Model-Based Design of Volatile Functionality in Web Applications*

Gustavo Rossi[1,2], Andres Nieto[1], Luciano Mengoni[1], Nahuel Lofeudo[1],
Liliana Nuño Silva[1], Damiano Distante[3]

[1]LIFIA, Facultad de Informática, UNLP, La Plata, Argentina
[2]Also CONICET
[3]RCOST : Research Centre on Software Technology, Department of Engineering, University of Sannio, Italy
{gustavo,anieto,lmengoni,nlofeudo,lilica}@sol.info.unlp.edu.ar, distante@unisannio.it

Abstract

In this paper we present a model-based approach to integrate dynamic and volatile functionality in Web Applications. Our approach comprises an extension to the OOHDM design method and a software framework which supports the injection of volatile functionality into the design model. We first motivate our work by discussing the problems which arise when dealing with volatile functionality; some meaningful examples are presented. We briefly describe our design approach, showing how to decouple volatile functionality from the core design model. We finally describe an implementation framework which supports the presented ideas extending Apache Struts with the notion of services and service affinities. Finally, we compare our approach with others' and present some further research we are pursuing.

1. Introduction and motivation

Complex Web applications evolve permanently and even after being "finished", having satisfied the initial requirements, they often have to be expanded to accomodate new functionality. Some of this new code and design will become a permanent addition to the applications, and some of it it will be discarded after a short time.

Among many others, web portals and e-commerce sites are a clear example of evolution over time. Pressure from the market, the availability of new technologies and unforeseen business opportunities sometimes force the developers to quickly add code that they know it will be discarded a few days or weeks later, not justifying a complete redesign or test of the whole application. For example, an e-commerce site devoted to sell CDs could be expanded to sell the tickets to a band's concert; but the sale should only appear for as long as the band is in tour. A related real-world example of this is Amazon.com's "Fishbowl performance", which adds rich content (online video) to the pages that refer to any artist that participate on the program, but only for the time it lasts.

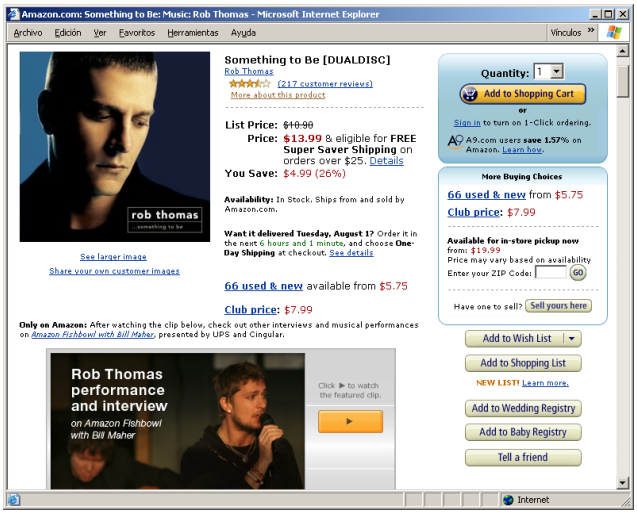

Figure 1. Fishbowl performance in Amazon.com

Another example could be a system to accept donations for a relief fund when a catastrophe occurs; clearly there is no time to redesign or test a complete application in this case because time is of essence, and it will only be needed as long as the emergency exists.

In the rest of this paper we will use the term "volatile" to refer to this kind of functionality which is by its own nature available for only a short amount of time. The word has been chosen to describe the ephemeral nature of the service or services added to an existing application.

Another, more complex example also based on volatile functionality can be seen in Figure 2. In this case EMI and

*This paper has been partially funded by SECYT under project PICT 13623

Amazon.com organize a draw of tickets to assist to a Rolling Stones' concert. In the top right corner of the figure, one can see the additional content that was added to the standard CD page.

Figure 2. Draw of tickets for a concert in Amazon.com

It is clear that all these examples encompass some kind of volatile requirements. In the first example, Amazon.com may eliminate the video if it is not widely used and perhaps when the program is canceled. In the case of the relief fund, the donations will be needed only while the emergency exists. In the last one, we even know in advance that the functionality will be eliminated after the concert.

