Journal of Web Engineering, Vol. 8, No.1 (2009) 025-047
© Rinton Press

OBLIVIOUS INTEGRATION OF VOLATILE FUNCTIONALITY IN WEB APPLICATION INTERFACES

JERONIMO GINZBURG

Departamento de Computación, FCEyN, Universidad de Buenos Aires, Argentina
jginzbur@dc.uba.ar

DAMIANO DISTANTE

Faculty of Economics,Tel.M.A. University, Rome, Italy
distante@unitelma.it

GUSTAVO ROSSI MATIAS URBIETA

LIFIA, Facultad de Informática, UNLP, La Plata, Argentina and Conicet
{gustavo, matias.urbieta}@lifia.info.unlp.edu.ar

Received November 28, 2007
Revised June 19, 2008

Web applications are used to fast and continuous evolution. In response to new or changing requirements, additional code is developed and existing one is properly modified. When new requirements are temporary, i.e., when they specify some volatile functionality that is expected to be online only for some time and then removed, the additions and changes are destined to be later rolled back. This way to proceed, apart from being time and effort demanding, by involving the intrusive editing of the application's source code, brings along the risk of polluting it and introducing mistakes. In this paper, we present an approach to deal with volatile functionality in Web applications at the presentation level, based on oblivious composition of Web user interfaces. Our approach, which is inspired by well-known techniques for advanced separation of concerns such as aspect-oriented software design, allows to clearly separate the design of the application's core user interface from the one corresponding to more volatile functionality. Both core and volatile user interfaces are oblivious from each other and can be seamlessly composed using a transformation language. We show that in this way we simplify the application's evolution by preventing intrusive edition of the user interface code. Using some illustrative examples, we focus both on design and implementation issues, presenting an extension of the OOHDM design model that supports modular design of volatile functionality.

Keywords: Web engineering, volatile functionality, user interfaces, separation of concerns, OOHDM.

1 Introduction

Most Web applications must deal with a myriad of requirements which usually belong to different application themes. While obtaining a modular design model and mapping this model onto a proper implementation is itself a big problem, the evolving nature of Web applications' requirements makes things much more difficult.

Research in the software engineering arena has shown that by grouping sets of requirements belonging to the same theme or interest in modules called concerns, by clearly decoupling these concerns in each development stage, and by using proper mechanisms to weave (i.e., compose) corresponding design and implementation artifacts, we can get more evolvable (Web) software [22]. While there are still discussions regarding which is the “best” approach to modularize requirements, there is already a general agreement about the nature of the problem and the fact that no single approach to modularization (e.g., classes) can solve the problem by itself. As a consequence, different techniques for advanced separation of concerns, such as architectural and design patterns [19] and aspects [18], have been already introduced in the Web field (see for example [6]); however, there are still many open problems, related to concern separation and composition, which have not been fully addressed.

In this paper we focus on a particular kind of application requirements, those typically arising after a Web application has been deployed and that are valid for short periods of time after which they are discarded; we refer to this kind of requirements as to *volatile requirements* and we call the temporary functionality they originate *volatile functionality* (or *volatile services*). In particular, we concentrate on the design of volatile functionality and its impact on presentation issues.

In a Web application there may be many different kinds of volatile requirements giving rise to volatile functionality. Some of them may arise during the application’s evolution to check acceptability of the users’ community (such as, beta functionality) and might be later considered or not core application services (such as, forums and users’ tags in Amazon.com). Others are known to be available only on short and determined periods of time, such as functionality for specific holidays (e.g., St. Valentine), or sales for fixed periods of time (e.g., Christmas). Others are even more irregular: some kinds of price discounts in e-stores (e.g., for specific leftover stock products), draws and concert tickets selling (associated with a CD), Amazon’s “Related Video” for new releases from an artist, etc.

Volatile functionality may also affect irregular subsets of objects; for example, only some CDs in a store are involved in a draw, artists’ performances in a video appear only in some novelties, etc. As an example, in Figure 1 we show the Amazon page of the last CD of Norah Jones featuring a short video of the artist, which may be removed after some weeks. Additionally, at the bottom of the page, there is a link to a St. Valentine’s store, a volatile sub-store which was removed after St. Valentine’s day.

Volatile functionality poses many challenges to designers and design methods; these challenges manifest both at modeling and implementation time. Suppose, for example, that we have designed a CD class or entity type using any Web design method. If we want that some CD objects exhibit some new behavior or data (e.g., a video performance of the artist), we can either add an attribute to the class, create a subclass for the new behavior, redesign the class so that it is now a composite class encompassing as a component the new features, etc. Some of these solutions might be better than others, depending on the context (e.g., sub-classification or class editing are not good solutions to deal with functionality attached only to certain instances). However, in all cases, we face a serious problem: being this functionality volatile, we might need to deactivate it after a certain time. This means, once again, some kind of intrusive editing in the application’s source code or model, which is always cumbersome; for example, if we are using a model driven approach, the model still has to be edited and this operation is certainly error prone. To make matters worse, if additional requirements arise

after introducing volatile modules, we might have a more complex tangle of core and volatile design components.

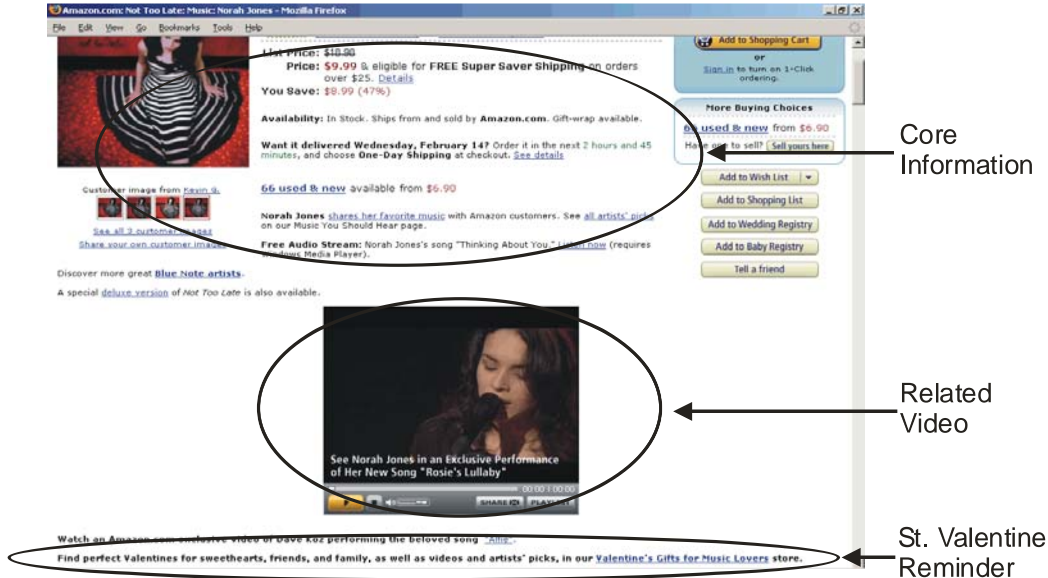

Figure 1. Different volatile functionality in Amazon.com.

As extensively discussed in the literature (see [18] for example), code and design tangling (i.e., the implementation of two or more concerns in the same module) is a source of maintenance problems. Even using powerful configuration management environments, rolling back to the desired application's configuration might be a nightmare. Notice that the same situation arises with the volatile link to the St. Valentine store in Figure 1: in this case the problem is reflected in the navigational model in which we have to add and later delete a link to the store. Additionally, the same problem is propagated to the user interface, as we need to modify it accordingly to the changes we made in the navigational model. If we have used templates for specifying both interfaces (the core and the volatile), we need a way to seamlessly merge them. This problem manifests itself independently of the technology we use, as we show in Section 3.1

Unfortunately, the armory of existing Web design methods (object oriented ones, like OOHDM [37], UWE [25] and OOWS [32], or those based on data modeling approaches, like UWA [40] and WebML [9]) lacks modeling primitives to deal with this problem.

