgets are all made readily available. This approach is based on the premise that tasks are all equal, regardless of how they are supported.

By introducing mashup tasks, this approach now falls short. Side-by-side composition is still possible: portlets and gadgets can be co-located in portal pages. However, mashup tasks should not be readily available but only when they are needed. Otherwise you will end up with cluttered

[2]Gadgets are a concrete technology for widgets.

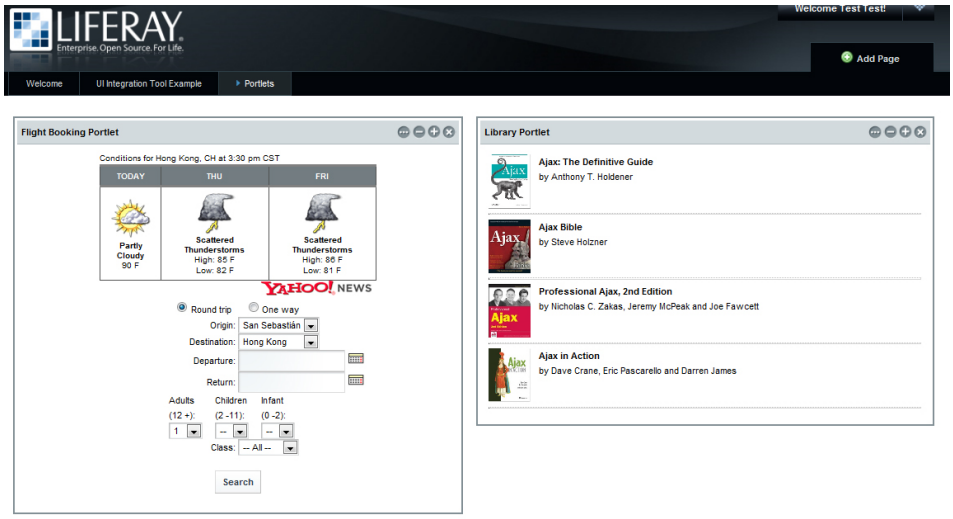


Figure 6.2: Inlay composition.

portal pages full of gadgets with no obvious purpose. The purpose of a mashup task should be sought in the context of a main task. In our previous example, *weatherForecast* only makes sense when *flightBooking* reaches the point of prompting for the destination airport. Once *flightBooking* moves to the next stage, *weatherForecast* is of no use. Additionally, the *weatherForecast* gadget should be located the closest to the entry form for the destination airport. However, side-by-side composition would assign *flightBooking* and *weatherForecast* distinct (although co-located) cells. It is then possible for the departure airport to appear at the bottom of the cell while forecast data is rendered upper on the page.

Mashup-aware portals depart from side-by-side composition by permitting portlet markup to become the canvas (i.e., the rendering space) for gadgets. Figures 6.2 provides an example. First, *flightBooking* and *library* are presented side-by-side as realization of main portal tasks. By contrast, mashup tasks such as *weatherForecast*, can now be inlayed within the rendering space of *flightBooking*.

Therefore, what varies here is the portlet markup that can be now

extended with different mashup markups depending on the current user, whereas portlet's data and business logic remain untouched.

6.3 When Can It Vary

The design decision here is to support the addition of variants by end-users. For instance, *Mr. Douglas* prefers the *weatherForecast* widget to be inlayed within the *flightBooking* services, whereas another user may prefer to check his *agenda*. This implies that variation points should be open for adding new variants at runtime.

6.4 How Is It Supported

When providing variability to end-users, one cannot expect end-users to edit and compile source code, so any variability technique that requires this would be unsuitable for this kind of variant feature. Additionally, the chosen variability realization technique must enable end-users to extend the portlet with new variants at runtime. Such variants are normally referred to as plug-ins and these may often be developed by third parties. In the case at hand, plug-in are implemented as widgets conforming mashups that have been developed by end-users using mashup development tools.

In this scenario, the chosen variability realization techniques must facilitate the modification of the portlet after delivery. This is the main reason to choose the binary replacement technique described in [SvGB05]. However, this techniques needs to be adapted in order to be applied to markup entities. This is the main contribution of this chapter and it is detailed in the next section.

6.5 Mashup-based Personalization

This scenario introduces three actors, namely

- the portal user. Once the portal is deployed, he can require additional data to better accomplish some portal services. Akin to the DIY approach, it is up to him to find the appropriate widget.

- the portlet provider. Portals provide the means to integrate services from third parties through portlets. Portlet providers ensure the quality of the service (data integrity, service throughput, etc.). Now, they are also responsible to decide how the service can be mashed up. Portlet designers should foresee placeholders to inlay additional information on accomplishing the portlet task (e.g., on selecting the destination airport),

- the portal (i.e., the portlet consumer). Akin to the portal-as-an-integration-space, portals should offer weaving mechanisms that permit data to seamlessly flow between main tasks and mashup tasks. *WeatherForecast* provides an example: its parameter *"location"* is to be obtained from the *flightBooking* destination airport, so that every time the airport is changed, the inlayed gadget is refreshed.

6.5.1 Realizing the Portlet Provider Perspective

Portlets support well-focus functionality: booking a fly seat, handling a bank transfer, and so on. On the other hand, portlets are born to be reused. The same portlet can be offered through different portals. As in any other component technology, this implies an attempt to foresee requirements for distinct potential consumers. However, traditional component development already advises that "*no design can provide information for every situation, and no designer can include personalized information for every user*" [Rho00]. This is when mashups come into play. Mashups permit portal users to complement portlet functionality. It is most important to notice that we *do not mashup the portlet as such but the offering of this portlet through this portal.* The very same portlet can have a different mashup when offered through another portal.

A similar situation arises in *XML Schema*. Schema standards are set by international bodies. Since the specificities of each sector/country can be difficult or inappropriate to be directly captured by the general schema, extension points are defined for consumers to adapt the schema to their own contexts. Likewise, portlet designers need to find a balance when supporting the portlet functionality, i.e., the portlet should be general enough to be appropriate for a large set of consumers while including "mashup placeholders" to cater for mashup specifics (hereafter referred to as *"mashcells"*). Their role is similar to the <*any*> element in XML schemas.

Mashcells do not have any presentation impact other than pinpointing where the portlet markup can be extended. Implementation wise, mashcells are supported as the *CSS* class *"mashcell"*. Figure 6.3 shows a snippet for a markup fragment for *flightBooking*. Notice that a portlet task is rarely accounted for through a single markup fragment. Rather, a portlet tends to involve a succession of markups that are accordingly presented to the user (e.g., enter trip details, select the seat, provide bill details, etc.). To this end, the *"view" CSS* class is introduced to denote each of these markup fragments. The sample markup corresponds to the *"search_form_view"*.

Mashcells permit portlet designers to foresee places where ancillary tasks can be inlayed to ease the user to accomplish the main task. For the sample markup, the designer decides to provide two mashcells right before and after the entry form: *"top-mashcell"* and *"bottom-mashcell"*. Portal tools can then light up these mashcells, pinpointing mashup points for portal users to fill up with the desired gadgets. This moves us to the consumer perspective.

6.5.2 Realizing the Portal Perspective

Mashup-able portlets exceed side-by-side composition. Portlets themselves now become "the canvas" where gadgets can be placed. The design space is then set on a portlet basis. Broadly, this space comprises

```
<div id="search_form_view" class="view">
  <div id="top-mashcell" class="mashcell"></div>
  <form method="post" action="<portlet:actionURL>...</portlet:actionURL>">
    <table width="100%" border="0" cellpadding="0" cellspacing="0"><tbody>
      <tr><!-- ROUNDTRIP FORM FIELD --></tr>
      <tr><!-- ORIGIN FORM FIELD --></tr>
      <tr>
        <td width="50%" align="right">Destination:</td>
        <td width="50%" align="left">
          <select name="destination">
              <c:forEach var="airport" items="${airports}">
              <option value="${airport.IATACode}">${airport.name}</option>
              </c:forEach>
          </select>
        </td>
      </tr>
      <!-- MORE FORM MARKUP -->
    </tbody></table>
  </form>
  <div id="bottom-mashcell" class="mashcell"></div>
</div>
```

Figure 6.3: Portlet markup with *mashcells* as mashup placeholders.

three dimensions (see Figure 6.4):

- *what* to include (i.e., gadget selection). The values of this dimension stand for the gadgets available at the portal. Permission can be granted for portal users to add their own gadgets, hence personalizing their own data purveyors,
- *where* is to be included. Values correspond to the mashcells available at the portlet at hand,
- *how* is to be included. Values stand for potential "data feeds" to be obtained from the portlet markup.

