Designing Volatile Functionality in E-Commerce Web Applications

Gustavo Rossi*, Andres Nieto, Luciano Mengoni, and Liliana Nuño Silva

LIFIA, Facultad de Informática, UNLP, La Plata, Argentina
{gustavo, anieto, lmengoni, lilica}@sol.info.unlp.edu.ar

Abstract. In this paper we present a flexible design approach and a software framework for integrating dynamic and volatile functionality in Web applications, particularly in e-commerce software. We first motivate our work with some examples. We briefly describe our base design platform (the OOHDM design framework). Next, we show how to deal with services that only apply to a particular set of application objects by clearly decoupling these services from the base conceptual and navigation design and by defining the concept of service affinity. We describe an implementation environment that seamlessly extends Apache Struts with the notion of services and service's affinities. Finally, we compare our approach with others' work and present some further research we are pursuing.

1 Introduction

Complex E-Commerce Web applications are hard to build and harder to maintain. While they initially comprise a myriad of diverse functionality, which makes development a nightmare, their evolution tend to follow difficult to characterize patterns; quite often, new services are added and tested with the application's users community to determine whether they will be consolidated as core application services or not. Moreover, there are services which are known to be temporary, i.e. they are incorporated into the application during some time and later discarded, or they are only activated in specific periods of time. In this paper we are interested in the design and implementation of those, so called volatile requirements and the impact they have on the design model and on the application's architecture. We present an original approach to deal with these requirements modularly; by clearly decoupling the design of these application's modules we simplify evolution.

There are many alternatives to deal with this kind of volatile requirements. One possibility is to clutter design models with new extensions. The main problem with this approach is that it involves intrusive editing and therefore it may introduce mistakes as new functionality is added or edited. A second possibility is to consider that volatile functionality does not deserve to be designed (as it is usually temporary) and deal with these changes only at the implementation level. This approach not only is error prone but it also de-synchronizes design documents with the running system, therefore introducing further problems.

* Also CONICET.

K. Bauknecht et al. (Eds.): EC-Web 2006, LNCS 4082, pp. 92 – 101, 2006.

Volatile requirements pose a new challenge: how to design and implement them in order to keep the previously described models and the implementation manageable [15,16]. For example suppose we want to support donations (e.g. as in Amazon after South Asian Tsunami in 2004); this functionality arose suddenly, and implied adding some new (fortunately simple) pages and links from the home page. This kind of additions are usually handled in an ad-hoc way (e.g. at the code level), making design documents obsolete.

Keeping design models up to date is not straightforward: Should we clutter the design models with these new navigation units and then undo the additions when the requirement "expires"? How do we deal with those requirements that are not only volatile but moreover they apply only to some specific objects (e.g. not to the complete set of one class's instances)? Should we modify some specific classes? Add new classes in a hierarchy? Add some new behaviors to existing classes? The main risk of not having a good answer to these questions is that the solution will be to patch running code, making further maintenance even harder.

In this paper we describe our model-based approach for dealing with volatile functionality. We describe a simple approach which can be easily incorporated into the design armory of existing methods. It comprises the definition of a Service layer, describing volatile services both in the conceptual and navigational models, and uses the concept of service's affinities as defined in IUHM [8] to bind new services with application objects. Services are more than just plain behaviors, but may encompass complete (conceptual or navigation) models. We also describe an implementation architecture to show the feasibility of our approach and an extension to Apache Struts that supports the architecture. To make the discussion concrete, we describe our ideas in the context of the Object-Oriented Hypermedia Design Method (OOHDM).

The main contributions of this paper are the following:

- We present a design approach for clearly separating volatile functionality, particularly when it involves the definition of new nodes and links in the Web application.
- We show how to integrate this functionality by using the concept of service affinity.
- Finally, we describe an implementation architecture and framework supporting our ideas.

The rest of the paper is organized as follows: In Section 2 we present some simple motivating examples. In Section 3 we describe the core of our approach by discussing services and affinities. In Section 4 we briefly describe and implementation approach. In Section 5 we compare our work with other related approaches and finally in Section 6 we present some concluding remarks and further work on this area.

2 Motivating Examples

In order to show what kind of volatile application functionality we aim to deal with, we next show some examples in the context of the Amazon Web application.

In Figure 1 we show part of the page of the last Harry Potter's book. Below the book information and the editorial reviews, there is an excerpt of an interview with

the author, and a link to the full interview which is only accessible from this book and not from others of the series (and certainly it does not make sense in other authors' books). The interview is an aggregation of questions and answers with hypertext links to other books and authors. One can assume that as time passes, this interview (now, a novelty) will be eliminated. We face two problems when designing this simple functionality: how to indicate that it is available from some specific pages, and being volatile, how to keep it separated from the rest of the design.