We have been working on design issues related to volatile functionality for some time. In [34] we presented an extension of the OOHDM approach to deal with volatile functionality, both at the conceptual and navigational level. Following our approach, we model volatile features as first class entities (e.g., classes), which are completely decoupled from core classes. Core and volatile classes are then "weaved" together at run-time, by using an integration specification and an integration engine.

In this paper we focus on the problem of tangled design and code at the presentation level; we describe how we extended our ideas to the user interface realm, showing that it is also possible to

create concern-specific interfaces that are obliviously composed according to integration specifications. Obliviousness (in our context defined as the lack of explicit knowledge relationships between composed interface components) has been found as a key property to diminish evolution problems [18].

To make the paper self-contained, we present a brief summary of our approach to introduce volatile requirements at the conceptual and navigational layers.

The main contributions of this paper are the following:

- We show with simple examples that volatile functionality represents a serious problem at the user interface level.
- We present a modular and scalable solution for improving interface design that can be used for any kind of design concern, not only volatile.
- We present an implementation of the solution we propose using the OOHDM Web design method and a supporting framework named CAZON.

The rest of the paper is structured as follows. In Section 2 we present the background of our work and provide an overall view of our approach for the integration of volatile functionality in Web applications, with reference to the OOHDM method. In Section 3 we focus on the presentation layer: we first present an extension of the Abstract Data Views (ADV) interface design notation [13] used in OOHDM to specify abstract Web interfaces; then we introduce the idea of interface composition using transformations and illustrate our implementation with the CAZON framework. We discuss the benefits of our approach in Section 4 and some related work in Section 5. Finally, in Section 6, we conclude the paper and present some further work we are pursuing.

2 Background and Overall View of the Approach

2.1 OOHDM and CAZON in a Nutshell

OOHDM, similarly to other Web development approaches such as UWA [40], OOWS [32], UWE [25], or WebML [9], organizes the process of development of a Web application into five activities: requirements gathering, conceptual (or content) design, navigational design, abstract interface (or presentation) design and implementation. During each activity a set of object-oriented models describing particular design concerns are built or enriched from previous iterations.

The first activity is intended to collect and analyze the stakeholders' requirements. Use Cases [23] and User Interaction Diagrams [21] can be used for this purpose.

After gathering requirements, a set of guidelines is followed in order to define from them the conceptual (or content) model of the application. During this phase of conceptual design, a model of the application domain is built using well-known object-oriented modeling principles. The conceptual model may not reflect the fact that the application will be implemented in a Web environment, since the key Web application model will be built during navigational design.

The navigational structure of the Web application is defined by a schema containing navigational classes. OOHDM offers a set of predefined types of navigational classes (navigation design

primitives), i.e., nodes, links, anchors and access structures, many of which cannot be directly derived from conceptual relationships. By using a viewing mechanism, classes in the conceptual model are mapped to nodes in the navigational model while relationships are used to define links among nodes.

The last design phase prior to implementation is the abstract interface design. In this design phase, the user interface of the application is specified by means of Abstract Data Views (ADVs) [13] which support an object-oriented approach for interface design. In OOHDM we define an ADV for each node class, indicating how each node's attribute or sub-node (if it is a composite node) will be perceived by the user. An ADV can be seen as an Observer [19] of the node, expressing the node's perception properties as nested ADVs or primitive types (e.g., buttons). A configuration diagram [37] is used to express how these properties relate with the node's attributes. ADVs are also used to indicate how interaction will proceed and which interface effects take place as a result of user-generated events. These behavioral aspects are specified using ADV-charts [13] but we will not go more in detail on them as they are out of the scope of this paper that is focused on structural interface aspects.

During the implementation activity, conceptual, navigation and interface objects are mapped onto the particular runtime environment being targeted. In order to foster the adoption of the OOHDM approach, we have implemented a framework, named CAZON, on top of Apache Struts [2], which supports the semi-automatic generation of code from OOHDM models, including the instantiation of Web pages from OOHDM navigational models. The navigation module of CAZON receives the navigational model in the form of an XML configuration file that specifies nodes, links and other navigation primitives. The information is processed and transformed into navigational objects that constitute the navigation layer of the application. For the presentation layer CAZON uses Stxx [38], an extension of Struts that allows action classes to return XML data documents to be transformed using different technologies into appropriated presentation formats, such as HTML, WAP or PDF. JSP can also be used to implement user interface objects.

As we will explain later in this paper, the CAZON framework also provides support for dynamic and seamless integration of volatile functionality into implemented Web applications.

2.2 Extending the OOHDM Approach to Integrate Volatile Functionality

Our approach to deal with volatile functionality in Web applications is based on the idea that even the simplest volatile functionality (e.g., the temporary addition of a video to a page, as in Figure 1), must be considered a first-class functionality and, as such, designed accordingly. At the same time, volatile functionality must be kept separated from core functionality and integrated in the application in a seamless way, in order to be easily removed when required.

To put into practice the ideas above stated, we have extended the OOHDM Web applications development method with the following solutions:

- We decouple volatile from core functionality by introducing a modeling layer specific for volatile functionality (named Volatile Layer), which comprises a conceptual, a navigational, and an interface model for each volatile concern.
- New behaviors, i.e., those belonging to the volatile functionality layer, are modeled in the volatile conceptual model; they are considered as a combination of Commands and Decorators [19] of the core classes.

- Therefore, we use inversion of control (IOC) to achieve obliviousness; i.e., instead of making core conceptual classes aware of the new features, we invert the knowledge relationship. New classes know the base classes on top of which they are built. Core classes, therefore, have no knowledge about the additions.
- Nodes and links belonging to the volatile navigational model may or may not have links to the core navigational model. The core navigational model is also oblivious to the volatile navigational classes, i.e., there are no links or other references from the core to the volatile layer.
- We use a separate integration specification to specify the connection between core and volatile functionality called "affinity" [31]. In this specification, we indicate which node instances are affected by a volatile functionality and the way in which the navigational model will be extended (e.g., by adding a link, inserting new information in a node, etc.). The set of nodes affected by the volatile functionality represents the affinity of the new functionality.
- We design (and implement) the interfaces corresponding to each functionality separately; the interface design of the core classes (described using ADVs [13]) are oblivious with respect to the interface of volatile functionality. Core and volatile interfaces (at the ADV and implementation levels) are woven by executing an integration specification, which is realized using XSL transformations at runtime. This specification indicates the bi-dimensional pointcut of the core user interface where the new elements have to be inserted.

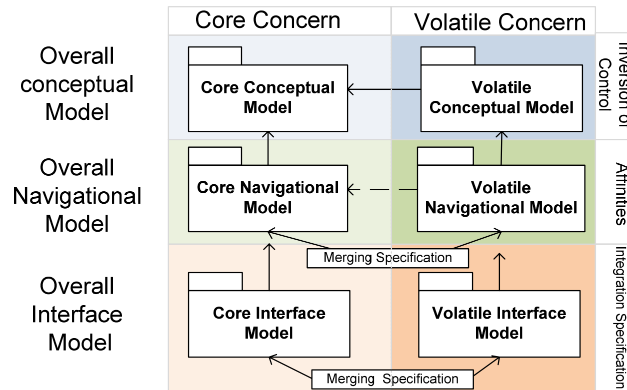

Figure 2. A schematic representation of our approach to dealing with volatile functionality in Web applications.

Figure 2 schematically represents our approach by highlighting, in particular, the core and volatile modeling layers, the models included in each layer, the relationships among them, and the solutions to seamless integrate the modeled volatile functionality into the core application.

Table 1 summarizes our solutions to modeling and weaving volatile functionality in Web applications throughout the development process. Instead of using a unified modeling and weaving mechanism (e.g., aspects) in the three modeling levels, we devise more customized approaches according to each model's feature.