There are many alternatives to deal with this kind of volatile requirements. One possibility is to clutter design models with new extensions. In an object-oriented approach, such as OOHDM [13] or UWE[8], this would mean to add new classes, or new attributes and methods to existing classes. For the previous example all CDs could have an attribute which contains the video performance and a method to support operations on video. This solution has an additional problem: as not every CD will support video operations we might be defining attributes which are not always usable.

Creating sub-classes (e.g. VideoEnhancedCD) is not a good solution because it introduces spurious specialization criteria and the sub-classing prevents us to add or suppress the video fragment dynamically on a particular instance.

In the example of the rock concert, meanwhile, we need to add the link to the tickets draw and the functionality to support it.

The main problem with the previous approach is that it involves intrusive editing of the existing code base and therefore it may introduce mistakes as new functionality is added or changed.

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 is not only error prone but it also de-synchronizes design documents with the running system, therefore introducing further problems.

In this paper we present a model-based approach in which not only we model and design volatile aspects, allows the core application model to remain oblivious to the introduction or removal of the volatile functionality, keeping the benefits of model-based design in a seamless way.

Our approach, based on a simple strategy of separation of concerns and inversion of control in design and a powerful integration environment, can be easily incorporated into the design armory of existing methods.

The main contributions of this paper are:

- A rationale for treating volatile services using a model-based instead of an implementation-based approach.
- A design approach for clearly separating volatile functionality, in both the conceptual and the navigational level of the application.
- An implementation architecture and framework which supports the seamless integration of volatile functionality.

The rest of the paper is organized as follows: In Section 2 we introduce our approach. In Section 3 we discuss the high level architecture of the support framework and in Section 4 we illustrate it with a case study. In Section 5 we detail the framework components. In Section 6 we discuss how to refactor this kind of functionality when it becomes part of the application's core. In section 7 we compare our work with other related approaches and finally in Section 8 we present some concluding remarks and further work on this area.

2. An overview of our approach

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 designed using good engineering techniques. It is evident that adding a link to a single page can be easily handled using modern implementation environments, and perhaps without much burden at the code level. However, by surpassing the need to design volatile functionality, we not only compromise the relationships among design models and the actual application but also lose reuse opportunities, as many times a new volatile feature might arise in different contexts. A model-based approach, instead, allows increasing the level of abstraction of

these features, improving comprehension and further evolution. Additionally, it can be automated to produce running implementations [10].

We next present the basic elements of our approach.

2.1. The OOHDM design framework

OOHDM as other development approaches such as OOWS [12] UWE [8] or WebML [2] partitions the development space into five activities: requirements gathering, conceptual design, navigational design, abstract interface design and implementation which have been extensively discussed elsewhere. As OOHDM follows the object-oriented paradigm, when the application evolves new classes emerge, new behaviors or attributes are added to corresponding classes, and new node and link classes are incorporated to the existing navigational schema, therefore extending the base navigation topology.

There are two problems with this approach when we use it in the context of volatile functionality: first, as explained before, it is based on intrusive editing of design models; and, as shown in the examples, volatile functionality might follow irregular patterns, e.g. not all instances of a class may support a new feature, not all pages of the same type contain the same link or host the same information. A naïve use of the object-oriented paradigm would result in the definition of new sub-classes which, as indicated in the introduction may solve the problem but at a high cost regarding modularity and further evolution. In the following subsections we elaborate our design strategy for coping with volatile functionality.

2.2. 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.
- We use inversion of control to achieve obliviousness of the core model , 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. Thus we achieve reusability of core and volatile features and manage irregular extensions.

2.3. The volatile services layer

In our approach we consider services as a combination and generalization of Commands [4] and Decorators [4]. A service[1] is a 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. In our approach, services might be new *behaviors* which are added to the conceptual model (and which might encompass many classes) or full-fledged *navigation models*, containing new nodes, links and even relationships with conceptual classes. We treat each volatile requirement as a self-contained sub-system and model it using the OOHDM approach. The development process is shown in Figure 3.