In Figure 2 we see the page of Rolling Stone's "A bigger Band" CD; in the end of the page (also shown in the Figure) we can see a link to a site for buying tickets for Stones' next concert in Argentina. The same link appears in all Stones' disks. It is reasonable to think that this functionality will be eliminated after the concert is over.

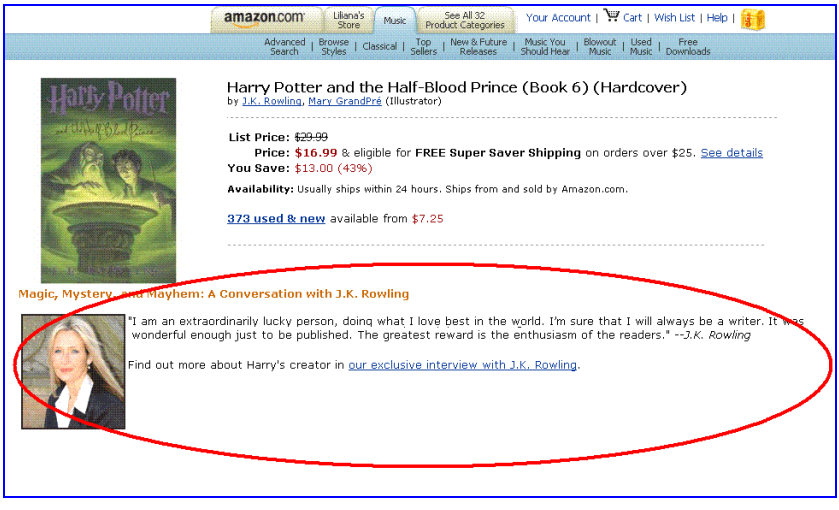

Fig. 1. Interviewing a book's author concert

Similar examples such as the functionality for full search inside a book, the Mozart store (celebrating his 250 anniversary), etc. share the same characteristics: they are known to be volatile and in some cases the new services only apply to some specific pages in the system. A naive approach for solving these problems would be to pollute the design model, by adding the intended information, links or behaviors to the corresponding conceptual and navigational classes. This approach fails because of two main reasons:

- It neglects the fact that certain functionality does not apply to a complete class (e.g., not every book is linked to an interview with the author, not every CD includes a pointer to a ticket selling service)
- It implies that the design models have to be often edited intrusively (e.g. changing attributes and behaviors of a class)

We next elaborate our approach for tackling these problems.

Fig. 2. Selling tickets for a group's concert

3 Our Approach in a Nutshell

The rationale behind our approach is that even the simplest volatile functionality (e.g. the links added to the page in Figure 2) must be modeled and design using good engineering techniques. We think that by surpassing the need to design volatile functionality, we not only compromise the relationships among design models and the actual application but also loose reuse opportunities, as many times a new (volatile) feature might arise once and again in different contexts. A model-based approach, instead, allows increasing the level of abstraction in which we reason with these features, improving comprehension and further evolution. We next present the basic elements of our approach.

3.1 A Brief Description of the OOHDM Model

OOHDM as other development approaches such as OOWS [9], UWE [6] partitions the development space into five activities: requirements gathering, conceptual design, navigation design, abstract interface design and implementation. Though OOHDM does not prescribe a particular strategy for implementing a hypermedia or Web application, its approach can be naturally mapped to object-oriented languages and architectural styles, such as the Model-View-Controller. Some MDA [7] tools already exist for OOHDM [2]; in this paper we describe a semi-automatic approach for generating running implementations which exhibit volatile services.

Usually, new behaviors (or services) are added to corresponding classes, and new node and link classes are incorporated to the existing navigational schema, therefore extending the base navigation topology. As previously indicated, there are two problems with this approach; first it is based on intrusive editing of design models; besides, and as exemplified, there is no easy way to characterize which objects should be the host of new links or services, when they are not defined in the class level. We next describe how we extended the methodology to cope with volatile functionality.

3.2 Modeling Volatile Functionality in OOHDM

Our approach is based on four basic principles:

- We decouple volatile from core functionality: We define two design models; a core model and a model for volatile features (called VService Layer).
- New behaviors, i.e. those which belong to the volatile functionality layer are modeled as first class objects, e.g. following the Command [4] pattern.
- To achieve obliviousness, we use inversion of control, i.e. instead of making core classes aware of their new features, we invert the knowledge relationship. New behaviors know the base classes on top of which they are built.
- We use a separate integration layer to bind core and volatile functionality. In this way, we achieve reusability of core and volatile features and manage irregular extensions.