Table 1. Summary of solutions adopted for modeling, weaving and implementing volatile functionality.

		Solution for			
		Modeling	**Weaving Method**	**Concern Implementation**	**Concern Weaving Implementation ***
Layer	**Conceptual**	UML class Diagram	IOC	Commands and Decorators	---
	Navigational	OOHDM Navigation Diagram	Navigational Affinities	Navigational Nodes	Navigational Affinities (XML configuration file)
	Interface	ADV and ADV-Chart	ADV Integration Specification	XML-based Documents	XSL Transformations

* Each technology has its own runtime weaving engine

The conceptual model in OOHDM is a pure object-oriented behavioral model. Therefore we chose not to change its basic semantics but to use the preferred solution for seamlessly adding new behaviors to a class: object's decoration [19], which relies on inversion of control.

Next, the navigational model is expressed using a more structural object model in which objects' structure is defined opportunistically (i.e., not using the same design guidelines as in more "pure" object-oriented approaches). This means that the attributes and behaviors provided by nodes are defined according to usability needs, more than design modularity constraints. Besides, we aimed to enrich the nodes' structures dynamically. As a consequence we devised an approach based on the concept of affinity, which has been originally conceived in the field of structural computing [31]. Our approach allows us to add new attributes or services to the volatile functionality affinity at run-time.

Finally, the interface specification is defined using ADVs which, even being object-oriented artifacts, describe some interface's peculiarities (such as its "two dimensional" arrangement); as a consequence, a conventional aspect-oriented solution, or even other advanced object-oriented solutions are unable to cover the need for expressing weaving in two dimensions. We will further elaborate on this in Section 3.

By focusing on volatile services since the early stages of the application's development, we aim to address the impact of crosscutting concerns in design models in order to improve traceability of design decisions instead of facing this problem at coding stage, where any radical change would cost lot of effort.

It is worth to note that though we present our approach with reference to the OOHDM design method, the solutions we propose are applicable to any other design method that, similarly to OOHDM, partitions the design space of a Web application in the three layers represented in Figure 2: conceptual (or content), navigational and interface (or presentation) layer.

We take advantage of runtime weaving and enable the application to be configured online (enabling/disabling concerns) without shutting it down, re-weaving it and finally starting it up again, as it should be done using weaving at deployment time.

Table 1 shows the solutions we adopt for modeling, weaving and implementing at runtime volatile functionality, for each of the three layers mentioned above.

2.3 Integrating Volatile Functionality into Conceptual and Navigation Layers

As depicted in Figure 2, when the approach described in Section 2.2 is adopted, the overall architecture of the application consists of a package for the core model and a package for each of the volatile concerns. They represent the core and volatile layers, respectively. Inside these packages, there are sub packages for the conceptual, navigational and presentation layer of each concern.

The overall conceptual model of the application is therefore comprised of core and volatile models, which are realized in the conceptual sub packages of each layer. The core conceptual model sub packages contain classes related to the core functionality. The volatile conceptual model sub packages are composed of command classes and additional classes that give support to the new functionality. Each command class encompasses a new (volatile) functionality, following the Command pattern [19]. Moreover, these classes have a knowledge relationship with the core classes, while these ones remain oblivious with the commands.

Classes in the volatile layer have a knowledge relationship to classes being extended with the new services in the core layer.

The overall navigational model is built analogously to the conceptual model. Each concern (core or volatile) contains a sub package for its navigational model. There are no links between core and volatile nodes (indicating obliviousness); however there might be links from the volatile to the core navigational model (specified with a dashed line in the diagram of Figure 2) indicating a possible navigation path to existing nodes. Nodes comprising volatile functionality are woven into the core navigation model using an integration specification which is decoupled both from volatile and core conceptual model classes, thus making the nodes oblivious to the integration strategy. In this way, the same volatile functionality can be attached to different nodes at different times, according to the application's needs. The specification indicates the nodes that will be enhanced with the volatile functionality, and the way in which the navigation model will be extended. For example, we can add links, insert new information or components in a node, etc. Nodes affected by the new functionality are called *affinity* of the functionality. We specify affinities with the same query language used in OOHDM to specify nodes [24, 37], which is based on object queries. Nodes matching the query are those that will be affected by the volatile service.

A query has the form: *FROM* $C_1 \ldots C_i$ *WHERE Predicate*, in which $C_1 \ldots C_j$ indicate node classes and the *Predicate* is defined in terms of properties of model objects. A query expression also indicates the kind of integration between nodes and services, which can be *Extension* or *Linkage*. An *extension* indicates that the node is enhanced to contain the volatile service information (and operations) while, a *linkage* indicates that the node does not contain additional operations; it "just" allows navigation to the service and does not support new behaviors. In the case of *linkage* service, we can also specify additional features such as attributes or anchors that have to be added to the affected node to clarify the linking.

In Figure 3 we show the core and volatile conceptual and navigational models for the CD example mentioned in Section 1; the intent of this figure is to serve as an anchor for further examples in Section 3. More elaborated examples of volatile functionality and navigational integration specifications can be found in [34].

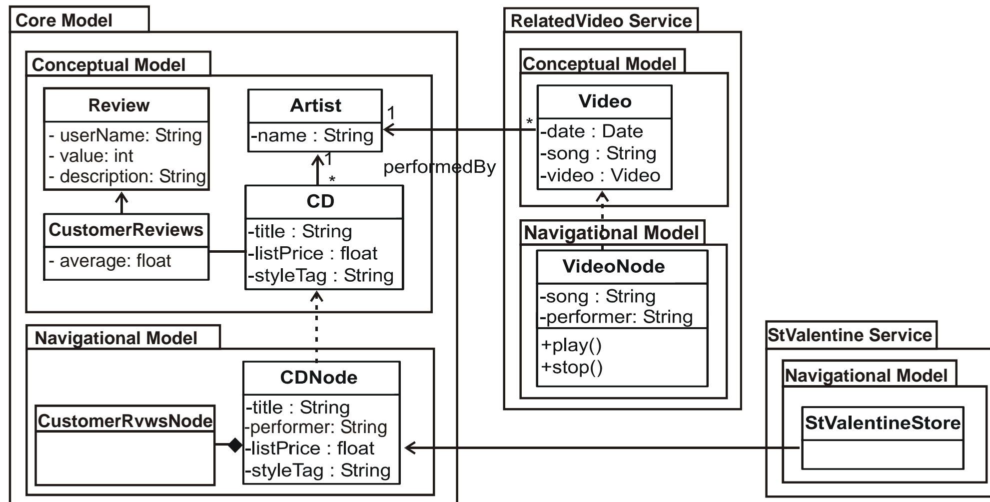

Figure 3. Core and Volatile Conceptual and Navigational Models.

The navigational integration specification for the video volatile functionality is the following:

Affinity NorahJones (FROM VideoNode WHERE performer = ‘Norah Jones’)
FROM CDNode WHERE title = 'Not too Late' AND performer = 'Norah Jones'
Integration: Extension

This specification indicates that the nodes corresponding to the CD with title “Not too late” and performed by “Norah Jones” will be extended with the Norah Jones video (the VideoNode which fulfills the specification), as if new attributes where added. Meanwhile, the specification for the volatile St. Valentine’s store is as follows:

Affinity Valentine
FROM CDNode and BookNode WHERE styleTag = “Romantic”
Integration: Linkage (StValentineStore)
Additions: [Message: Text (“Find perfect Valentines for sweethearts....”)
ToStore: Anchor]

In this case, we wish to add a link towards the singleton StValentineStore to those products (CDs and Books) which are tagged as “Romantic”. The text message to be shown is indicated in the specification and the anchor is associated to the link created at weaving time. Notice that in both cases the interface specification and, therefore, some aspects of the implementation should also change to show the volatile node/link.