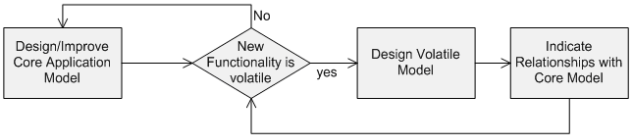

Figure 3. Development process for volatile functionality

Given a new (volatile) requirement we first model its conceptual and navigational features in a separate layer using the OOHDM approach; then we indicate the relationships among services and existing conceptual and navigational classes; finally (and not shown in the picture) we indicate the way in which volatile and core models are integrated. In Figure 4 we show the resulting OOHDM model corresponding to the example in Figure 2.

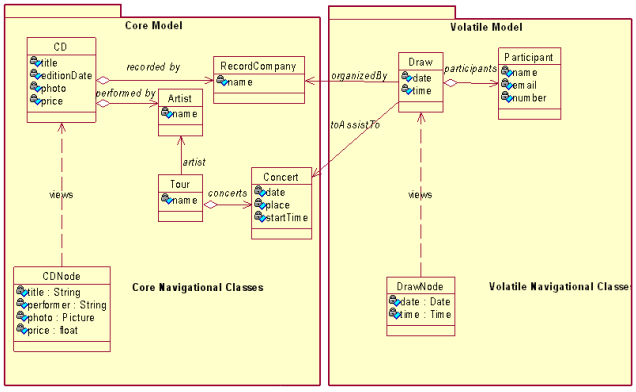

Figure 4. The OOHDM comprising the VModel

In the left of Figure 4 we show the base model containing core (stable) application abstractions and in the right we

[1] We use the term *Service* to refer to a piece of volatile functionality. This term shouldn't be confused with the popular concept of *Web Service* as it embodies a more general feature.

present the specification of the volatile service. On the top we show the two conceptual models (core and volatile) and on the bottom a single class representing a part of the navigational model.

Notice the knowledge relationship among the Draw class and the Record Company class which inverts the "naïve" solution in which the company knows the draw (thus coupling both classes), and the absence of links between the CDNode and the DrawNode classes. While the former is characteristic of Decorators (we are wrapping the model with a new service) the latter gives us the flexibility to specify different integration strategies; for example we can either link the new functionality from the application (as shown in the example of Figure 2) or insert it in the base node (as in Figure 1). For the sake of simplicity we don't show in Figure 4 the process for determining the winner of the Draw, nor the procedure to notify the winner (perhaps an email) which are part of the volatile functionality.

2.4. The connection layer

VServices are connected to the application level using an integration specification, decoupled both from services and core classes thus making these classes oblivious to the integration strategy. The specification indicates what nodes will be enhanced with the volatile service, and how the navigation model will be extended (e.g. adding a link, inserting new information in a node, etc). The set of nodes affected (or enhanced) by a service is called the Affinity of the service and was inspired by [11]. We specify affinities with the same query language used in OOHDM to specify nodes [7, 13]. Those nodes which match the query are affected by the service.

A query has the form: FROM $C_1..C_J$ WHERE PREDICATE in which the C_j indicate node classes and the predicate is defined in terms of properties of model objects. As in OOHDM, the qualifier subject allows to refer to conceptual model objects. A query expression also indicates the kind of integration between application nodes and services, which can be Extension or Linkage. An extension indicates that the application node is enhanced to contain the service information (an operations) while, in a linkage the node "just" allows navigation to the service and does not support new behaviors. In the case of linkage extensions we can also specify additional features such as attributes or anchors that have to be added to the extended node. For example:

```
Affinity Draw
    From CDNode
    where (performer = Rolling "Stones")
    Integration:  Linkage (DrawNode)
    Additions:  [DrawSpec:  Text.
        Conditions:  Anchor (ToConditions)
        Results:  Anchor (ToResults)]
```

The affinity named Draw (corresponding to the example in Figure 2) indicates that all instances of a CD performed by Rolling Stones will have a link to DrawNode. Notice that the integration specification does not compromise the granularity of Draw, which might be a Singleton (only one instance) or not.