Figure 6.4 shows the design space for the *flightBooking* portlet. This space frames the setting for deciding *what-where* (hereafter, referred to as *"composition coordinate"*), and *where-how* (referred to as *"orchestration*

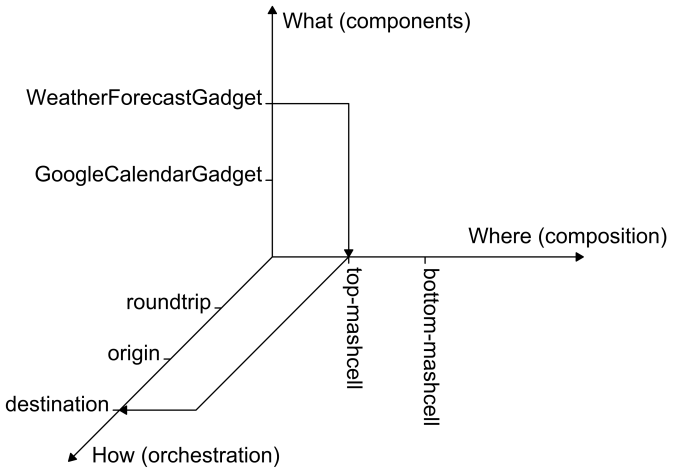


Figure 6.4: Design space for the *flighBooking* portlet.

coordinate"). For the sample problem, *weatherForecast* gadget is to be inlayed into the *"top-mashcell"* (composition coordinate). Additionally, this gadget is to be fed after the *destination* airport entry form (orchestration coordinate). Next paragraphs introduce two requirements to be fulfilled by the implementation technology: dynamic binding and presentation-based orchestration.

Dynamic binding: coordinates can be set once the portal is already deployed. Behaving as a kind of portal preferences for decision taken, gadgets and the associated coordinates can be added by portal users at any time. This hot-deployment of gadgets implies the ability to dynamically define coordinates.

Presentation-centred orchestration: gadget parameters can be obtained from portlet markup. Being both portlets and gadgets presentation components, it is just natural to use events for this purpose. Portlets already have a standard that permits portlets to synchronize their life cycles through events [Jav08]. But this leaves gadgets out. Departing from the specific technology, Yu et al. propose an orchestration model but now based on a canonical model for presentation components [YBSP+07]. Presentation components are encapsulated through an interface where

events and operations are specified. The orchestration model is next defined on top of these interfaces: events from component *C1* can trigger operations (i.e., state changes) in component *C2*. The approach requires the use of adapters to map platform-specific technologies (e.g., portlets, gadgets, etc.) into their platform-independent component model.

The proposal introduced in [YBSP+07] strives to abstract from the distinct technologies of presentation components, and provides a common framework where components can be seamlessly orchestrated. The approach brings several benefits: interoperability, portability or maintenance. However, it also imposes an important footprint:

1. on the search for generalization (i.e., a canonical model for presentation components), this indirection can eventually incur in some efficiency penalty. Platform-specific events (e.g., portlet events) need first to be mapped into the canonical framework. Next orchestration rules are triggered which, in turn, cause the enactment of operations. Finally, these canonical operations need to be mapped down to platform-specific operations on portlets/gadgets. These mappings from component-specific technology platforms to the component-independent canonical platform (and vice versa) occur at runtime.

2. on the search for abstraction, components offer high-level events (e.g., seat booked) rather than UI, low-level events (e.g., mouse over). This is certainly an advantage since it encapsulates the component implementation, and in so doing, shelters component consumers from changes in how the presentation is achieved (i.e., UI events). However, it also restricts the freedom of the consumer who is now limited to subscribe to those predefined events available at the component interface. This can be especially severe in case of using third-party components where component events are necessarily general.

On these grounds, we decide to stick to UI events. The fact of *HTML* being the standard for delivering rendering through the Web makes *HTML* the *lingua franca* for portlet-widget communication. That is, the *DOM API* is used [W3C00]. This *API* provides a set of low-level UI events (e.g., load, mouse over, etc.) to operate on low-level *UI* components (e.g., menus, button, etc.). Of course, this is far from abstraction but it provides generality: components, no matter whether they are portlets or gadgets, can "be operated" basing on the same *DOM* events. The only requirement is for the output to be *HTML* markup. That is, there is no need for the component provider to facilitate a high-level event interface. Additionally, this approach relies on an existing technology (mainly *DOM* and *JavaScript*) rather than requiring the portal administrator to become knowledgeable about a new component model. Therefore, we rely on *DOM* events to specify the orchestration model. This forces composition to take place at the client[3].

Having said that, we are also aware of the drawbacks this decision conveys. First, upgrades in the component markup can impact the orchestration. Second, maintenance of the orchestration model becomes more demanding. Being described in terms of low-level UI events rather than high-level events, the orchestration becomes more verbose and error prone.

Our contention is that this trade-off should be solved in a *portal* basis. Each portal administrator should balance the complexity and potential evolution of the portal's orchestration model *versus* the indirection costs caused by the "canonical" component model. It is worth noticing that, unlike mashups, portals offer a more controlled environment for composition. Portlet providers should be certified before delivering their portlets through the portal. In this setting, portal administrators can impose portlet providers to communicate any change in the portlet markup so that no sudden changes cause the orchestration to stop working.

[3] Actually, portlets produce the markup at the server while gadgets are scripts to be run at the client.

Based on these two requirements (i.e., dynamic binding, and client-based, *DOM* event-based orchestration), we select the *XML Binding Language (XBL)* [W3C07] as the implementation technology. Next subsection gives a brief on this language.

A Brief on XBL

XBL is a *W3C* candidate recommendation for describing bindings that can be attached to elements in other documents. It is currently supported by the main browsers. The element that the binding is attached to, called the *bound element*, acquires the new behaviour specified by the binding [W3C07]. An XBL document has <*bindings*> as a root. The <*bindings*> tag contains a collection of <*binding*> elements that describe element behaviour. Each <*binding*> can include the <*content*> tag, which is used to describe anonymous content that can be inserted around a *bound element*, an <*implementation*> tag that captures new properties and methods, and a <*handlers*> tag to account for event handlers on the *bound element*.

Bindings can be attached to elements through *CSS* using the *-moz-binding* property. Figure 6.5 shows an example where the *CSS* specification attaches *newBreeds* binding to the *CSS* class *breeds* elements. The binding is identified by referring to the *XBL* file using the # notation. The example shows how the dog-variation list changes as a result of applying the binding, in this case, by adding two additional dog variations.

Next sections look at how *XBL* can be used to realize both *composition coordinates* and *orchestration coordinates*. *XBL* files can be bound dynamically, hence, changing the coordinates based on either the profile of the current user or the portal configuration parameters.