Our notation allows expressing different affinities for the same volatile concern, therefore allowing high flexibility in the resulting nodes without polluting core classes. For example, once the CD is no more a novelty, we can weave the video to another node, if necessary, with a different specification; in Amazon, particularly, some of these videos are relocated (during a period of time) in the home page.

In CAZON, the integration between core and volatile services at the navigational level is performed by executing the queries which specify the service affinities. A service manager evaluates affinity queries (stored in XML files) on the node being built, and when a query succeeds, it augments

the node (in fact, the corresponding XML description) with the attributes, links or aggregated nodes indicated in the specification. As a result of a request, CAZON returns a node which now contains the corresponding volatile functionality and whose presentation is handled using the base Struts mechanisms and tools. A full description of CAZON can be found in [34].

In Section 5 we discuss the similarities and differences between this solution and an aspect-oriented one (e.g, using AspectJ [4]) both in the conceptual and navigational models; as a summary, while using an aspect language in the conceptual model does not change the spirit of our approach, the main disadvantage of using AspectJ in the process of "building" nodes is that it works at deployment time, while our solution allows a more dynamic, i.e. run-time, addition of services by editing the queries specification and without requiring re-compilation. Additionally, AspectJ can not be used on user interface tags.

In the following sections, we focus on the presentation layer and explain how we managed to apply the ideas of oblivious composition of volatile functionality in the user interface layer of a Web application, both at the design and implementation stage. Though we focus on our implementation with CAZON, most of the concepts can be easily applied in a broader context. Particularly, the idea of separation and oblivious composition of user interfaces is applicable to all kinds of design concerns, though we exemplify it with volatile ones.

3 Dealing with Volatile Functionality in Web Interfaces

3.1 Motivation

As we previously mentioned, user interfaces also suffer the impact of the addition and editing of volatile functionality both at design and implementation levels. Even if we push languages like JSP to their limits regarding modularity, it is practically inevitable that the code describing the interface of core components is polluted with tags belonging to volatile functionality and therefore that both concerns (core and volatile) get tangled.

As an example, the JSP page that implements the CD interface showed in Figure 1 will have knowledge of both the St. Valentine store and the Related Video component. Figure 4 shows a portion of this JSP page. In it, we can identify two blocks of code which refer to the Related Video and the St. Valentine concerns and which are clearly tangled with the code of the core (CD) concern, therefore compromising modularity. A similar situation would arise at the interface design level, regardless of the used notation. The classes corresponding to the core interface will have explicit references (either as attributes or as aggregated classes) to the volatile ones.

It is important to state that this problem is conceptual and not technological. In common implementation technologies (e.g., JSP, JSF, etc.) or user interface description languages (e.g., usiXML [28], UIML [33], etc.) we may experience this kind of tangling due to the lack of primitive constructs to implement either a solution based on polymorphism (e.g., using Decorators to make oblivious interface code insertion) or on pointcuts such as in aspect-orientation. The *include* primitive, typical of scripting languages, does not guarantee obliviousness because it yields an explicit invocation.

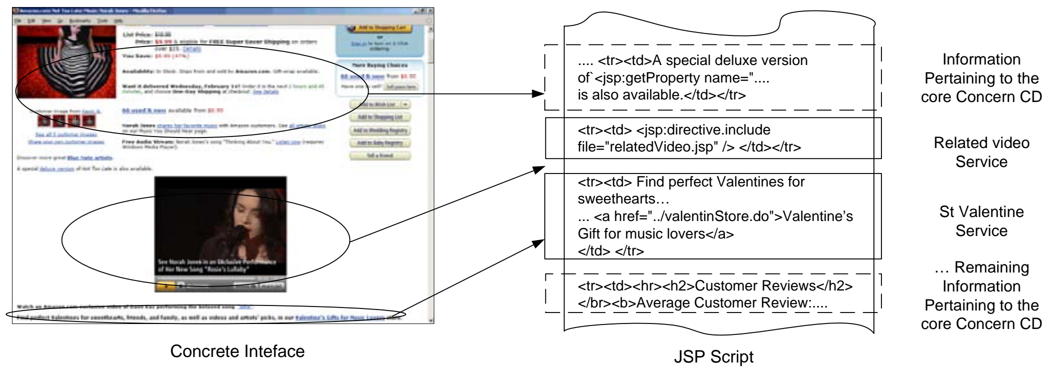

Figure 4. Code tangling in the user interface.

It should not be surprising that separation of concerns is such elusive at the interface level; most modern Web engineering techniques have treated these aspects as lower-level issues (e.g., relegating it to the implementation stage). Avoiding tangling at the interface level allows better composition of existing interfaces and interface designs. Though this impact is obvious and seems more harmful at the code level (e.g., in a JSP page) it should be also addressed at design time. For the sake of understanding, we describe our approach in two different sub-sections, addressing design issues first, and then showing how we realized these ideas in the implementation stage in the context of CAZON.

3.2. Composing Web User Interface Design Models

As we have explained in Section 2.1, in OOHDM the user interface is specified using ADV. We have slightly modified the ADV notation in such a way that the position of nested objects in the ADV reflects the look and feel of the user interface. Figure 5 shows an example of this. This notation, which is inspired by a similar one for UWE [25], allows improving discussions with different stakeholders, though it cannot be processed automatically by standard ADV-based tools.

Each concern (core and volatile) will comprise ADVs for its corresponding nodes; when necessary, e.g., when a node should exhibit some volatile functionality, we weave volatile and core ADVs using an integration specification. Figure 6 shows these ideas schematically.

To express the integration, we have defined a simple specification language that allows indicating pointcuts and insertions at the abstract interface level, i.e., the position of the volatile ADV when it is inserted in the core ADV. The specification generalizes the idea of pointcuts to the two dimensional space of Web interfaces. A pointcut and the corresponding insertion are specified using the following template:

Integration: *Integration name*
Target: *ADV target name*
Add: *ADV source name | Insertion Specification*
Relative to: *ADV name*
Position: [*above | bottom | left | right*]

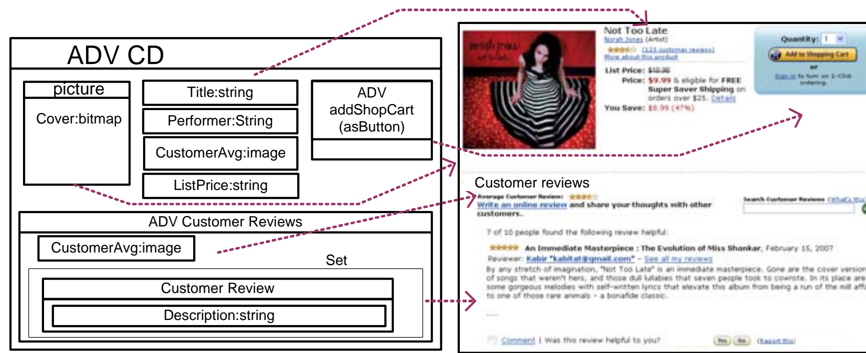

Figure 5. The ADV corresponding to the CD Node.

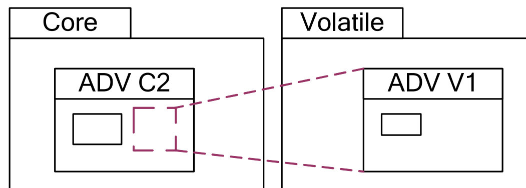

Figure 6. ADV V1 woven into ADV C2.

The *Integration* field is a name identifying the integration specification. The *Target* field indicates the name(s) of the ADV(s) that will host the volatile interface code. Inner ADVs may be specified using a "." notation, such as CD.Reviews, to indicate that the insertion will take place in the ADV Reviews, which is a part of the ADV CD. The *Add* field indicates which elements, either an ADV or an immediate specification, must be inserted in the target ADV(s). Immediate specification is an anonymous definition that is used when the inserted field is simple enough to avoid the reference to another (auxiliary) ADV. Finally we indicate the insertion position by using the *Relative to* and *Position* fields. The relative field points out some comprised component of the target ADV. The position field sets the relative position taking into account as a pivot the target ADV component specified in the "relative to" field.