Service classes might of course have more than one instance; for example in the case of the first motivating volatile functionality, fish bowl performances may be dedicated to different singers, book writers, directors, actors, etc. Each single fish bowl program 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 1 has the following integration rule:

```
Instance Affinity RobThomas (Fishbowl)
    FROM CDNode
        WHERE title = 'Something to Be'
        AND performer = 'Rob Thomas'
    Integration:  Extension
```

Other example of the same volatile service is the interview to Stephen King, talking about his last novel "Cell", which is only available in the book's page

```
Instance Affinity StephenKing (Fishbowl)
    FROM BookNode
    WHERE author = 'Stephen King'
        AND bookTitle = 'Cell'
    Integration:  Extension
```

Notice that we could have integrated this service instance using another strategy if necessary.

Our approach has many advantages, the most important being:

- By decoupling the integration from the services and the core model, we can use different integration schemas at different time and for different instances. For example, with new CDs it is reasonable to extend the CD with the live performance; we can later use a Linkage relationship and, once the CD is no longer a novelty, eliminate the access to the service entirely.
- Using different affinity specification we can fine tune the way in which different objects are affected by the service.
- Obliviousness allows the evolution of core and service classes independently from each other and from the integration specification.

An affinity specifies a temporal relationship between a service and the core model which can be evaluated either

during the compilation of the model, thus requiring recompilation each time the affinity changes, or dynamically during page generation, as will be explained in section 4.

2.5. Service composition

As services are modeled using standard OOHDM primitives, we can also treat them as core models regarding other services which might apply to them. In this way we can compose services seamlessly and particularly we may use core (sub) models as services if necessary.

An interesting example of this reflective property of our approach relates with the functionality described in Figure 2. Suppose that our store offers a travel service. This service may be defined as a full-fledged sub-system and accordingly modeled with OOHDM; the service might be accessed from a Landmark as the sub-systems in www.expedia.com, and once selected it may require that the user enters the kind of service (car rental, hotels or flights), destination, dates, etc.

Additionally, we can arrange with a travel company to offer our travel service as part of the draw of tickets to the Rolling Stones concert in a given city by allowing to include the hotel booking in the Draw. In this case we will specify an affinity between the travel service and the Draw service as below:

```
Affinity TravelService
    From DrawNode
    where (Subject.toAssistTo.place= 'Paris')
    Integration:  Linkage (HotelNode)
```

When the user enters his data for the draw, he can additionally book a hotel in the city by choosing the newly offered link. The integration specification (here presented in an oversimplified style) could contain the necessary adaptation to make the two services work together.

Once again we obtain a clear separation between services and their target objects (being them core application nodes or other services). The Travel service can be used in multiple other situations just by specifying corresponding affinities.

3. From Models to Running Applications. The Architecture in a Nutshell.

We have implemented on top of Apache Struts a framework, named Cazon, 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 the user interface development according to the guidelines of the OOHDM abstract interface specification [13]. A high-level description of the architecture of the framework is depicted in Figure 5.

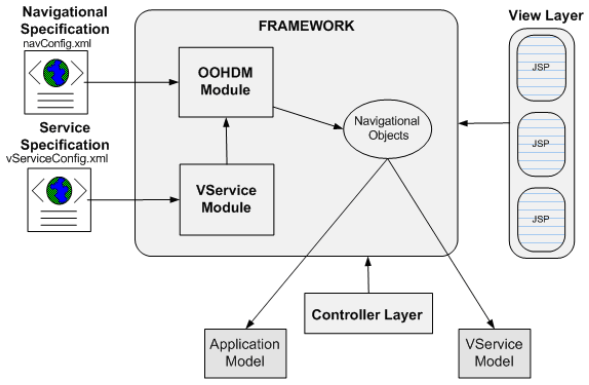

Figure 5. Architecture of Cazon

Cazon is a light-weight framework which 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

As shown in Figure 5 the two main components of Cazon are the OOHDM and the VService modules. The former deals with the definition of navigational constructors (basically nodes and links), and can be used with or without the latter. The VService module is in charge of augmenting the application nodes with volatile functionality in accordance with each node's properties.