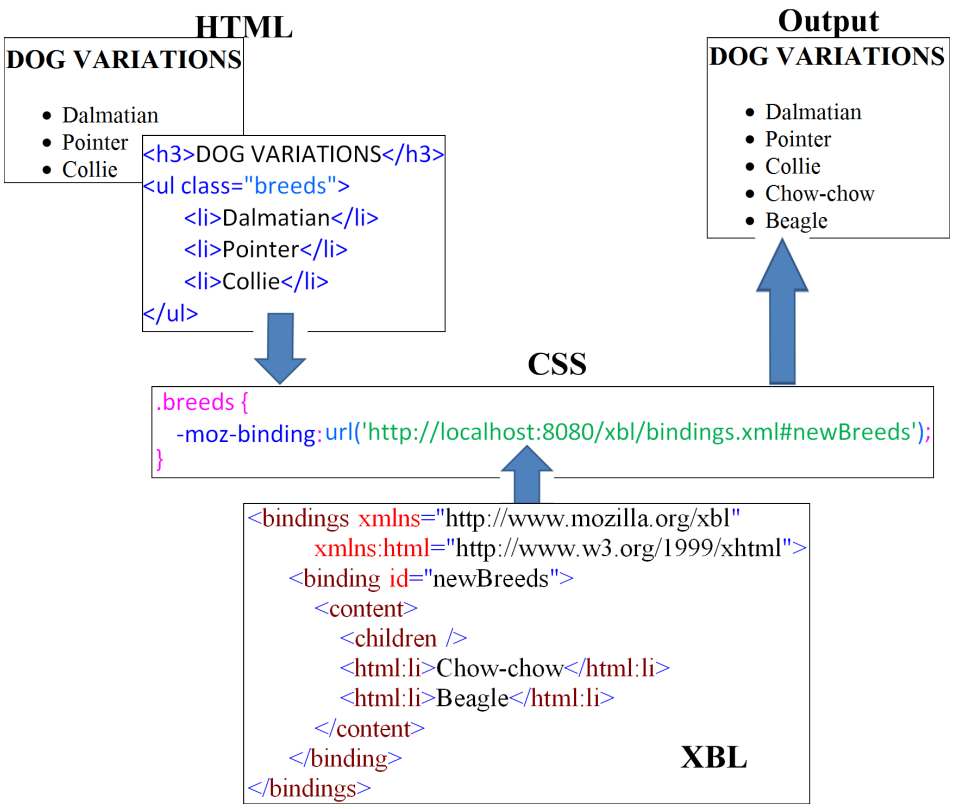


Figure 6.5: An XBL sample.

6.5.3 Setting Composition Coordinates

A composition coordinate *(what, where)* specifies what gadget is to be place in which mashcell. *XBL* wise, the binding provides the *what* as a <*content*> element, while the *CSS* provides the *bound element*, i.e., the *where*. Figure 6.6 provides an example for the coordinate *(WeatherForecastGadget, top-mashcell)*.

What. The binding *"gadget_21"* describes this gadget as anonymous content. Gadget preferences are supported as properties. In this case, getter and setter methods are provided for the two gadget preferences: *location* and *units*. This file is kept at *"localhost:8080/xbl/wrappers.xml"*.

Where. Gadgets are placed in mashcells. The *CSS* file in Figure 6.6 bounds the *gadget_21* binding to the "*top-mashcell*" to be found in the

```
<bindings       xmlns="http://www.mozilla.org/xbl"
                xmlns:html="http://www.w3.org/1999/xhtml">
 <binding id="gadget_21">
        <content>
            <html:iframe anonid="gadget_21" scrolling="no" style="display:block;" frameborder="no"></html:iframe>
        </content>
        <implementation>
            <field name="iframe"><![CDATA[ document.getAnonymousElementByAttribute(this,"anonid","gadget_21"); ]]></field>
            <field name="gadgetURL"><![CDATA[ "http://localhost:8080/shindig/.../WeatherForecastGadget.xml" ]]></field>
            <field name="moduleId">21</field>
            <field name="language"></field>
            <field name="country"></field>
            <field name="gadgetMetadata"></field>
            <constructor><!-- Initializes gadget's preferences & calls "refresh" method.--></constructor>
            <property name="location">
                <getter><![CDATA[ return this.getAttribute('location'); ]]></getter>
                <setter><![CDATA[ this.setAttribute('location',val); ]]></setter>
            </property>
            <property name="units">
                <getter><![CDATA[ return this.getAttribute('units'); ]]></getter>
                <setter><![CDATA[ this.setAttribute('units',val); ]]></setter>
            </property>
            <method name="store"><!-- Makes gadget's preferences persistent & calls "refresh" method. --></method>
            <method name="refresh"><!-- Asks the server for gadget's metadata & re-renders the gadget's markup. --></method>
        </implementation>
 </binding>
   ...
</bindings>
```

CSS

```
div[id="portlet-wrapper-FlightBookingPortlet2_WAR_FlightBookingPortlet2"]
*[id="search_form_view"]
*[id="top-mashcell"] {
   -moz-binding: url('http://localhost:8080/xbl/wrappers.xml#gadget_21');
}
```

Figure 6.6: *XBL* support for the composition coordinate *(WeatherForecastGadget,top-mashcell)*.

markup of the *flightBooking* portlet. This markup is shown in Figure 6.3: the *"top-mashcell"* div pertains to the *"search_form_view"* (notice that a portlet can deliver a set of markups) which in turn is wrapped by a portal decorator (a.k.a. portlet wrappers)[4]. The *CSS* addresses the whole portal page; hence the *CSS* style must identify a single cell within the portal page. This explains the three *ID* conditions in a raw found in the *CSS* sample.

6.5.4 Setting Orchestration Coordinates

Gadget parameters can be obtained from portlet markup. For instance, the *weatherForecast's location* parameter is to be obtained through subscription to changes in the *destination* form field of the *flightBooking* portlet. This stands for the orchestration coordinate *(top-mashcell,*

[4]The latter identification is needed since it is possible for the same portlet to be instantiated more than once in the very same portal page. That is, they can be distinct *flightBooking portlet* instances in the very same portal page.

```
<bindings       xmlns="http://www.mozilla.org/xbl"
                xmlns:html="http://www.w3.org/1999/xhtml">
        <binding id="onDestinationChange">
                <handlers>
                    <handler id="handler_1" event="change"><![CDATA[
                      // Call the extractors.
                      var extractor1_result = jQuery(this).val();

                      // Call the converter
                      jQuery.get("http://localhost:8080/emml/IATACode2YahooCode_Converter", {iataCode:extractor1_result}, callback);

                      function callback(yahooCode){
                      // Get the bound element.
                      jQuery("      *[id=portlet-wrapper-FlightBookingPortlet2_WAR_FlightBookingPortlet2]
                                    *[id=search_form_view]
                                    *[id=top-mashcell]").each(function(index, boundElement){
                                    boundElement.location = yahooCode;
                                    boundElement.store();
                            });
                      };
                    ]]></handler>
                </handlers>
        </binding>
        ...
</bindings>
```

CSS

```
div[id="portlet-wrapper-FlightBookingPortlet2_WAR_FlightBookingPortlet2"]
*[id="search_form_view"]
> form > table > tbody > tr:nth-child(3) > td:nth-child(2) > select {
   -moz-binding: url('http://localhost:8080/xbl/extensions.xml#onDestinationChange');
}
```

Figure 6.7: *XBL* support for the orchestration coordinate *(top-mashcell, destination)*.

destination).

Figure 6.7 illustrates the realization of this coordinate using *XBL*. A *binding* is specified that declares a handler on the *DOM* event *"change"*. The handler proceeds along three steps: (1) calling the data extractors, (2) enacting the converters for data transformation, if required, and finally (3), localize the *bound element* where the gadget resides. Data conversion is needed since data formats can differ between the provider and the consumer of the data. In this case, a converter is used to map *IATA* airport identification used by the portlet to *Yahoo!* city code utilized in the gadget.

Next, the companion *CSS* file associates the previous *onDestinationChange* binding to the form field that collects the *destination* airport. The *CSS* locates the *bound element* by first identifying the portlet decorator, next the portlet view, and finally the DOM node that holds the data. At enactment time, rendering this view will cause the handler to be bound to the form field so that any change in its value will cause to refresh the *weatherForecast* markup.

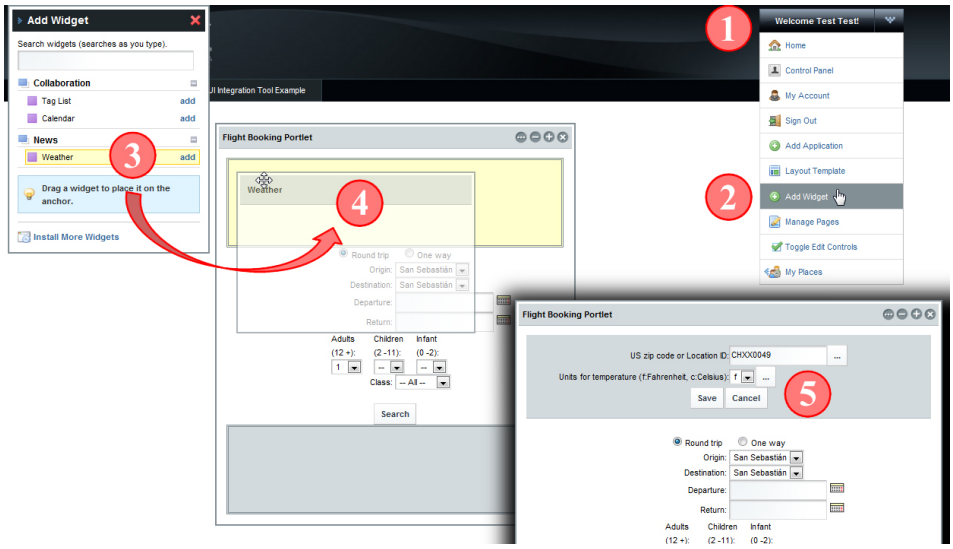


Figure 6.8: Mashup process: composition step.