Notice that the specification defined according to the above described template is still abstract, thus leaving space to fine tuning during implementation.

In Figure 7.a we show the ADV for the Related Video volatile functionality and the integration specification that corresponds to the abstract interface of Figure 5. The result of the weaving process in the abstract interface is shown in Figure 7.b. As shown, the ADV Related Video is placed between the CD's core information and the inner ADV Customer Review. The final concrete interface, which comprises both aspects, is shown in Figure 7.c.

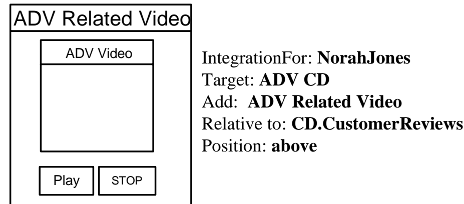

Figure 7.a The ADV and the integration specification for the Related Video volatile functionality.

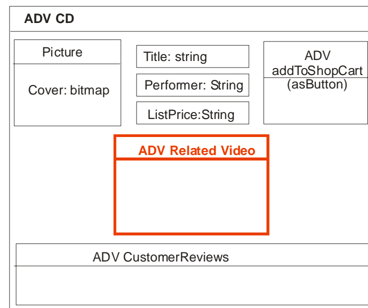

Figure 7.b. Expected ADV structure after weaving.

Figure 7.c. Resulting concrete interface.

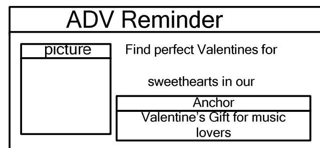

Figure 8.a. ADV Reminder.

IntegrationFor: **St Valentine**
Target: **ADV CD**
Add: **ADV Reminder**
Relative to: **ADV CD.CustomerReviews**
Position: **above**

Figure 8.b. Integration specification for ADV Reminder.

Figure. 9. ADV CD after weaving and the resulting concrete interface.

Sometimes the integration requires additional interface objects, for example when the navigation extension is of type *linkage*, as in the case of the St. Valentine store. The corresponding objects might be defined either as ADVs belonging to the specific volatile concern package (e.g., for reusing them in

other specifications), indicated in the integration specification (e.g., when they are simply strings), or separately defined as "integrators" ADVs (e.g., when only used for this particular specification).

In Figure 8.a we show the ADV Reminder that is used during the integration process of the St. Valentine's store. The result of weaving the ADV Reminder into the ADV CD according to the specification in Figure 8.b gives as a result the ADV and the concrete interface shown in Figure 9.

The sequence of the compositions is important. If we first perform the weaving of the Related Video ADV with the CD ADV using the integration specification of Figure 7.a and, after that, the weaving with the St. Valentine's Reminder ADV using the integration specification of Figure 8.b, we will get as a result an ADV that represents the concrete interface of Figure 1, with the St. Valentine's Reminder below the Related Video component.

At the same time, as both core and volatile components are modeled using standard OOHDM primitives, we can also define pointcuts over volatile targets. These kinds of compositions, however, must be done carefully because volatile targets may not be available at the moment of integration due to its inherent volatile property. The problem which arises from the ordering of weaving is similar to the "aspect conflicts" problem which is being researched by the aspect-oriented community [17]. A further discussion on this issue is outside the scope of this paper.

It is worth noticing that, as described in [37], ADVs can be mapped systematically onto concrete interface specifications in different running environments.

Next, we show our approach to achieve obliviousness of interface code in the implementation stage.

3.3 Using Transformations to Compose XML Documents

The problem of achieving obliviousness between core and volatile code at the user interface layer can be expressed in terms of XML documents as follows: given two documents A and B which express the original and additional content of a node, respectively, we need to describe how to obtain a document that integrates B into (a specific part of) A, without an explicit reference to B inside A. Moreover, in the case of (irregular) volatile functionality, we need that this integration is done in all documents that fulfill some conditions; this might eventually involve specific instances of different document types.

The core of our solution is to use XSL transformations [43] to compose volatile and core interfaces, and XPATH [42] to indicate the parts of the source document in which the insertions are done. The transformation acts as an aspect in aspect-orientation: the content of a template is like an advice, while the XPATH specification that matches the template indicates the pointcut where the advice is inserted. An XSL engine (e.g., Xalan [41], Saxon [35]) performs the weaving process.

For example, in order to add the Related Video component to the CD Node, we can apply the XSL transformation shown in Figure 10 over the CD interface. The XPATH expression (pointcut) "//tr[contains(.,'Customer Reviews')]" refers to the HTML table row containing the text "Customer Reviews" (see Figure 4). The template (advice) leaves the existing elements of that table row unchanged and inserts above it a new one with the Related Video element.

In the case of class-based volatile functionality (e.g., functionality which applies to all instances of a class), and given that JSP pages can be written as well-formed XML documents, the XSL

transformations could be applied statically to incorporate the tags with volatile functionality without polluting the source code, as represented in Figure 11.

```
<xsl:stylesheet version="1.0" xmlns:xsl="http://...." xmlns:jsp="http://...." >
   <!--Imports a transformation which copies all the elements-->          Pointcut
   <xsl:import href="defaultTemplate.xsl"/>
   <xsl:template match="//tr[contains(.,'Customer Reviews')]">   Advice

      <tr><td> <jsp:directive.include file="relatedVideo.jsp" /></td></tr>
      <tr><xsl:apply-templates select="*"/></tr>

   </xsl:template>
</xsl:stylesheet>
```

Figure 10. Transformation that inserts Related Video component.

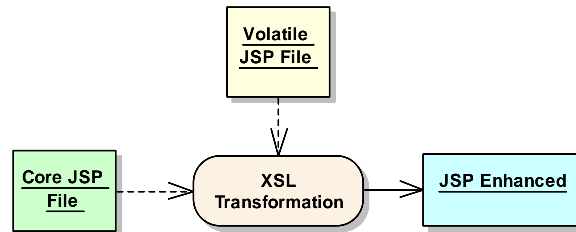

Figure 11. Statically weaving of volatile interfaces.

A similar solution cannot be applied when volatile functionality only affects some instances of a class (e.g., those products that are recommended to be St. Valentine's gifts): in this case, in fact, some kind of conditional structure should be included in the tags, thus polluting the resulting code. As applying XSLT transformations in bare JSP at run time is cumbersome, we decided to use a flexible approach using a more "pure" XML-based framework in which the publishing process is done by applying XSL style sheets to the XML content. We describe our approach in the following sub-section.

3.4 Supporting Interface Composition with CAZON

In CAZON, for each node type we define a style sheet that transforms the XML representation of the node in a physical presentation object (e.g., HTML, WAP, etc.). When the node contains aggregated sub-nodes, the style sheet contains calls to the style sheet templates corresponding to the nested parts.

As explained before, when the framework receives a request to perform an action, it produces as a result a node instance described with an XML document containing its attributes, anchors and its recursively aggregated nodes. The presentation layer gets this document and transforms it according to the corresponding style sheet. As an example, in Figure 12 (left side) we show the style sheet associated to the CDNode type; when applied to the XML node representation corresponding to the

Norah Jones CD (Figure 12, right side) it yields a concrete interface, as the one in Figure 9, but without the reference to the St. Valentine's store.