3.1. OOHDM module

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. Cazon offers support for defining nodes that 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 a configuration file (NavConfig.xml) in which the designer specifies nodes, links and other navigation primitives. The information is transformed into navigational objects which constitute the navigational layer of the application.

3.2. Volatile Services Module

The volatile service module supports the integration of volatile functionality in a non-intrusive way by releasing the developer from re-factoring existing classes or configuration files. This module is in charge of the administratioon and and composition of 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. For each service, the developer must indicate the "home" service node; this is the node where the service (being activated from an application node) starts. In the case of an integration of type "Extension", this node will be inserted into those nodes which match the service affinity. If the integration kind is "Linkage", a link to the "home" node will be added to the selected nodes.

Node affinities are computed 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.

4. Cazon in action. An example

Cazon comprises a library of personalized tags which encapsulate navigation functionality and integration of volatile functionality into Struts JSP Pages. The most relevant tags are the following

- **node:attribute**: shows a node's attribute
- **node:agregatedList**: refers to an aggregation of nodes without using parameter or control structures in lists
- **node:aggregated**: inserts an aggregated node which can belong to a list
- **node:action**: performs an action specified in the actual node.

We have also defined a custom tag for adding volatile functionality: vservice:insert, which also supports variations according to the integration type.

As an example we next show a simple CD store portal with the kind of volatile functionality shown in previous sections.

4.1. Modeling the OOHDM structure

The OOHDM conceptual schema is mapped in a straightforward way onto a class schema (e.g. in Java); as (conceptual) volatile functionality is developed using well-known design patterns its transformation does not deserve further explanation; therefore we devote this section to the specification of navigational functionality, regarding both the core and volatile models.

In Figure 6 we show the configuration file specifying (part of) the navigational schema of our application, which is derived from the OOHDM navigational model. Notice that this XML file also shows the specification of the navigational objects which correspond to the volatile service.

Figure 7 shows the look and feel of the CDView page generated from the specification using Cazon custom tags.

```
<navigation-spec>
        .
        .
<node-class name="cdView">
   <page name="/pages/cdView.jsp"/>
   <conceptual-class>example.model.CD</conceptual-class>
   <attributes>
        <attribute name="title">subject.title</attribute>
        <attribute name="performer">subject.performer.name</attribute>
        <attribute name="cdFront">subject.front</attribute>
        <attribute name="listPrice">subject.listPrice</attribute>
        <attribute name="offerPrice">subject.offerPrice</attribute>
        <attribute name="rating">subject.rating</attribute>
        <attribute name="publisher">subject.publisher.name</attribute>
        <attribute name="editionDate">subject.formattedDate</attribute>

   </attributes>

   <actions>
        <action path="navigateToPerformerView"/>
   </actions>
</node-class>
        .
        .
<extension name="drawExtension">
        <attribute name="drawSpec">
                With the purchase of this product you participate in the Draw of tickets for
                the Rolling Stones concert.
        </attribute>
        <actions>
                <action path="navigateToConditions"/>
                <action path="navigateToResults"/>
        </actions>
</extension>
        .
        .
</navigation-spec>
```

Figure 6. Part of the navigational model

4.2. Modeling Volatile Services

In Figure 8 we show part of the servConfig.xml file which specifies the integration of the EMI draw volatile functionality into CDView nodes. Figure 9 shows the result of this integration.

5. Inside Cazon

In this section we describe the most important architectural decisions behind our framework. Our intent is to show those design features that might be useful in similar endeavors. For the sake of conciseness we devote to the highest level and most outstanding modules

Figure 7. CDView from the specification in Figure 6

```
<services-types>
        .
        .
        <service-type name="EmiDraw"
                    home="drawView"
                    integration-kind= "extension">
                        <affinity>
                                from cdView
                                where subject.recordCompany.name = 'EMI'
                        </affinity>
                        <extensions>
                                <extension name="drawExtension"/>
                                        .
                                        .
                        </extensions>
        </service-type>
          .
          .
</services-types>
```

Figure 8. The XML file for service integration

5.1. OOHDM Module and Controller

The OOHDM Module comprises:

- Core objects in charge of the life-cycle of navigational nodes.
- Mediation objects which interact with the Struts controller, intercepting the ordinary request flow, and communicating with Cazon's core objects to perform the navigation activity.
- A small set of custom tags to ease the presentation of information units inside a JSP.