6.5.5 Extending Portal Design Tools

Previous sections evidence the feasibility of mashup-able portlets using *XBL*. Portal developers master *CSS*, *HTML* or *JavaScript*, so learning *XBL* is just a question of hours. However, mashups aim to be an end-user technology. To this end, *XBL* should be hidden through GUI tools that permit portal users to define basic mashups based on their favourite gadgets. This section outlines such a tool for *Liferay*.

The tool achieves mashups in two steps: composition and orchestration. Composition starts by opening the dock menu (1) and selecting the *Add Widget* option (2). This causes the rendering of the *Widget palette* (3) (see Figure 6.8). So far, this palette is set by the portal administrator but the idea is for portal users to certify their favourite widgets to be uploaded into the portal (recall gadgets are a concrete technology for widgets). Next, the user selects a widget and drops it into a portlet mashcell (4). Available portlet mashcells are lighted up as the user moves around. Once the widget is dropped in a mashcell, the user can set the default values for the widget preferences (5). This concludes

the composition step that ends up with the generation of the corresponding *XBL* file.

So far, the gadget is inlayed into the portlet realm. A tightened integration can be achieved by feeding the gadget after the portlet data. To this end, (1) the user first selects the *"orchestration"* button (see Figure 6.9) that brings up the *Orchestration Editor* (2). By clicking on the *"new"* button (3), you are initiating the creation of the second *XBL* file behind the scenes. An *XBL* binding can now be generated for each gadget preference (e.g., *location*, *units*). Creation of a binding starts by prompting its name (4), indicating from where the data is to be obtained by clicking on the portlet markup (and realized as an *XPath* expressions (5)), and finally, pointing to the gadget to be fed (6). This defines a set of sources and a set of targets. Next, the user can draw a line from a source to a target that causes the generation of the *XBL* binding (7). However, not always the data can be consumed as extracted. In our sample case, *IATA* code should first be converted into *Yahoo!* city code. Constructing such converters is commonly outside end users' skills. *EMML* editors can be used to facilitate this task. A repository of converters can be available at the portal. In this case, the tool permits to display certified converters that can be inlayed into the piping process just as an intermediary link (8). In this way, a handler-based *XBL* file and its *CSS* counterpart as the one shown in Figure 6.7 is generated.

6.6 Related Work

Recently, there has been a proliferation of research papers about mashups. In [LHSL07], mashups as facilitators of Web Service composition is presented. Mashups encapsulate web services that are later composed through some additional *JavaScript* glue code. Like in our approach, *HTML* is used as the integration layer to achieve Web Service composition. On the other hand, *MashMaker* [EG07] and *MARGMASH* [DPnP07] allow end users to augment the content of an *existing* Web application

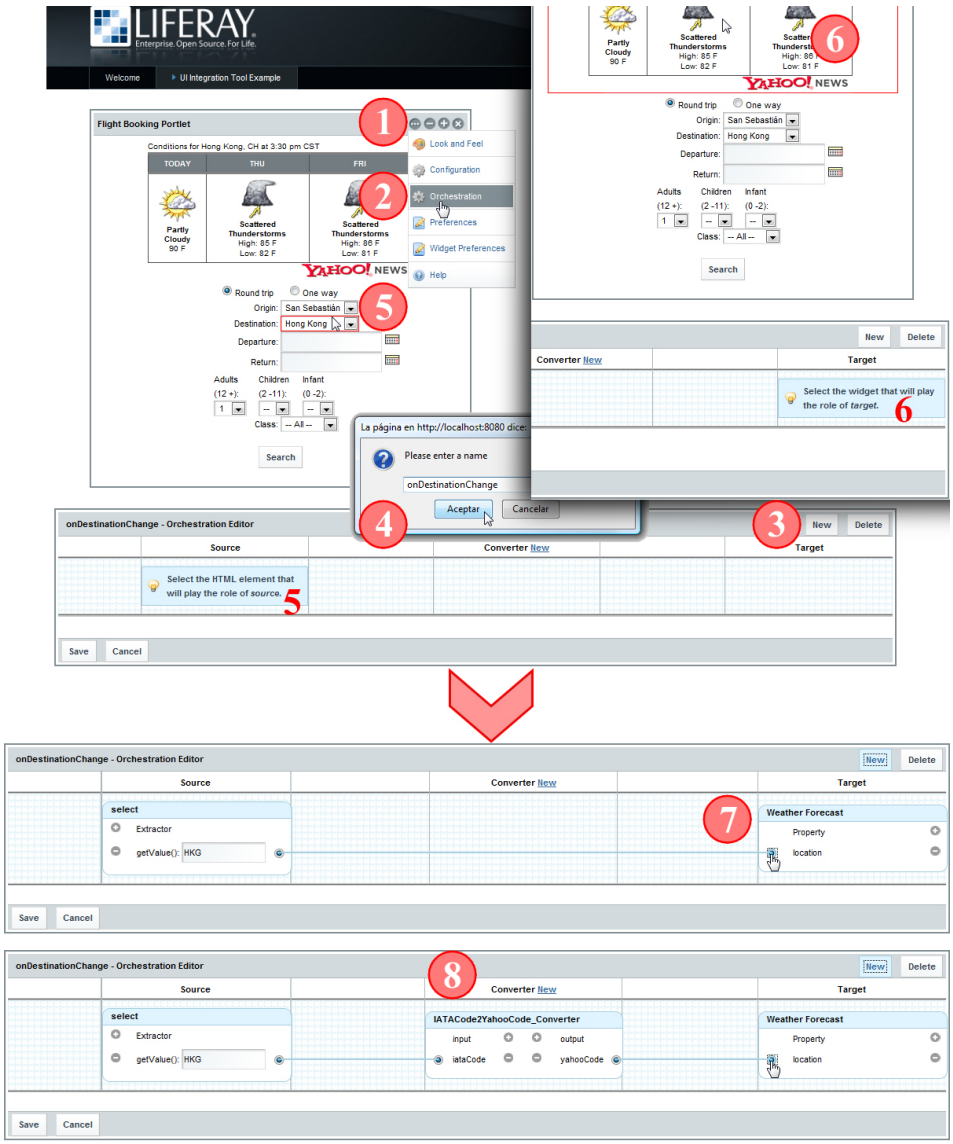


Figure 6.9: Mashup process: orchestration step.

by inserting mashups' markup throughout the web application's pages. However, extension points are not controlled by the existing application but screen-scraping techniques are used "to glue" the mashup to the existing code. Enterprise Information Portals offer a more controlled environment where *existing* portlets participate in the mashup effort by setting the extension points in their markup (i.e., the *mashcells*).

In [GZF+08], a new Web component called *gadget* is proposed. A gadget can interact to other gadgets through contract-based channels. Gadgets can contain other gadgets. This approach aligns to our efforts to permit presentation component to hold other components. The differences stems from the technological settings. Portals do not have the freedom that gadget exhibit. First, [GZF+08] proposes to extend *HTML* with a new *HTML* element—the so-called <*gadget*>. To avoid *ad-hoc HTML* extensions, our approach opts for using W3C candidate recommendation XBL to specify the bindings. Second, unlike gadgets, portlet markup cannot be extended at any place but at selected location pre-set by the portlet provider (i.e., the *mashcell* placeholders). Third, the portal frames the setting where the mashup is achieved. Although not yet explored, this permits to capitalize on portal utilities such as single-sign on, customization or resource sharing.

6.7 Conclusions

Enterprise Information Portals achieve front-end integration for presentation-oriented Web Services (a.k.a. portlets). As any other Web application, portlets can also be subject to mashup. However, their special characteristics (i.e., reusable components being offered by third parties) make portlet mashup a combined endeavour of portlet consumers and portlet providers. We have presented an architecture where portlet providers facilitate mashup placeholders (i.e., *mashcells)* to add companion widgets. As for portlet consumers, *XBL* bindings are used to dynamically bound user mashups to *mashcells*. The approach strives

to find a balance between portal reliability and mashup freedom. This architecture is borne out for *Liferay*.

