Transparent Interface Composition in Web Applications

Gustavo Rossi [2], Jeronimo Ginzburg [1], Matias Urbieta[2], Damiano Distante[3]

[1]Departamento de Computación, Universidad de Buenos Aires, Argentina
jginzbur@dc.uba.ar
[2]LIFIA, Facultad de Informática, UNLP, La Plata, Argentina and Conicet
{gustavo, matias.urbieta}@lifia.info.unlp.edu.ar
[3]RCOST – Research Centre on Software Technology
Department of Engineering, University of Sannio, Italy
distante@unisannio.it

Abstract. In this paper we present an approach for oblivious composition of Web user interfaces, particularly for volatile functionality. Our approach, which is inspired on well-known techniques for advanced separation of concerns such as aspect-oriented software design, allows to clearly separate the design of the core's interface from the one corresponding to more volatile services, i.e. those that are offered for short periods of time. Both 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 interface code. Using some illustrative examples we focus both on design and implementation issues, presenting an extension of the OOHDM design model which support modular design of volatile functionality

1 Introduction

Even simple Web applications must deal with a myriad of concerns, each one of them encompassing multiple requirements. It is well known that by clearly decoupling application concerns in each development stage, and using proper composition mechanisms to weave corresponding design and implement artifacts, we can get more evolvable (Web) software [11]. However, while different techniques for advanced separation of concerns such as architectural and design patterns [9] and aspects [8] have been already introduced in the Web field (for example [2]), there are still many open problems, related with concern separation and composition, which have not been fully addressed. In this paper we focus on the design of volatile functionality, particularly on presentation issues.

In a Web application there may be many different kinds of volatile requirements giving raise to volatile functionality. Some of them may arise during the application's

evolution to check acceptability of the users' community (like beta functionality) and might be later considered or not core application services (e.g. forums or users' tags in Amazon.com). Others are known to be available only on short and determined periods of time such as functionality for giving donations after a catastrophe 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 left-over stock products), draws and concert tickets selling (associated with a CD), Amazon's "fishbowl performance" for new releases of an artist, etc.

Volatile functionality may also affect irregular sub-sets 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, including a short video, which will be surely removed after some weeks. Additionally at the bottom of the page there is a link to a Valentine's store, a volatile sub-store which was removed after St. Valentine's day.

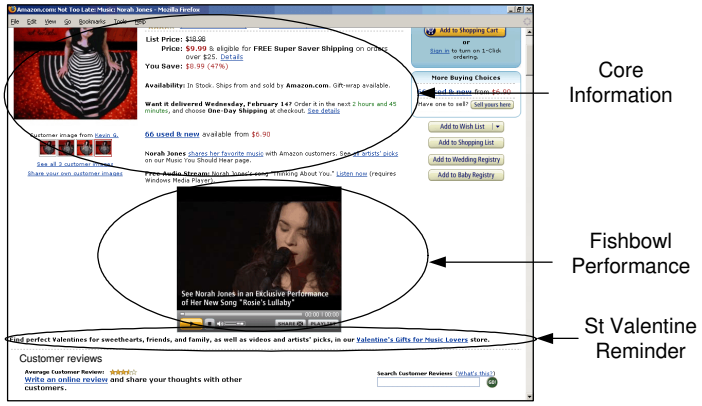

Figure 1: Different volatile functionality in Amazon.com

Volatile functionality poses many challenges to the designer, and also to design approaches. Suppose for example that we have designed a CD class or entity type using any Web design approach. If we want that some CD objects exhibit some new behaviors or data (e.g. a video performance), we can either add an attribute to the class, create a sub-class for the new behavior, re-design the class (type) so that it is now a composite, 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 to certain instances). However, in all cases we face a serious problem: being this functionality volatile, we might need to deactivate it after certain time. This means, once again, some kind of intrusive editing, which is cumbersome even if we are using a model driven approach and tool: the model has to be edited and this operation is certainly error prone. To make matters worse, if new requirements arose after introducing volatile modules, we might have a more complex tangle of core and volatile design components. Even using powerful configuration management environments, rolling back to the desired application's configuration might be a nightmare.

Unfortunately, the armory of existing Web design methods (object oriented ones like OOHDM [21], UWE [12] or OOWS [17], or those based on data modeling approaches like WebML [3]) lacks modeling primitives to deal with this problem.

In [18] we presented an extension to the OOHDM approach to deal with volatile functionality, both at the conceptual and navigational levels. Following our approach, we model volatile features as first class entities (e.g. classes), which are completely decoupled from core modules. Core and volatile classes are then "weaved" together at run-time, by using an integration specification. In this paper, we describe how we extended these ideas to the user interface realm, showing that it is also possible to create concern-specific interfaces which are obliviously composed according to integration specifications.