The life cycle of an application derived from the framework begins when the modified Struts controller initializes it. At this moment, a customized plugin object creates a Cazon context to be used later by the application. A node factory is subsequently created to work with the navigational model (defined by the corresponding configuration file).

When the application receives a request to perform some action, the action's mapping is delegated to the Cazon request processor.

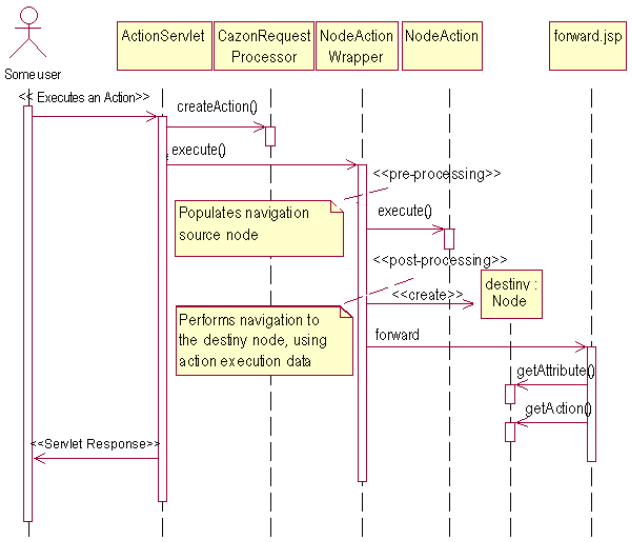

Figure 9. Request life cycle until the JSP is generated

The main task of this processor object is to override the way in which the default Struts processor creates actions. This implementation consists basically in wrapping the requested action when it involves a node. The wrapper object performs some activities with Cazon's OOHDM core objects before and after the action is executed. Before delegating the execute method, the wrapper populates the node information from which the request has been triggered. Once the execution is delegated to the wrapped action, the node factory is asked to provide the node that complies with the forward specification. Once we have the node, we include it as the current node to be used by the JSP, and proceeds with the normal Struts request flow. We show a detail of the request flow using a UML sequence diagram in Figure 10.

5.2. The Volatile Service Module

The implementation of this module is divided in two parts:

- An extension of the OOHDM Controller and a volatile services administrator.
- A sub-module responsible for calculating matches between services affinities and node surroundings.

5.2.1 The Controller Extension. This extension, called ServiceNodeFactory (SNF), acts as a decorator of the standard OOHDM node factory. It delegates all operations to the base factory except for the getNode() operation, which is the facade of the navigation behavior of the OOHDM Module. To perform this operation, the SNF collaborates with the OOHDM factory to obtain the target node and augment it with the corresponding volatile functionality. This latter process is performed by the volatile service manager, according with node surroundings and services affinities, as shown in the sequence diagram of Figure 10. Once the node has been augmented the request life cycle proceeds normally.

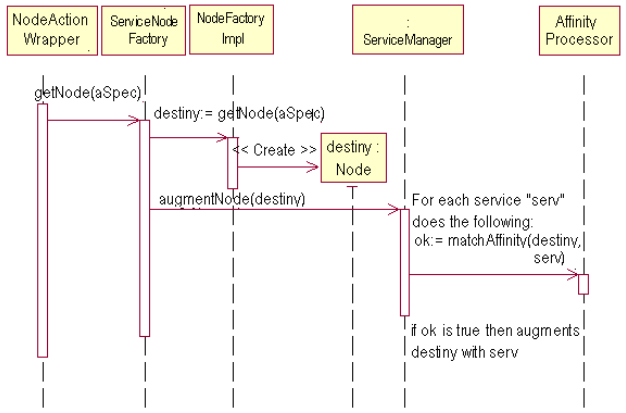

Figure 10. Interaction of objects in the VService module

5.2.2 The Affinity Processor The affinity processor analyzes an affinity query which is a part of a dynamic service specification, and decides whether or not to apply it on the current node. A class schema showing the main abstractions in the affinity processor is shown in Figure 11. Affinity queries are specified in a pseudo SQL query language. A typical query has the form:

FROM NODE-SPEC WHERE COND-SPEC

A node-spec may be a node class, a set of classes, the literal "*" to indicate that all classes are candidates to be enhanced with a service, or a nested query. The cond-spec admits all kind of comparisons, unary or binary logical expressions, and data transformations such as length (string) which are applied to attributes or relationships among classes in the application models.