3.3 The Volatile Services Layer

The introduction of the VService Layer was inspired in part in the IUHM model in which services are described as first class objects. We considered services as a combination and generalization of Commands and Decorators [3]. A service is a kind of command because it embodies an application behavior in one class, instead of a method. It can be considered also as a decorator because it allows adding new features (properties and behaviors) to an application in a non intrusive way. Services may be plain behaviors that are triggered as a consequence of some user action or might involve a navigational presence, i.e. a set of pages with information or operations corresponding to the service (as in Figure 1). We are particularly interested in this last kind of volatile services. Given a new (volatile) requirement we first model its conceptual and navigational features in a separate layer using the OOHDM approach. A second step is to indicate the relationships among services and existing conceptual and navigational classes; Figure 3 shows a preliminary specification of this connection. In the left we show the base model containing core (stable) application abstractions and in the right we present the specification of the service.

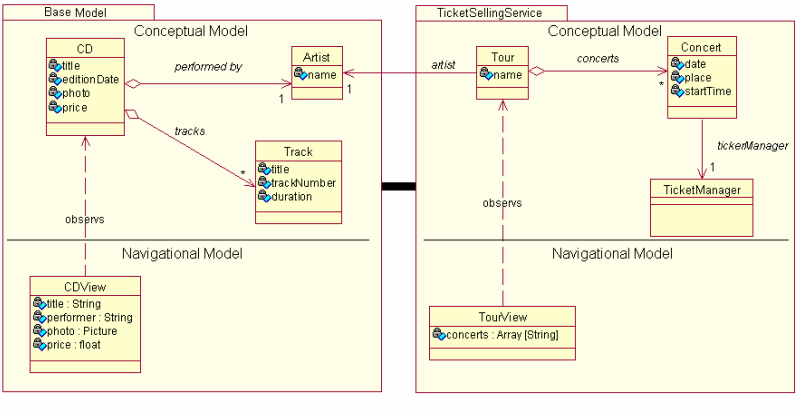

Fig. 3. Separating Volatile services from the base model

Notice the knowledge relationship among the Tour object and the performer's CD which inverts the "naive" solution in which Artists know the tour (thus coupling both classes), and the absence of link between the node CD and the Tour node. While the former is characteristic of Decorators, i.e. we are wrapping the model with a new service, the latter gives us the flexibility to specify different navigation strategies; for example we can either link the new functionality from the application (Figure 2) or insert it in the base node (Figure 1).

3.4 Integrating Volatile Services into the Core Design

VServices are connected to the application level using an integration specification, which is decoupled both from services and base classes. This specification indicates the nodes that will be enhanced with the volatile service, and the way in which the navigation model will be extended (e.g. adding a link, inserting new information in a node, etc). Notice that in the previous examples we aimed at extending only some specific instances of the CD (respectively Book) nodes. For example we might want to link some Rolling Stone's CD's to the ticket selling services for a concert in Argentina.

We define the affinity of a service as the set of nodes (respectively objects) in the design model which will be affected by the services, i.e. those nodes from which we will have access to the service. According to [8] we specify the affinity of a service in terms of objects' properties. Affinities are specified using a query language similar to the one that OOHDM itself uses for nodes specification [12] which was inspired in [4]. Those nodes which match the query are affected by the service. A query has the form: FROM $C_1 \ldots C_i$ WHERE *predicate* in which the C_j indicate node classes and the predicate is defined in terms of properties of the model. Queries can be nested and a generic specifier (*) can be used to indicate that all nodes can be queried. As in OOHDM, the qualifier *subject* allows to refer to conceptual model objects. A query indicates the kind of integration between application nodes and services which can be *extension* or *linkage*. An *extension* indicates that the application node is "extended" to contain the service information (an operations) while, in a *linkage* the node "just" allow navigation to the service. For example:

Affinity Concert
From CDView *where* (performer = TourView.subject.artist.name)
Integration: Linkage (TourView.name)

The affinity named Concert (corresponding to the example in Figure 2) indicates that all instances of a CD node will have a link to those instances of TourView such that the performers are the same. The link is enriched with the name of the tour. Service might of course have more than one instance; for example in the case of the second motivating volatile functionality, many artists may be on tours. Each tour ticket selling functionality has its own data and the most important remark, may have its own integration style into core nodes. Thus, we may have to specify an affinity for each service instance, which is called an Instance Affinity to differentiate it from a Class Affinity. The functionality in Figure 2 has the following integration rule:

Instance Affinity Concert
Where (artist=U2 and TourView.subject.location= "Argentina")
Integration: Extension (TourView)

An affinity specifies a temporal relationship between a service and the model which can be evaluated either during the compilation of the model, thus requiring re-compilation each time the affinity changes, or can be evaluated dynamically during page generation, as will be explained in section 4. Notice that model objects (conceptual and navigational) are oblivious with respect to services and their affinities and then they can evolve independently of their volatile nature.