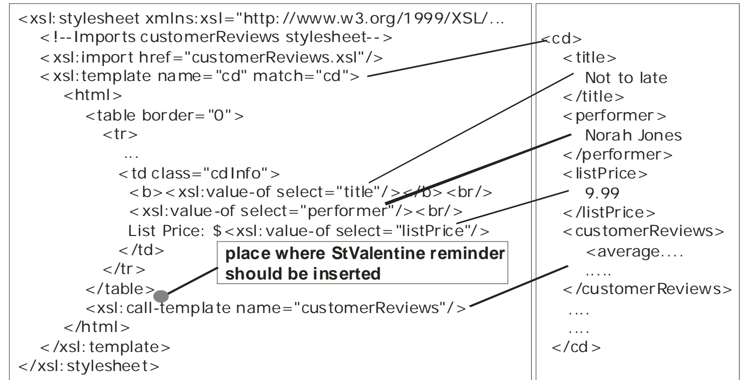

Figure 12. CD style sheet and CDNode instance.

As explained in Section 2, when a node instance satisfies an affinity query corresponding to a volatile service, the corresponding node (in fact its XML representation) is augmented according to the integration specification, either with a link, attributes or nested nodes. Therefore, to complete the task in the user interface and according to Section 3.2, for each integration specification we need to implement an XSL transformation that, applied over the style sheet corresponding to the core node, inserts the newly added elements. The transformation associated with the St. Valentine's Reminder integration specification of Figure 8.b is shown in Figure 13.

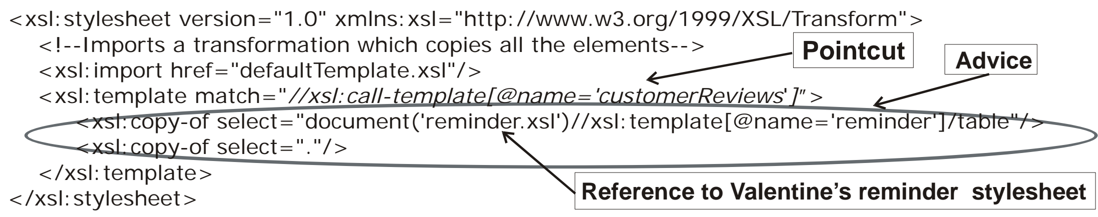

Figure 13. Transformation that inserts St. Valentine's reminder.

In order to allow dynamic weaving of style sheets during run time in CAZON, we created a subclass of the Stxx AbstractXSLTransformer such that the method transform() collaborates with CAZON's ServiceManager to get the list of volatile services that have affinities with the actual node. For each of these volatile services, its associated integration transformation is obtained and a transformation pipeline is built with them. As we have mention in Section 3.2, the order in which the transformations are applied is important. Therefore a precedence value must be indicated for each service. The first transformation of this pipeline modifies the original style sheet loaded by the transformer and returns the result to the following, which takes it as an input. This goes on with the rest of the transformations, until the final result is obtained.

Finally, the transformed style sheet is applied to the XML representation of the node instance (which has been previously augmented) and the final user interface is obtained.

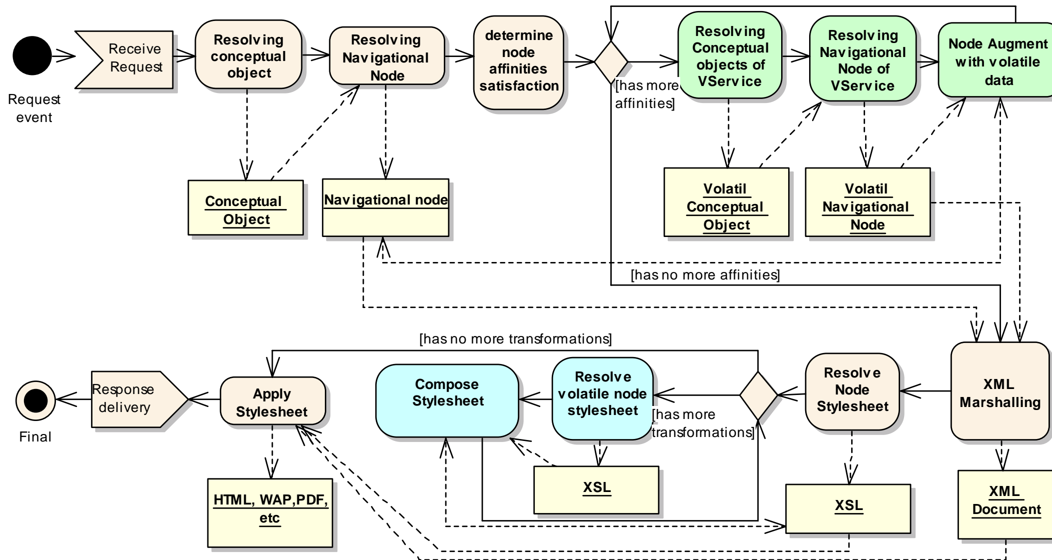

Figure 14. Overview of request processing in CAZON.

Figure 14 shows a UML activity diagram that summarizes the core steps of our implementation. In response to an HTTP request, a resulting node is resolved, enhanced and finally rendered. In order to do so, the conceptual object corresponding to the request and its navigational node are resolved. If the node has affinities with some volatile services, it is augmented (introducing attributes or links) with the nodes of the respective services. Next, the presentation layer obtains an XML representation of the node content which is processed with an appropriate XSL style sheet that provides the definitive visual format for the XML elements. If the node has affinities with volatile services, the core style sheet associated to the node is composed with the XSL documents which describe the user interfaces corresponding to the volatile nodes. This step is performed by constructing a pipeline with the XSL transformations obtained from each volatile service. When the core style sheet is introduced in the pipeline, each transformation is applied over it. Therefore the style sheet is successively enhanced with the interface elements of each volatile concern, until the definitive one is obtained. Finally, the enhanced style sheet is applied over the augmented XML representation of the node content to produce the response with the definitive rendering.

4 Discussion

Some of the main benefits of our approach to dealing with volatile functionality in Web applications follow:

- Thanks to separation of concerns, the design and implementation artifacts of the application remain cleaner and easier to understand, therefore simplifying the evolution of the application. In particular, there is no need to use dirty patches for volatile functionality in the source code. As a consequence, it is simpler to incorporate and remove new functionality as volatile features, mainly those that were not originally considered.

- Through the implementation of the approach we propose, it is possible to modify the behavior of the application at run time, and to have different configurations where different set of services are enabled at a given moment only by choosing which ones will be active. This high degree of modularization allows separating each concern in different developable modules avoiding maintaining several branches of a unique application project which contains tangled code of different concerns in the version control system and using complex software configuration management tools.
- By simply modifying the integration specification, it is possible to move a given volatile functionality in a different position (in the same page, or to a different page of the application) as well as to change the set of objects to which it is applied (affinity).
- Assuming that the new (volatile) functionality is wisely designed, it may be used in other contexts of the same or different applications. Therefore, different developers can work on different concerns in an independent way, avoiding conflicts between them. In addition, third party components developed with these ideas will be easier to integrate. Such is the case of Google Maps which has been incorporated to several web applications where most of them don't belong to Google.

Even though in this paper we focus on volatility at the interface level, as summarized in Section 2.2 and 2.3, our approach also covers the other layers usually characterizing a Web application, namely the conceptual and navigational ones. Therefore, most of the benefits discussed in this section also apply to those layers.

Applying a seamless concern integration approach combined with a model-driven style offers two possibilities of final composition. As a first possibility, we can produce the implementation model directly from the woven design model. This approach has the disadvantage that volatile functionality cannot be added at runtime. A second solution that we prefer is to weave "separated" implementation models at runtime. In order to do this, core and volatile design models have to be translated into implementation code and integration specifications into XSL transformations.

Finally, though we illustrated our approach with reference to the OOHDM method and the CAZON implementation framework, the basic ideas underlying it, specifically separation and oblivious integration of volatile concerns and some of our design and implementation schemas, can be used also with any model-driven approach, such as WebML [9], OOWS [32], UWE [25], UWA [15], etc.