The Affinity processor decomposes the query, generating an expression tree that represents the node set to which the affinity applies (node-spec) and the conditions which nodes have to satisfy (condition-spec). The queries (in particular, the node-spec) can be nested to describe more accurately the selected nodes. Figure 12 shows an example expression tree for one of our examples.

The affinity processor is activated after the NodeFactory has obtained the actual node that has to be presented to the user, and before its transformation into an HTML page (that will be sent as the response to the actual request).

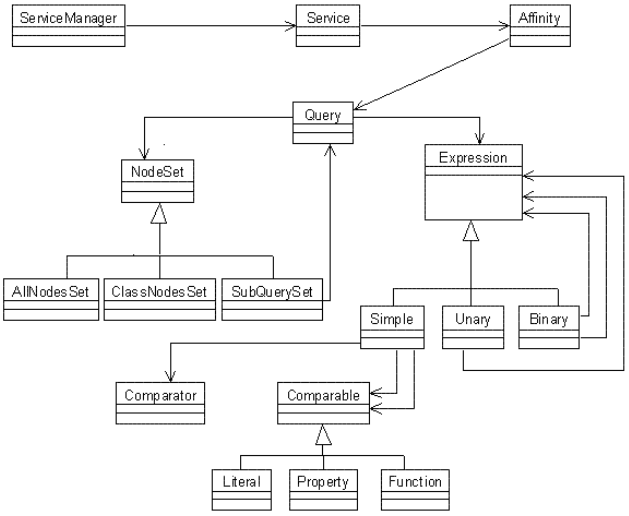

Figure 11. Diagram of AffinityProcessor collaborators

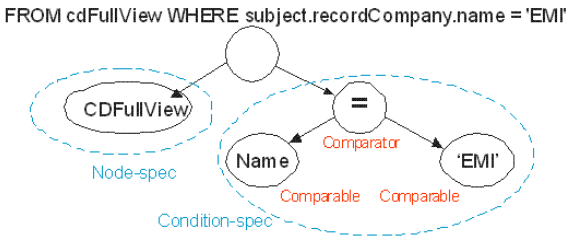

Figure 12. Example of a simple expression tree

The affinity processor is invoked from a Query object, along with the previously mentioned node. The Query first obtains (from its node-spec) the set of nodes to which it can apply the affinity. If the node is an element of the set defined by the node-spec, the affinity condition is evaluated on it, and if the evaluation is successful, the Service object communicates to the ServiceManager to apply the service to the actual node.

6. Refactoring Volatile Functionality

Volatile Services pose another challenge to the designer: what should be done when this kind of functionality becomes part of the business core? In other words, how should our design evolve according to this type of system evolution?

This software design problem has been recently analyzed in a broader sense [3]. Following an object-oriented perspective, refactoring means to re-design the base class abstractions and their relationships in order to incorporate abstract classes, introduce the use of appropriate design patterns, etc, to achieve better modularity in order to improve further maintenance.

While it is not one of the paper's objectives to discuss refactoring of volatile functionality in detail, there are certain issues to consider in order to address the problem. First, by clearly decoupling volatile functionality we aimed at reducing the need to refactor core classes when the functionality is removed. From this point of view, we consider that most functionality which has been labeled as "volatile" will be removed. However, as a (small) percentage of this functionality might be later considered as core functionality, we need a systematic strategy to integrate it seamlessly into the core. Taking into account that both the core and volatile models are specified using OOHDM, this is rather straightforward and can be easily systematized; we perform the refactoring process by applying the rules (which are just outlined) below:

- When volatile functionality follows a regular pattern, i.e. when all instances of a class must be affected/enriched with these additional features, we make commands (services) evolve into corresponding class methods. Additionally we refactor the affected node classes adding corresponding links (for Linkage integration) or attributes (for Extension ones).