3.5 Further Issues

We treat services as first class objects in our model. In consequence we can define services which apply to a service also using affinities, and therefore composing services in a seamless way, without a need to couple services with each other. A nice example is the following: Suppose that we want to offer a travel service in our e-store; the service may be a general one, i.e. accessible as a landmark (See for example www.amazon.com) or it must be accessible only when certain offers arise. For example, we could offer those people who buy tickets to a concert our travel service when the service takes place in a particular city. In this case we will specify an affinity between the travel service and the ticket service as for example:

Affinity TravelService
From TourView *where* (Subject.concert.place= “Paris”)
Integration: Linkage (TicketView)

Once again we obtain a clear separation between services and their target objects (being them base application nodes or services). The Travel service can be used in multiple other situations just by specifying corresponding affinities. A Service can be used for example in the context of a business process activity, e.g. as defined in [11], just by indicating the affinity and the target node (e.g. an activity node in the check-out process).We have also defined the concept of service specialization (a kind of inheritance in the realm of services) but for the sake of conciseness we omit to discuss this here.

4 Architectural Design and Implementation

We have implemented a framework on top of Apache Struts which supports semi-automatic translation of OOHDM models, including the instantiation of Web pages, from the OOHDM navigational schema and their integration with volatile services. The framework also provides a set of custom tags to simplify user’s interface development according to the guidelines of OOHDM’s abstract interface specification. A high-level description of the framework’s architecture is depicted in Figure 4.

Our light-weight framework aims to:

- Allow the specification, and the straightforward implementation, of a web application navigational model, which contains nodes and links primitives such as those defined in OOHDM.
- Provide support for dynamic integration of volatile functionality.

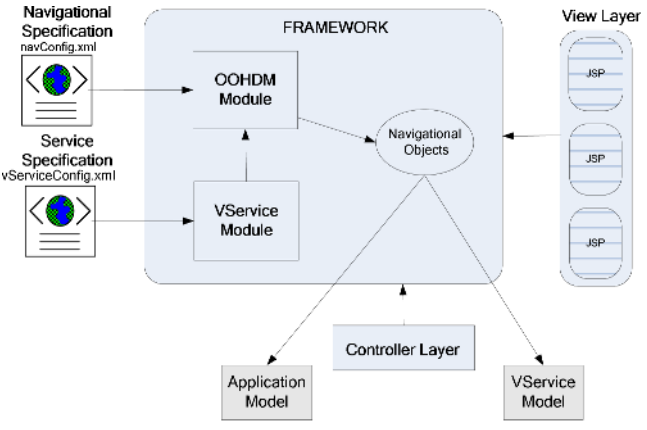

Fig. 4. Architecture of a framework for volatile services

The OOHDM module provides tools to represent a navigational layer between application domain objects and the user interface. We use the standard Struts controller objects to act as navigation controllers and to perform the interaction with conceptual objects. In this module the developer defines actions and links which allow representing the concepts in a navigation schema. This module offers support for defining nodes which contain the information which will be displayed in a page and profits from Struts custom tags for defining interface issues. Nodes contain Struts actions to manage navigation logic which is completely delegated to the Struts basic engine. The OOHDM module receives the navigational model in the form of a configuration file (NavConfig.xml) in which the designer specifies nodes, links and other navigation primitives. The information is processed and transformed into navigational objects which constitute the navigation layer of the application.

The volatile service module supports the integration of volatile functionality in a non-intrusive way; i.e. by releasing the developer from re-factoring existing classes or configuration files. This module is in charge of administrating and gluing volatile services in the target application, and uses the OOHDM module as a collaborator, delegating controller and navigation tasks to it. As mentioned before, a service is composed of a set of navigational nodes and conceptual objects that comply with a specific requirement. Nodes affinities are computing according to the actual state of the node's context which is defined as the set of direct and indirect relationships with other nodes and conceptual objects. The developer also provides all service information through a configuration file.

5 Related Work

Volatile requirements have been recently dealt by the requirements engineering community; particularly in [10] the authors propose to use aspectual constructs to isolate volatile requirements, mainly when they cross-cut "core" application behavior. Our approach also aims at separating these concerns, but without using aspects.