5 Related Work

Volatile Requirements have been a research focus in the requirement engineering community for some years. In [30], the authors present an aspect-oriented approach for representing and composing volatile requirements using composition patterns [11]. Though we deal with the problem at design and implementation level, our approach complements the ideas in [30] for the kind of volatile services which are usual in the Web. We have also got inspiration from the so called symmetric separation of concerns approaches, such as Theme/Doc [10].

Incorporation of volatile functionality can be achieved, in the conceptual level, by means of a more classical aspect-oriented solution, such as [4], and with some syntactic sure the aspect-oriented

paradigm could also be used both in the navigation and interface levels. Our choice was to make a finer tuning of design and weaving mechanisms in different modeling concerns. As mentioned in Section 2, the main rationale of this approach has been first the object-oriented nature of OOHDM; we also took into account the specific features in each modeling level (e.g. navigation classes and ADVs). Besides we lean towards favoring symmetry during modeling and dynamism at weaving time. Finally, the fact that at the moment, to the best of our knowledge, there is no way to incorporate aspects in user interfaces of Web application (both during modeling and implementation) decided us to define a more “proprietary” language for the specification and XSL transformations at implementation time.

Some of the differences between our approach and the aspect-oriented one are the following:

- Our approach to specify volatile interfaces is symmetric and therefore similar to the approach proposed in [10]: both volatile and core entities are modeled using the same concepts (OOHDM primitives). On the other hand, an aspect approach is asymmetric: volatile concerns are modeled using a different programming construction (Aspects). Symmetric design increases concern’s independence and allows each concern to be modeled without awareness of other concerns (so they are modeled obliviously). While the controversy on symmetry has given raise to many (sometimes unfruitful) discussions in the community, we think that at the interface realm symmetry is preferable; while an aspect has no life without the corresponding core concept, volatile interfaces (specified as ADVs) stand by themselves and can be used both as extensions of a core interface or eventually as self-contained interfaces.
- With a typical aspect-oriented approach, the specification of join points where volatile functionality will be added is realized in the conceptual layer through the definition of pointcuts over syntactic elements of the implementation language. In our approach, we utilize a specific mechanism for each model we use: an object query language for expressing node affinities in the navigational layer and bidimensional pointcuts in the user interface layer (which are implemented with XPATH).
- An aspect affects all instances of a class, while specific instances of node classes can be indicated using affinities; in our solution affinities are evaluated for each instance at runtime so the aspect isn’t full applied to the class.

Weaving of XML documents can also be realized using DOM (Document Object Model) [16] which is a platform and language-neutral interface description that allows programs and scripts to dynamically access and update the content, the structure and the style of documents. Our decision of using XSLT was because of its flexibility and growing popularity.

Modern Web design methods have already recognized the importance of advanced separation of concerns for solving the problem of design (and code) tangling and scattering. For example, aspect oriented concepts are used in [6] and [36] to deal with adaptivity and customization respectively. Particularly, AspectUWA [36] uses aspect-orientation to overcome the "tyranny of the predominant decomposition". Though we don’t use aspects to deal with volatile functionality, the approaches are somewhat equivalent.

In [26] the authors present an XML Publication Framework based on the UWE approach; the transformation concept presented in this paper can be used to extend the framework of [26] to allow volatile concerns. One just needs to append transformations steps into the Cocoon pipeline.

Composition of interfaces has also been addressed in [27] by using operators of the tree algebra with the UsiXML description language [28]. A visual tool (ComposiXML) has been implemented with these ideas. The aim of that research is to help in the reuse of interface components. Our proposal uses affinity specifications and XSL transformations instead of tree algebra, focusing on oblivious integration of core and volatile interfaces; element reutilization is a consequence of the achievement of a decoupled design through oblivious components.

So far, we are not aware of any approach supporting oblivious composition of interface design models; meanwhile, in the XML field, the AspectXML project [5] has ported some concepts of aspect-orientation to XML technology, by allowing the specification of pointcut and advices similarly to Aspect Java. The project is still in a research stage.

In [12] a J2EE framework (named AspectJ2EE) that incorporates aspects on EJB components is presented. AspectJ2EE may be used to incorporate volatile concerns on J2EE applications at the model layer, though navigational and interface aspects are not mentioned in the project. The aspect weaving is performed at the deployment stage, while we propose to perform it in runtime in order to deal with volatile functionality that only affects some instances of a class dynamically.

Although Mash-Up technologies [14] allow services composition (even at interface layer), its intent and nature is quite different: applications consume services from several sources and services are orchestrated in order to provide a composed service. Volatile services may be external services but they are mainly developed for local application requirements. Indeed, a Mash-Up consumes services as a black box (using an API) and in the other hand Volatile Services may enhance some application service.

Navigational models enhancements were also discussed in [8] for performing adaptation in Web applications. By using aspects on the navigational units (at the navigational layer) adaptive behavior is introduced. This work presents an aspect oriented weaver which provides necessary aspect concepts (pointcuts and advices) for adding, removing and replacing different components of Navigational Units; the aforementioned affinities provide the same features at a higher level of abstraction.

As an alternative for XSL and AspectXML, an engine that allows enhancing XML document by means of refinements is presented in [1]. Refinements refer to functions (or composed functions) which incrementally introduce or override new element into XML documents. But the tool does not provide obliviousness because hooks (into the target XML document) have to be defined.

A topic not addressed in this paper but extensively detailed in [39] is the application of these ideas in the field of Rich Internet Applications (RIAs), particularly those built using AJAX or similar technologies (combination of XML and scripting languages). The behavior of RIA can be modeled using ADV-charts (a kind of Statechart). In the same paper we propose to use a solution based on broadcasting and reinterpretation of events (inspired on [29]) in order to weave ADV-charts belonging to different concerns. For the implementation stage, we use an XML + JavaScript based technology (such as, AJAX, Laszlo or XUL) and an API (described in [3]) which provides aspect oriented features for JavaScript. This API takes advantage of JavaScript facilities to redefine functions at runtime and to wrap one function into another. XSL transformations are used to incorporate blocks of JavaScript code belonging to the volatile functionality, which are then easily weaved with the core JavaScript functions by means of this aspect API. An example of this approach can be found in [39].

6 Concluding Remarks and Further Work

In this paper we have presented an approach for seamless and oblivious composition of Web applications' user interfaces. Our approach is grounded on the well-known principles of advanced separation of concerns to improve modularity and therefore to foster reuse and software evolution.

We have focused on one specific kind of application's concerns: those that encompass volatile functionality, i.e., the kind of functionality that cannot be guaranteed to be stable during the time.

Using our compositional approach, conceptual, navigation, and user interface models corresponding to core concerns can be made oblivious to the models corresponding to volatile requirements, which are designed using the same primitives. In particular, we referred to primitives of the OOHDM design method, but any other method distinguishing three layers in a Web application, namely conceptual (or content), navigational and presentation (or interface), could be used instead.

By using a very simple syntax, we indicate the way in which interface designs are composed and by using XSL transformations we are able to weave the corresponding XML files. In this way we don't need to pollute the application design model or source code with references to components that may be eliminated afterwards, thus requiring newer model and code editions.

We have realized these ideas in the context of CAZON, an OOHDM-based framework that automates dynamic weaving of volatile functionality into core application's modules both at the navigation and interface levels.

Although we have mainly focused on volatile concerns, the ideas in this paper can be used for weaving any kind of concern into the core application in those cases in which we want both (the concern and the core functionality) to evolve separately and obliviously.

Though we have followed a model-driven approach to tackle this problem, the same ideas can be applied even if more agile (i.e., code based) methodologies are used. For these approaches, the message of this paper is to keep core elements unaware of volatile concerns, no matter how the weaving process is finally realized.

We are currently working in several research directions related to this topic: first we are building tools to automate the translation of ADVs into XSL files, and pointcut specifications into XSL transformations to improve model-driven support in CAZON. Secondly, we are also analyzing other kinds of concerns (either volatile or not) in which interface weaving might be crosscutting, i.e., involving further changes in the core interface. Though XSL transformations can cope with this situation, we aim to improve our specification language to support more complex crosscutting. Third, we are working on the specification of the volatile functionality's lifetime: by means of an event query language, based on Esper [7], we can define the precise circumstances in which a service has to be activated o deactivated.