Next follow-on includes capitalizing on the portal utilities for leveraging mashup. Single-sign on, access control, customization mechanisms are now at our disposal to adapt mashup techniques when achieved through a enterprise portal.

Parts of the work described in this chapter have been previously presented:

- Oscar Díaz, Sandy Pérez and Iñaki Paz. Providing Personalized Mashups Within the Context of Existing Web Applications. In proceedings of the 8th International Conference on Web Information Systems Engineering (WISE2007), Nancy, France, 2007.

- Sandy Pérez, Oscar Díaz. Mashup-Aware Corporate Portals. In proceedings of the 11th International Conference on Web Information Systems Engineering (WISE2010), Hong Kong, China, 2010.

Chapter 7

Conclusions

"Every new beginning comes from some other beginning's end."
-Seneca

7.1 Overview

Enterprise Information Portals (EIP) enable people to communicate and collaborate, provide a unified point of access to dynamic content from business applications, break down silos of content, and deliver information effectively through context-driven personalization. This makes EIP a must in today organizations [VMP+12]. Key to this vision is the notion of portlets.

Portlets are the fundamental building blocks of portals. Portlets are used by portals as pluggable user interface Web components. The *Java*™ *Portlet Specification* (formerly JSR 286) [Jav03, Jav08] establishes a standard API for ensuring portlet portability, i.e. developers can create one portlet and reuse it in any portal that supports the *Java*™ *Portlet Specification*. However, portability is not only a standardization matter. Portlets tend to be coarse-grained components since they encapsulate both the presentation layer and the functional layer. It is well-known in the component community that, the larger the component, the more reduced

the reuse. To make portlet reuse practical and effective, portlets need to offer some degree of variability in order to accommodate the differences between individual portals. This will certainly increase reusability, and hence, helping to pay off the upfront investment portlets require.

A complementary standard, *Web Services for Remote Portals* (WSRP) [ftAoSIS03, ftAoSIS08] defines a common interface and protocol for creating pluggable, user-facing, interactive Web services. JSR 286 portlets can be exposed as WSRP-compliant Web services. Therefore, portlets can be locally deployed or be provided remotely through third-party providers. For instance, a portal can offer the possibility of blogging, purchasing a book, or arranging a trip, all without leaving the portal. Some of these portlets can be built in house whereas others can be externally provided by, e.g., *Amazon* (amazon.com) or *Expedia* (expedia.com). However, where the notion of portlet gets its full potential is when portlets are deployed remotely, provided by third parties. This dissertation focuses on variability for remote portlets.

The WSRP protocol describes the conversation between portlet *Providers* and *Consumers* on behalf of *End-Users* :

- providers are presentation-oriented web services that host portlets which are able to render markup fragments and process user interaction requests,
- consumers use these web services as part of presenting markup to End-Users and managing the End-User's interaction with the markup,
- end-users are the clients of consumers

These roles constitute the different contexts in which **variability for remote portlets** can be set and decided.

- the portlet provider. Here, the portlet provider conducts a deep domain analysis and provide portlet consumers with a fixed set of variants to choose from.

- the portlet consumer. Here, the portlet consumers may also create and add their own product-specific variants, e.g., to account for requirements that can be met by using functionality external to the portlet.

- the end user. Additionally, there is an increasing trend to provide variability to end users (e.g., plugin mechanisms). However, we cannot expect end users to edit and compile source code.

This dissertation undertakes the challenge of supporting variability for each of these scenarios. This chapter reviews the main results of this work, assesses its limitations, and suggests work for future research.

7.2 Results

This dissertation develops the content of the research through three main chapters which are next summarized.

***Chapter* 4 is an example of the first scenario (i.e., provider-based variability).** It 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. The result is an organization-aware, WSRP-compliant architecture that lets portlet consumers register and handle "family portlets" in the same way that "traditional portlets".

***Chapter* 5 is an example of the second scenario (i.e. consumer-based variability).** It advocates for means to better account for the portlet consumer specifics. Tagging is used as an example. It argues for tagging to be orthogonally supported as a crosscut on top of portlets, i.e., as a portal commodity. Tagging functionality is up to the portal (i.e., the portlet consumer) but offered through companion portlets. In the same way that portlets adapt their rendering to the portal aesthetic,

tagging through portlets should also cater for the peculiarities of the consumer portal. Currently, most portal vendors support tagging as a portal functionality. The portal is regarded as a content manager. The portal owns the resources, and provides functionality for tagging. Tagging is restricted to those resources within the realm of the portal. Additionally, portals are also integration platforms, making external resources available through portlets. This outsource of content description does not imply that external resources are not worth tagging. This work highlights that tagging should be seamlessly achieved across the portal, regardless of the type (messages, books, hotels, etc.), or origin (i.e., *Amazon*, *Expedia*, etc.) of the resource. This is akin to portals as integration platforms. Implementation wise, this implies that portlets should be engineered to be plugged into this commodity. The result is a novel architecture where *RDFa* annotations are used as a means for portlets to communicate the portal the existence of *tag-able* resources, and portlet events as the mechanism for portals to broadcast tag-based queries to portlets.

***Chapter* 6 illustrates the third scenario (i.e., user-based variability).** Motivated by "*no design can provide information for every situation, and no designer can include personalized information for every user*" [Rho00], this chapter brings mashups into portlets through the combined effort of portlet providers and portlet consumers. Portlet providers facilitate mashup placeholders (i.e., *mashcells*) to add companion widgets. Portlet consumers resort to XBL bindings to dynamically bound user mashups to mashcells. The approach strives to find a balance between portal reliability and mashup freedom. Unlike traditional mashing scenarios: (*i*) the mashup is offered without leaving the portal, (*ii*) no additional plug-in is necessary since mashup weaving is already engineered into the portal, (*iii*) the portal "guides" users throughout the mashup process, and (*iv*), the portal provides the context for mashups to be seamlessly integrated into portal services. Additionally, a tool for generating XBL files has been developed. The tool is completely visual, and hides XBL technicalities.

Sandy Pérez González

Follow

Research Assistant, University of the Basque Country
Web Applications, Corporate Portals, Social Networks, Rich Internet Applications, Software Product Lines
Verified email at ehu.es

Title 1–8	Cited by	Year
A model-driven development for GWT-based Rich Internet Applications with OOH4RIA S Meliá, J Gómez, S Pérez, O Díaz Web Engineering, 2008. ICWE'08. Eighth International Conference on, 13-23	101	2008
Architectural and technological variability in Rich Internet Applications S Meliá, J Gómez, S Pérez, O Díaz Internet Computing, IEEE 14 (3), 24-32	35	2010
Facing interaction-rich rias: The orchestration model S Pérez, O Díaz, S Meliá, J Gómez Web Engineering, 2008. ICWE'08. Eighth International Conference on, 24-37	28	2008
Providing personalized mashups within the context of existing web applications O Dıaz, S Pérez, I Paz Web Information Systems Engineering–WISE 2007, 493	22	2007
Turning portlets into services: the consumer profile O Díaz, S Trujillo, S Pérez Proceedings of the 16th international conference on World Wide Web, 913-922	16	2007

Figure 7.1: Top five publications as for the number of references in Google Scholar [Accessed 8 December 2015].

7.3 Publications

Parts of the work explained in this thesis have been presented and discussed in distinct peer-reviewed forums. Figure 7.1 depicts the top five publications as for the number of references in Google Scholar.

Next we provide a more detailed account in terms of the different communities being addressed:

Web Engineering

- Oscar Díaz, Salvador Trujillo and Sandy Pérez. Turning Portlets into Services: The Consumer Profile. In proceedings of the *16th International World Wide Web Conference (WWW2007)*, Banff,

Alberta, Canada, 2007. Acceptance rate: 15%. *Rank **A+** in the CORE conference ranking* [DTP07].

- Oscar Díaz, Sandy Pérez and Iñaki Paz. Providing Personalized Mashups Within the Context of Existing Web Applications. In proceedings of the *8th International Conference on Web Information Systems Engineering (WISE2007)*, Nancy, France, 2007. Acceptance rate: 29%. *Rank **A** in the CORE conference ranking* [DPnP07].