- In the case of irregular functionality, e.g. functionality which is applied to specific instances of a class, as shown in previous examples, create a role type [14] supporting the extensions and make corresponding class objects (at the conceptual and navigational level) play the role. This can be done by defining a builder object [4] which decides (by checking object features) if the role will be instantiated and plugged onto the current class instance at the conceptual level, or instead it will define the corresponding JSP features at the navigational level.

7. Related Work

Volatile requirements have been recently dealt by the requirements engineering community; particularly in [9] the authors propose an aspect-oriented approach to model volatile concerns. Pattern Specifications are used to weave different concerns (core and volatile). Our approach focuses more on design than on requirements issues; instead of an aspect-oriented approach we use the concept of service affinity to weave functionality.

Well known Web Engineering approaches such as OOHDM, UWE and OOWS has early focused on separation of concerns to improve application evolution. None of these methods have already explicitly dealt with volatile functionality. In [1] the authors present an aspect-oriented approach for dealing with adaptivity. The concept of affinity could be easily introduced to mediate in the context of service integration of adaptive aspects weaving in UWE.

8. Concluding Remarks and Further Work

In this paper we have presented a model-driven 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. We have shown that volatile functionality poses several problems in the development cycle: introducing such functionality in the core model might clutter design abstractions with features that will be later removed; meanwhile, ignoring these aspects and treating them as code patches causes an impedance mismatch between design and implementation.

Our proposal implies the addition of a separate layer for modeling volatile functionality comprising both its conceptual and navigational aspects. To weave core and volatile models we use an integration specification which makes them oblivious with respect to the integration style. We use the concept of service affinity to determine the application pages which will be affected by the service. We have also presented an application framework which allows mapping our modeling constructs into a running application supporting volatile services. The framework acts as a light extension to the well known Apache Struts architecture. We are now pursuing several research directions: We are studying the implication of service inheritance and composition and analyzing the integration of external services (e.g. Web Services). We are also extending the affinity query language to support more complex specifications; we are extending our approach to the requirement stage by analyzing how volatile functionality is expressed using User Interaction Diagrams [5]. We are finally porting Cazon to other similar architectures like JSF and Shale [6].

9. 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] S. Ceri, P. Fraternali, and A. Bongio: "Web Modeling Language (WebML), A Modeling Language for Designing Web Sites". Computer Networks and ISDN Systems, 33(1-6), June (2000) 137-157

[3] M. Fowler: "Refactoring. Improving the design of existing code". Addison Wesley, 1999

[4] E. Gamma, R. Helm, R. Johnson, J. Vlissides: "Design Patterns. Elements of Reusable Object-Oriented Software", Addison Wesley 1995.

[5] N. Güell, D. Schwabe, P. Vilain. "Modeling Interactions and Navigation in Web Applications". ER (Workshops) 2000. (Utah,USA, October 2000) 115-127

[6] Java Server Faces Home page. In www.jsfcentral.com/

[7] W. Kim, "Advanced Database systems", ACM Press, 1994.

[8] N.Koch, A. Kraus, A., and R. Hennicker: "The Authoring Process of UML-based Web Engineering Approach", Proceedings of the 1st International Workshop on Web-Oriented Software Construction (IWWOST 02), Valencia, Spain (2001) 105-119

[9] A. Moreira, J. Araujo, J. Whittle: "Modeling Volatile Concerns as Aspects", to be presented in CAiSE 2006, Luxemburg, June 2006.

[10] OMG Model-Driven-Architecture. In http://www.omg.org/mda/

[11] 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.

[12] O. Pastor, S. Abrahão, J. Fons: "An Object-Oriented Approach to Automate Web Applications Development", Proceedings of EC-Web, 2001: 16-28.

[13] 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 nr.4, October, 1998, 207-225.

[14] F. Steimann: "On the Representation of Roles in Object-Oriented and Conceptual modeling". Data and Knowledge Engineering 35 (2000) 83-106