At the same time we are extending Cazon with the ability to automatically manage volatile services with this activation specification.

We are also studying the problem of conflicts among volatile models, and the impact that the order of execution of integration specifications has in the interface look and feel; this problem is similar to the problem of conflicts among aspects already reported in [18].

Finally, we are researching on the process of building prototypes from requirement specifications, which encompasses separated concerns using early aspects approaches such as those described in [20].

References

1. Anfurrutia, F. I., Díaz, O., and Trujillo S. "On Refining XML Artifacts". Proceedings of the 7th International Conference on Web Engineering (ICWE2007: July 16-20, 2007; Como, Italy).
2. Apache Struts framework. http://struts.apache.org/
3. Aspect Oriented Programming and Javascript. http://www.dotvoid.com view.php?id=43 (2007).
4. AspectJ, http://www.eclipse.org/aspectj/
5. AspectXML. The AspectXML home page. In www.aspectxml.org.
6. Baumeister, H., Knapp, A., Koch, N. and Zhang, G. "Modelling Adaptivity with Aspects". Proceedings of the 5th International Conference on Web Engineering (ICWE'05). Springer Verlag, Lecture Notes in Computer Science.
7. Bernhardt, T and Vasseur, A. "Esper: Event Stream Processing and correlation". ONJava, in http://www.onjava.com/lpt/a/6955, O'Reilly. August 2007.
8. Casteleyn, S., Van Woensel, W., Houben, G.J. "Semantics-based Aspect-Oriented Approach to Adaptation in Web Engineering", Proc. of 18th ACM Conf. on Hypertext and Hypermedia (HT'07; UK).
9. Ceri, S., Fraternali, P., Bongio, A. "Web Modeling Language (WebML): A Modeling Language for Designing Web Sites". Computer Networks and ISDN Systems, 33(1-6), 137-157, (2000).
10. Clarke, S., Baniassad, E. "Aspect-Oriented Analysis and Design. The Theme Approach". Addison-Wesley, Object Technology Series, 2005. ISBN: 0-321-24674-8.
11. Clarke, S., Walker, R. "Composition patterns: approach to designing reusable aspects" Proc. of 23rd Int. Conf. on Software Engineering. (Canada, 2001). ACM Press, 5-14.
12. Cohen, T. and (Yossi) Gil, J. AspectJ2EE = AOP + J2EE Towards Aspect Based, Pro-grammable & Extensible Middleware Framework. Proc.of ECOOP 2004. LNCS 3086, 219-243.
13. Cowan, D. Pereira de Lucena, C. "Abstract Data Views: An Interface Specification Concept to Enhance Design for Reuse". IEEE Transactions on Software Eng. 21(3): 229-243 (1995).
14. Daniel, F., Yu, J., Benatallah, B., Casati, F., Matera, M., Saint-Paul, R. "Understanding UI integration: survey of problems, technologies, and opportunities". IEEE Internet Computing (2007).
15. Distante, D., Pedone, P., Rossi, G., Canfora, G. "Model-Driven Development of Web Applications with UWA, MVC and JavaServer Faces." Proc. of 7th Int. Conf. on Web Engineering (ICWE2007, Italy).
16. Document Object Model. http://www.w3.org/DOM/
17. Douence, R., Pascal Fradet, P. and Mario Südholt, M. "Trace-based Aspects". Aspect-Oriented Software Development. Addison-Wesley, 2004.
18. Filman, R., Elrad, T., Clarke, S., Aksit, M. (eds.). Aspect-Oriented Software Development. Addison-Wesley, 2004.
19. Gamma, E., Helm, R., Johnson, R., Vlissides, J. "Design Patterns. Elements of reusable object-oriented software". Addison Wesley, 1995.
20. Gordillo, S., Rossi, G. Moreira, A., Araujo, A.,Vairetti, C., Urbieta, M. "Modeling and Composing Navigational Concerns in Web Applications. Requirements and Design Issues". LA-WEB 2006, pp. 25-31.
21. Güell, N., Schwabe, D., Vilain, P. Modeling Interactions and Navigation in Web Applications. ER (Workshops). (USA 2000) 115-127.

22. Harrison, W., Ossher, H, Tarr, P. "General Composition of Software Artifacts". Proc. of 5th Int. Symposium on Software Composition (SC 2006; Austria), pp. 194-210.
23. Jacobson, I.Object-Oriented Software Engineering. ACM Press 1996.
24. Kim, W. "Advanced Database systems", ACM Press, 1994.
25. Koch, N., Kraus, A., and Hennicker, R. "The Authoring Process of UML-based Web Engineering Approach". Proc. of 1st Int. Workshop on Web-Oriented Software Technology (IWWOST2001, Spain), pp. 105-119.
26. Kraus, A. and Koch, N. "Generation of Web Applications from UML Design Models using an XML Publishing Framework". Integrated Design and Process Technology Conf. (IDPT'2002).
27. Lepreux, S., Vanderdonckt, J. "Towards Supporting User Interface Design by Composition Rules". Proc. of CADUI'2006, Berlin.
28. Limbourg, Q., Vanderdonckt, J., Michotte, B., Bouillon, L., López-Jaquero, V. "UsiXML: a Language Supporting Multi-Path Development of User Interfaces". Proc. of 9th IFIP Working Conference on EHCI-DSVIS'2004.
29. Mahoney, M., Bader, A., Aldawud, O., Elrad, T., Using Aspects to Abstract and Modularize Statecharts. 5th Aspect-Oriented Modeling Workshop In Conjunction with UML (2004).
30. Moreira, A., Araujo, J., Whittle, J. "Modeling Volatile Concerns as Aspects", Proceedings of CAiSE 2006, Luxemburg, June 2006, pp 544, 558.
31. Nanard, M., Nanard, J., King, P. "IUHM: A Hypermedia-based Model for Integrating Open Services, Data and Metadata". Hypertext 2003: 128-137
32. Pastor, O., Abrahão, S., Fons, J. "An Object-Oriented Approach to Automate Web Applications Development". Proceedings of EC-Web 2001: 16-28.
33. Phanouriou, C. "UIML: A Device-Independent User Interface Markup Language". Ph. D. Thesis, Virginia University, 2000.
34. Rossi, G., Nieto, A., Mengoni, L., Lofeudo, N., Distante, D., "Model-Based Design of Volatile Functionality in Web Applications", Proc. of LA-WEB 2006, Mexico, IEEE Press, pp. 179-188.
35. Saxon. In http://saxon.sourceforge.net/, 2007.
36. Schauerhuber, A., "AspectUWA: Applying Aspect-Orientation to Model-Driven Development of Ubiquitous Web Applications". PhD Thesis, Vienna University of Technology, 2007.
37. Schwabe, D., Rossi, G. "An object-oriented approach to web-based application design". Theory and Practice of Object Systems (TAPOS), Special Issue on the Internet, v. 4#4, 1998, 207-225.
38. Stxx. Struts for Transforming XML with XSL. In http://stxx.sourceforge.net, 2005.
39. Urbieta, M., Rossi, G., Ginzburg, J., Schwabe, D.: "Designing the Interface of Rich Internet Applications" Proceedings of LA-WEB 07, Santiago, Chile, IEEE Press (2007).
40. UWA Consortium, "Ubiquitous Web Applications". Proceedings of the eBusiness and eWork Conference 2002, (e2002: 16-18 October 2002; Prague, Czech Republic).
41. Xalan. In http://xalan.apache.org/, 2007.
42. XPATH. XML Path Language. In http://www.w3.org/TR/xpath, 2007.
43. XSL. The Extensible Stylesheet Language Family. In http://www.w3.org/Style/XSL/, 2007.