- Oscar Díaz, Sandy Pérez and Cristóbal Arellano. Tagging-Aware Portlets. In proceedings of the *9th International Conference on Web Engineering (ICWE2009)*, San Sebastian, Spain, 2009. Acceptance rate: 24%. *Rank **C** in the CORE conference ranking* [DPA09].

- Sandy Pérez, Oscar Díaz. Mashup-Aware Corporate Portals. In proceedings of the *11th International Conference on Web Information Systems Engineering (WISE2010)*. Hong Kong, China, 2010. Acceptance rate: 30%. *Rank **A** in the CORE conference ranking* [PD10].

Model-Driven Web Engineering

- Sandy Pérez, Oscar Díaz, Santiago Meliá and Jaime Gómez. Facing Interaction-Rich RIAs: the Orchestration Model. In proceedings of the *8th International Conference on Web Engineering (ICWE2008)*, Yorktown Heights, New York, United States, 2008. Acceptance rate: 34%. *Rank **B** in the CORE conference ranking* [PDMG08].

- Santiago Meliá, Jaime Gómez, Sandy Pérez and Oscar Díaz. A Model-Driven Development for GWT-Based Rich Internet Applications with OOH4RIA. In proceedings of the *8th International Conference on Web Engineering (ICWE2008)*, Yorktown Heights, New York, United States, 2008. Acceptance rate: 34%. *Rank **B** in the CORE conference ranking* [MGPD08].

- Santiago Meliá, Jaime Gámez, Sandy Pérez and Oscar Díaz. Architectural and Technological Variability in Rich Internet Applications. In *IEEE Internet Computing* journal, 2010, 14, 24-32. ***JCR*** *(ranked fifth in the top 10 magazines and journals in the computer science software engineering category)*, Impact factor 3.108 [MGPD10].

- Sandy Pérez, Frederico Durao, Santiago Meliá, Peter Dolog and Oscar Díaz. RESTful, Resource-Oriented Architectures: a Model-Driven Approach. In proceedings of the *1st International Symposium on Web Intelligent Systems & Services (WISS2010)*. Hong Kong, China, 2010 [PDM+10].

Figure 7.2 summarises the citations to these publications. *Google Scholar*. Accessed 8 December 2015. https://scholar.google.es/citations?user=4JnaSTQAAAAJ&hl=en

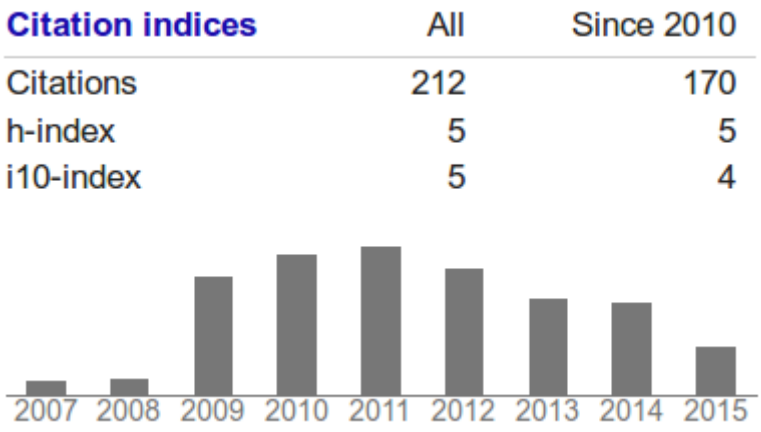


Figure 7.2: The author's Google Scholar metrics [Accessed 8 December 2015].

7.4 Research Stages

One of the outstanding benefits of performing a Ph.D. is the possibility of working together with international and well-regarded professionals, and above all, learning from them. The author performed a research visit from March to May of 2010 to the *Intelligent Web and Information Systems Research Group* (*IWIS*) at the Aalborg University, Denmark, under the supervision of Prof. Dr. Peter Dolog. This visit fostered discussion, broadened horizons and greatly helped to improve this work.

7.5 Assessment and Future Research

This dissertation explores and proposes solutions to implement variability in three scenarios of the portlet arena, which are not well cover by the mechanisms provided by current portlet standards: (1) the realization of Service-Oriented Architecture (SOA) using portals, (2) the provision of social tagging as a portal commodity and (3), mashup-based personalization. However, an objective assessment exposes some limitations of this work and shows there is still room for improvement and future research.

Realization of SOA using portals

- *Performance*. The presented architecture could be simplified by eliminating the need for the *applicationProducer*. To achieve this, generated portlets should be hot deployed directly in the portlet container residing in the *domainProducer*. This is possible by making use of the portlet container API. By eliminating a level of indirection, we hope to boost the performance of the whole system.

- *Validation*. Although the developed prototype evidence the feasibility of the proposed solution, real tests cases are needed to demonstrate that it really can be used in a real setting.

Provision of social tagging as a portal commodity

- *Expand the approach to others social networking services*. Although the proposed solution is focused on social tagging, nothing prevents it from being expanded to others social networking services such as rating or comments. To this end, the *PartOnt* (*Part*icipatory *Ont*ology) ontology needs to be augmented to capture the new concepts introduced by the new functionality. Additionally, the proposed REST API should be extended to cater for the new requirements.

- *Scalability*. Although the proof-of-concept demonstrates the feasibility of the approach, real tests cases are needed to fully test the scalability problems. In the proposed REST API, tag-based queries involve the execution of IN queries at the provider side in order to obtain the list of the resources tagged with a specific tag. These queries need to be measured with real examples involving large databases and large sets of resource URIs which conform the filter of the query.

- *Explore how the proposed REST API could be improved by the introduction of the Open Data Protocol (OData)* [ftAoSIS14]. OData is an open protocol to allow the creation and consumption of queryable and interoperable RESTful APIs in a simple and standard way. For example, it can be used to express query filters, define query options such as *orderby* or limit the number of returned entries.

- *Explore the viability of a JCR-compliant [Jav05] portlet provider*. As content managers, portlet providers could become standards content repositories, which could then be queried via standard services such as *Content Management Interoperability Services (CMIS)* [ftAoSIS12]. Although the CMIS specification does not specify how to interact with tags, Alfresco's *Enterprise Content Management* (`alfresco.com`) provides an example of how to

perform tagging via CMIS. However, this does not eliminate the need for agreements in commons terms and protocols.

Mashup-based personalization

- *Migration to HTML 5*. HTML5 may have implications in the realization of both the portlet provider perspective and the portal perspective (i.e., the portlet consumer). In the former case, for example, *mashcell* can be now supported via custom HTML elements (e.g., <mashcell/>) instead of CSS classes ( ). In the latter case, HTML 5 can help in obtaining and processing data feeds. For example, the element <input type=”email”> reveals the existence of an electronic mail address, which can now be validate as such.

- *Usability*. The prototype evidence the feasibility of the proposed solution. However, experiments should be conducted in order to determine how user friendly is the interface of the companion tool and how it can be improved.

- *Extend the approach to existent portlets*. The approach could be extended to make use of the mechanisms include in [DPnP07] in order to allow end users to identify which markup fragments play the role of the *mashup anchors* (i.e., *mashcell*).

7.6 Conclusions

This dissertation has shown how variability can be implemented in three scenarios where mechanisms provided by current portlet standards are not enough, namely:

1. the realization of Service-Oriented Architecture (SOA). Portals as integration platforms, offer an excellent conduit for realizing SOAs.

The thesis propose the use of Software Product Lines techniques that account for portlets to become truly reusable services,

2. the provision of social tagging as a portal commodity. That is, tagging functionality is up to the portal but offered through portlets. To account of the portal specifics, portlet events and RDFa annotations are used to build a novel architecture that allows portlets to be seamlessly plugged into portal tagging infrastructure,

3. mashup-based personalization. Here, mashing is regarded as an additional personalization mechanism whereby portal users can supplement portal services with their own data needs. An architecture is presented where portlet providers facilitate mashups placeholders and XBL bindings are used to dynamically bound user mashups to them.

The thesis develops the theoretical underpinnings, and provides different implementations as a proof-of-concept. All solutions are JSR-286 and WSRP compliant. However, while these implementations evidence the feasibility of the proposed solutions, real tests cases are needed to demonstrate that really pay off in a real setting. This will certainly imply moving from prototypes to products, and from users to customers as the target audience.