Web Engineering Methods have already faced the problems of e-commerce applications. Particularly, OOHDM [11] and UWE [5] have enriched their modeling armory for representing business processes. These methods have also defined means to personalize general application behavior and specifically business processes. OOWS [9] has also exemplified many of their features for specifying complex functionality using e-commerce software as a target. None of these methods have already explicitly dealt with volatile functionality. However, OOWS has been recently extended to incorporate external services in the context of business processes using a model-driven approach [13]. In [1], the authors present an aspect-oriented approach for dealing with adaptivity. In both cases, the concept of affinity could be easily introduced to mediate in the context of service integration in OOWS or adaptive aspects weaving in UWE.

6 Concluding Remarks and Further Work

In this paper we have presented an approach for dealing with volatile functionality in Web applications, i.e. for integrating those services which arise during evolution and are either known to be temporary or are being tested for acceptance. Incorporating this functionality in the conceptual and navigational model of a Web application might cause maintenance problems due to the need of editing classes which work properly or to clutter the existing model with possible spurious classes. We propose to add a separate layer for specifying volatile functionality. We have exemplified our approach with some simple examples and presented a way to integrate the VService Layer into the core application schemata, by using the concept of affinity. Affinities, which are expressed as queries, allow connecting services into those application objects which fulfill the desired properties. We have briefly described an implementation architecture that supports the evaluation of affinities and the injection of components defined in the VService layer into the core application objects.

We are studying the implication of service inheritance and composition and analyzing the integration of external services (e.g. Web Services). We are currently testing the described framework with demanding applications (e.g. those in which heavy queries must be executed). We are also studying the process of service integration via re-factoring of model classes.

References

1. H. Baumeister, A. Knapp, N. Koch and G. Zhang. Modelling Adaptivity with Aspects. 5th International Conference on Web Engineering (ICWE'05). Springer Verlag, Lecture Notes in Computer Science.
2. M. Douglas, D. Schwabe, G. Rossi, “A software arquitecture for structuring complex Web Applications” Journal of Web Engineering, Rinton Press, September 2002.
3. E. Gamma, R. Helm, R. Johnson, J. Vlissides: Design Patterns. Elements of reusable object-oriented software, Addison Wesley 1995.
4. W. Kim, "Advanced Database systems", ACM Press, 1994.
5. N. Koch, A. Kraus, C. Cachero, S. Meliá: Modeling Web Business Processes with OO-H and UWE. 3rd International Workshop on Web Oriented Software Technology (IWWOST03), Oviedo, Spain, 2003.

6. Koch, N., Kraus, A., and Hennicker R.: The Authoring Process of UML-based Web Engineering Approach. In Proceedings of the 1st International Workshop on Web-Oriented Software Construction (IWWOST 02), Valencia, Spain (2001) 105-119
7. OMG Model-Driven-Architecture. In http://www.omg.org/mda/
8. M. Nanard, J. Nanard, P. King: IUHM: A Hypermedia-based Model for Integrating Open Services, Data and Metadata. Proceedings of Hypertext 2003; ACM Press, pp 128-137.
9. O. Pastor, S. Abrahão, J. Fons: An Object-Oriented Approach to Automate Web Applications Development. Proceedings of EC-Web 2001: 16-28
10. A Rashid, P Sawyer, AMD Moreira, J Araujo Early Aspects: A Model for Aspect-Oriented Requirements Engineering. Proceedings of RE, 2002, pp 199-202.
11. H. Schmid, G. Rossi: Modeling and Designing Processes in E-Commerce Applications. IEEE Internet Computing, January/February 2004.
12. D. Schwabe, G. Rossi: An object-oriented approach to web-based application design. Theory and Practice of Object Systems (TAPOS), Special Issue on the Internet, v. 4#4, October, 1998, 207-225.
13. V. Torres, V. Pelechano, M. Ruiz, P. Valderas: "A Model Driven Approach for the Integration of External Functionality in Web Applications" Proceedings of MDWE 2005. ICWE 2005 Workshop on Model-Based Web Engineering.
14. The UML home page: www.omg.org/uml/
15. A. Van Lamsweerde: Goal-Oriented Requirements Engineering: A Guided Tour Fifth IEEE International Symposium on Requirements Engineering (RE'01) p. 0249
16. D. Zowghi, A Logical Framework for Modeling and Reasoning About the Evolution of Requirements Proceedings of the 4th Pacific Rim International Conference on Artificial Intelligence, Cairns, Australia, 1996.