Bibliography

[AN07] Morgan Ames and Mor Naaman. Why We Tag: Motivations for Annotation in Mobile and Online Media. In *25th ACM SIGCHI Conference on Human Factors in Computing Systems*, 2007.

[Bat05] Don Batory. Feature Models, Grammars, and Propositional Formulas. In *9th International Software Product Line Conference*, 2005.

[BFG+02] Jan Bosch, Gert Florijn, Danny Greefhorst, Juha Kuusela, J. Henk Obbink, and Klaus Pohl. Variability Issues in Software Product Lines. In *4th International Workshop on Software Product-Family Engineering*, 2002.

[BKPS06] Jeffrey Blattman, Navaneeth Krishnan, Dean Polla, and Marina Sum. Open-Source Portal Initiative at Sun, Part 2: Portlet Repository. Published at `http://www.oracle.com/technetwork/systems/articles/portlet-repository-141454.html`, 2006.

[Bos00] Jan Bosch. *Design & Use of Software Architectures: Adopting and Evolving a Product-Line Approach*. Addison-Wesley Professional, 2000.

[BRPA05] L. Balzerani, D. Di Ruscio, A. Pierantonio, and G. De Angelis. A Product Line Architecture for Web Applications. In *20th ACM Symposium on Applied Computing*, 2005.

[BTT05] D. Benavides, S. Trujillo, and P. Trinidad. On the Modularization of Feature Models. In *1st European Workshop on Model Transformation*, 2005.

[CD03] Rafael Capilla and Juan C. Dueñas. Light-weight Product-Lines for Evolution and Maintenance of Web Sites. In *7th European Conference on Software Maintenance and Reengineering*, 2003.

[CDG+08] Nicole Carrier, Tom Deutsch, Chris Gruber, Mark Heid, and Lisa Lucadamo Jarrett. The business case for enterprise mashups. Technical report, IBM Corporation, 2008.

[CN01] Paul Clements and Linda Northrop. *Software Product Lines: Practices and Patterns*. Addison-Wesley Professional, 2001.

[DMG+08] Joan DiMicco, David R. Millen, Werner Geyer, Casey Dugana, Beth Brownholtz, and Michael Muller. Motivations for Social Networking at Work. In *ACM Conference on Computer Supported Cooperative Work*, 2008.

[DPA09] Oscar Díaz, Sandy Pérez, and Cristóbal Arellano. Tagging-Aware Portlets. In *9th International Conference on Web Engineering*, 2009.

[DPnP07] Oscar Díaz, Sandy Pérez, and Iñaki Paz. Providing Personalized Mashups Within the Context of Existing Web Applications. In *8th International Conference on Web Information Systems Engineering*, 2007.

[DR04] Oscar Díaz and Juan J. Rodríguez. Portlets as Web Components: an Introduction. *Journal of Universal Computer Science (J.UCS)*, 10(4):454–472, 2004.

[DR05] Oscar Díaz and Juan J. Rodríguez. Portlet Syndication: Raising Variability Concerns. *ACM Transactions on Internet Technology*, 5(4):627–659, 2005.

[DTA05] Oscar Díaz, Salvador Trujillo, and Felipe I. Anfurrutia. Supporting Production Strategies as Refinements of the Production Process . In *9th International Software Product Line Conference*, 2005.

[DTP07] Oscar Díaz, Salvador Trujillo, and Sandy Pérez. Turning Portlets into Services: The Consumer Profile. In *16th International World Wide Web Conference*, 2007.

[EG07] Robert J. Ennals and Minos N. Garofalakis. MashMaker: Mashups for the Masses. In *ACM SIGMOD International Conference on Management of Data*, 2007.

[ftAoSIS03] OASIS (Organization for the Advancement of Structured Information Standards). Web Services for Remote Portlets Specification v1.0, 2003.

[ftAoSIS08] OASIS (Organization for the Advancement of Structured Information Standards). Web Services for Remote Portlets Specification v2.0, 2008.

[ftAoSIS12] OASIS (Organization for the Advancement of Structured Information Standards). Content Management Interoperability Services v1.1, 2012.

[ftAoSIS14] OASIS (Organization for the Advancement of Structured Information Standards). Open Data Protocol v4.0, 2014.

[GBS01] Jilles Van Gurp, Jan Bosch, and Mikael Svahnberg. On the Notion of Variability in Software Product Lines. In *Working IEEE/IFIP Conference on Software Architecture*, 2001.

[GEM04] Paul Grünbacher, Alexander Egyed, and Nenad Medvidovic. Reconciling Software Requirements and Architectures with Intermediate Models. *Software and Systems Modeling*, 3(3):235–253, 2004.

[GFdA98] M. L. Griss, J. Favaro, and M. d' Alessandro. Integrating Feature Modeling with the RSEB. In *5th International Conference on Software Reuse*, 1998.

[GZF+08] Rui Guo, Bin B. Zhu, Min Feng, Aimin Pan, and Bosheng Zhou. Compoweb: A component-oriented web architecture. In *17th International World Wide Web Conference*, 2008.

[Jav03] Java Community Process. Java™Portlet Specification v1.0, 2003.

[Jav05] Java Community Process. Content Repository API for Java™v1.0, 2005.

[Jav08] Java Community Process. Java™Portlet Specification v2.0, 2008.

[JB02] Michel Jaring and Jan Bosch. Representing Variability in Software Product Lines: A Case Study. In *2nd International Software Product Line Conference*, 2002.

[JBZZ03] Stan Jarzabek, Paul Bassett, Hongyu Zhang, and Weishan Zhang. XVCL: XML-based Variant Configuration Language. In *25th International Conference on Software Engineering*, 2003.

[JGJ97] Ivar Jacobson, M. Griss, and P. Jonsson. *Software Reuse: Architecture, Process and Organization for Business Success*. Addison-Wesley Professional, 1997.

[JP14] Paul Johannesson and Erik Perjons. *An introduction to design science*. Springer, 2014.

[JRLR00] Mehdi Jazayeri, A. C. M. Ran, Frank Van Der Linden, and Alexander Ran. *Software Architecture for Product Families: Principles and Practice*. Addison-Wesley, 2000.

[JS00] Stan Jarzabek and Rudolph Seviora. Engineering Components for Ease of Customization and Evolution. *IEE Proceedings-Software*, 147(6):237–248, 2000.

[KCH+90] Kyo C. Kang, Sholom G. Cohen, James A. Hess, William E. Novak, and A. Spencer Peterson. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical Report CMU/SEI-90-TR-021, Software Engineering Institute, 1990.

[KKL+98] Kyo C. Kang, Sajoong Kim, Jaejoon Lee, Kijoo Kim, Euiseob Shin, and Moonhang Huh. FORM: A Feature-Oriented Reuse Method with Domain-Specific Reference Architectures. *Annals of Software Engineering*, 5:143–168, 1998.

[KLM+97] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-Oriented Programming. In *11th European Conference on Object-Oriented Programming*, 1997.

[Kne] Torben Knerr. Tagging Ontology - Towards a Common Ontology for Folksonomies. http://tagont.googlecode.com/files/TagOntPaper.pdf.

[KPRS03] Gerti Kappel, Birgit Pröll, Werner Retschitzegger, and Wieland Schwinger. Customisation for Ubiquitous Web Applications: a Comparison of Approaches. *International Journal of Web Engineering and Technology*, 1(1):79–111, 2003.

[Kru06] Charles W. Krueger. New Methods in Software Product Line Development. In *10th International Software Product Line Conference*, 2006.

[LHSL07] Xuanzhe Liu, Yi Hui, Wei Sun, and Haiqi Liang. Towards Service Composition Based on Mashup. In *IEEE Congress on Services*, 2007.

[MCnP+05] Mª Ángeles Moraga, Coral Calero, Iñaki Paz, Oscar Díaz, and Mario Piattini. A Reusability Model for Portlets. In *Web Information Systems Quality (WISQ 2005) Workshop*, 2005.

[MGPD08] Santiago Meliá, Jaime Gómez, Sandy Pérez, and Oscar Díaz. A Model-Driven Development for GWT-Based RIAs with OOH4RIA. In *8th International Conference on Web Engineering*, 2008.

[MGPD10] Santiago Meliá, Jaime Gómez, Sandy Pérez, and Oscar Díaz. Architectural and Technological Variability in Rich Internet Applications. *IEEE Internet Computing*, 14(3):24–32, May/June 2010.

[ML97] Marc H. Meyer and Alvin P. Lehnerd. *The Power of Product Platforms*. Free Press, 1997.

[MYWF07] David Millen, Meng Yang, Steven Whittaker, and Jonathan Feinberg. Social Bookmarking and Exploratory Search. In *10th European Conference on Computer-Supported Cooperative Work*, 2007.

[PBvdL05] Klaus Pohl, Günter Böckle, and Frank J. van der Linden. *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer, 2005.

[PD10] Sandy Pérez and Oscar Díaz. Mashup-Aware Corporate Portals. In *11th International Conference on Web Information Systems Engineering*, 2010.

[PDM+10] Sandy Pérez, Frederico Durao, Santiago Meliá, Peter Dolog, and Oscar Díaz. RESTful, Resource-Oriented Architectures: a Model-Driven Approach. In *1st International Symposium on Web Intelligent Systems & Services*, Hong Kong, China, December 2010.

[PDMG08] Sandy Pérez, Oscar Díaz, Santiago Meliá, and Jaime Gómez. Facing Interaction-Rich RIAs: the Orchestration Model. In *8th International Conference on Web Engineering*, 2008.

[Phi05] Gene Phifer. A Portal May Be Your First Step to Leverage SOA, 2005. Gartner, Inc.

[RAV+04] W. Clay Richardson, Donald Avondolio, Joe Vitale, Peter Len, and Kevin T. Smith. *Professional Portal Development with Open Source Tools: Java™Portlet API, Lucene, James, Slide*. Wrox, 2004.

[Rho00] Bradley J. Rhodes. Margin Notes Building a Contextually Aware Associative Memory. In *International Conference on Intelligent User Interfaces*, 2000.

[RJ05] Damith C. Rajapakse and Stan Jarzabek. An Investigation of Cloning in Web Applications. In *14th International World Wide Web Conference*, 2005.

[Str02] Howard Strauss. All About Web Portals: A Home Page Doth Not a Portal Make. In *Web Portals and Higher Education:*

Technologies to Make IT Personal. Jossey-Bass, A Wiley Company, 2002.

[SvGB05] Mikael Svahnberg, Jilles van Gurp, and Jan Bosch. A Taxonomy of Variability Realization Techniques. *Software—Practice & Experience*, 35(8):705–754, 2005.

[Tat05] Arthur Tatnall. Portals, Portals Everywhere. In *Web Portals: The New Gateways to Internet Information and Services*. Idea Group Publishing, 2005.

[TBD07] Salvador Trujillo, Don Batory, and Oscar Díaz. Feature Oriented Model Driven Development: A Case Study for Portlets. In *29th International Conference on Software Engineering*, 2007.

[The14] The OSGi Alliance. OSGi Core Release 6, 2014.

[TSMM08] Jennifer Thom-Santelli, Michael J. Muller, and David R. Millen. Social Tagging Roles: Publishers, Evangelists, Leaders. In *26th ACM SIGCHI Conference on Human Factors in Computing Systems*, 2008.

[VMP+12] Ray Valdes, Jim Murphy, Gene Phifer, Gavin Tay, and Mick MacComascaigh. Magic Quadrant for Horizontal Portals. Published at `http://www.gartner.com/id=2170615`, 2012.

[W3C00] W3C. Document Object Model (DOM) Level 2 Events Specification, 2000. http://www.w3.org/TR/DOM-Level-2-Events/.

[W3C07] W3C. XML Binding Language (XBL) 2.0, 2007. http://www.w3.org/TR/xbl/.

[W3C08a] W3C. RDFa Primer: Bridging the Human and Data Webs, 2008. http://www.w3.org/TR/xhtml-rdfa-primer/.

[W3C08b] W3C. Widgets Family of Specifications, 2008. http://www.w3.org/2008/webapps/wiki/WidgetSpecs.

[YBSP+07] Jin Yu, Boualem Benatallah, Regis Saint-Paul, Fabio Casati, Florian Daniel, and Maristella Matera. A Framework for Rapid Integration of Presentation Components. In *16th international conference on World Wide Web*, 2007.

[ZJ97] Pamela Zave and Michael Jackson. Four Dark Corners of Requirements Engineering. *ACM Transactions on Software Engineering and Methodology*, 6(1):1–30, 1997.

Acknowledgments

This work would never have been possible without the support of so many people I would like to thank. First and foremost, thanks are due to my supervisor Oscar Díaz for investing a great deal of time in this work from the early stages to the very end and for his continuous encouragement. Thanks to my colleagues at ONEKIN group: Arantza Irastorza, Cristóbal Arellano, Felipe Ibañez, Gorka Puente, Iker Azpeitia, Iñaki Paz, Itziar Otaduy, Jokin García, Jon Iturrioz, Luis M. Alonso, Maider Azanza and Salvador Trujillo. Not a day goes by where I don't learn something from at least one of you.

I would also like to express my gratitude to Peter Dolog, head of the Intelligent Web and Information Systems (IWIS) research group at the Aalborg University in Denmark, for allowing me to visit his group. I really appreciate his valuable comments and his effort in making my stay in Aalborg as comfortable as my home. Thanks to the people at IWIS group who made it such a great place to work. Particular thanks go to Frederico Durão and Karsten Jahn.

Part of this thesis is based on the collaborative work with Santiago Meliá, Jaime Gómez, Peter Dolog and Frederico Durão. They contributed their own ideas, tested mine and provided practical assistance throughout the project. It is a great pleasure to thank them for a very pleasant and fruitful collaboration.

Huge thanks to my family to whom this dissertation is dedicated. Last but not least, I want to thank to all my close friends (i.e., "*la cuadrilla*") that helped me to distract from the thesis matters.

VITA

Sandy Pérez González was born in Cienfuegos, Cuba on 7th of August, 1982, son of José Pérez Molina (father) and Juana Caridad González Hernández (mother). Sandy is brother of Maickel Pérez González. Sandy received the Bachelor of Science (BSc in 2006) and the Master of Science (MSc in 2008) in Computer Science at the University of the Basque Country (UPV/EHU). Currently, Peréz is a software architect and support analyst at the Indaba Consultores S.L.

Contact him at sandyperezglz@gmail.com.

This dissertation was typed by the author.

Summary

Portlet standardization efforts, namely, the Java\texttrademark Portlet Specification (formerly JSR 286) and the Web Services for Remote Portlets (WSRP) promise to make portlets into universal and reusable plug-and-play components, making it possible to build a portal by plugging portlets from different vendors into a portal from any vendor. To make portlet reuse practical and effective, portlets need to offer some degree of variability to increase their capability to be tailored to the diversity of settings through which they might be delivered. So far, portlet standards account for variability by accessing and storing persistent configuration data and user profile parameters whose values are provided by the portal at runtime. These mechanisms imply that portlets are equipped with a number of variants, and are able, at runtime, to select between them. However, this might not be enough.

The portlet architecture distinguishes between the portlet provider and the portlet consumer. The provider builds the portlet, the consumer integrates the portlet. Providers and consumer might not coincide. This forces providers to foresee the different scenarios in which the portlet will be used. This is not easy. Currently available configuration mechanism falls short to account for the different needs that might arise when integrating portlets in third-party portals.

This thesis explores three scenarios where mechanisms provided by current portlet standards are not enough, namely:

1. the realization of Service-Oriented Architecture (SOA). Portals as integration platforms, offer an excellent conduit for realizing SOAs. The thesis propose the use of Software Product Lines techniques that account for portlets to become truly reusable services,

2. the provision of social tagging as a portal commodity. That is, tagging functionality is up to the portal but offered through portlets. To account of the portal specifics, portlet events and RDFa annotations are used to build a novel architecture that allows portlets to be seamlessly plugged into portal tagging infrastructure,

3. mashup-based personalization. Here, mashing is regarded as an additional personalization mechanism whereby portal users can supplement portal services with their own data needs. An architecture is presented where portlet providers facilitate mashups placeholders and XBL bindings are used to dynamically bound user mashups to them.

The thesis develops the theoretical underpinnings, and provides different implementations as a proof-of-concepts. All solutions are JSR-286 and WSRP compliant.