The rest of the paper is structured as follows: in Section 2 we present the background of our approach; next, in Section 3 we present an extension to the Abstract Data Views (ADV) interface design notation [7] used in OOHDM with ideas coming from aspect-oriented software design; we next introduce the idea of interface composition using transformations and illustrate our approach. In Section 4 we discuss some related work. Finally in Section 5 we conclude the paper and present some further work we are pursuing.

2 Integrating Volatile Functionality into OOHDM Models

Our approach is based on the idea that even the simplest volatile functionality (e.g. a video as in Figure 1), must be considered a first-class citizen and designed accordingly. Our extension to OOHDM can be summarized with the following design decisions, which are shown schematically in Figure 2:

- We decouple volatile from core functionality by introducing a model for volatile functionality (called Volatile Layer), which comprises both a conceptual and navigational models.
- New behaviors, i.e. those which belong to the volatile functionality layer are modeled as first class objects in the volatile conceptual model; they are considered as a combination of Commands and Decorators [9] of the core classes.
- As a consequence, we use inversion of control to achieve obliviousness; i.e. instead of making core conceptual classes aware of their 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.
- We design (and implement) the interfaces corresponding to each concern separately; the interface design of the core classes (described using ADVs [7]) are oblivious with respect to the interface of volatile concerns.

- Core and volatile interfaces (at the ADV and implementation levels) are woven by executing an integration specification, which is realized using XSL transformations.

In this section we focus only on conceptual and navigational issues while Section 3 describes user interface specification. The overall conceptual model of the application comprises two coarse grained packages, one containing the core functionality and the other which is itself composed of packages describing each volatile module. Classes in the volatile layer have a knowledge relationship to those classes which are extended with the new services. The navigational model is built analogously; there are no links between core and volatile nodes, though there might be links to the core navigational model (indicated with a dashed line in the diagram).

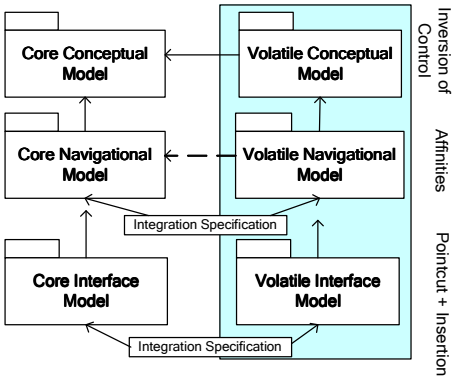

Figure 2: Overall Schema for Volatile Functionality

Nodes comprising volatile functionality are woven onto the core model using an integration specification, which is decoupled both from volatile and core 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. The nodes which are affected by the new functionality are called the affinity of the functionality.

In Figure 3, we present an oversimplified conceptual and navigational diagram, showing a couple of outstanding classes and node classes in the CD example; the intention of Figure 3 is to serve as an anchor for further examples in Section 3. More elaborated examples of volatile functionality and integration specifications can be found in [19].

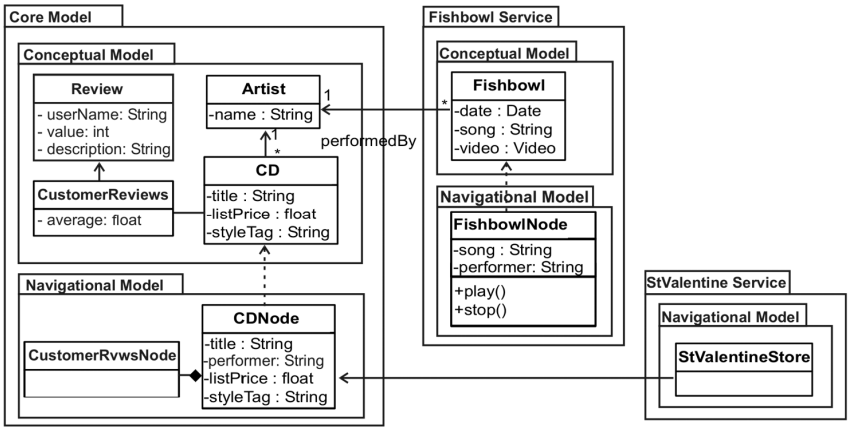

Figure 3: Core and Volatile Conceptual and Navigational Models

The integration specification for the video functionality is the following:

Affinity NorahJones (FROM FishbowlNode 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 enriched with the Norah Jones video (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 is indicated in the specification and the Anchor is associated to the unnamed link created at weaving time. Notice that in both cases the interface specification (and therefore some aspects of the implementation) also should change.

Our notation which is similar to the OOHDM node definition syntax (based on object queries) allows expressing different affinities for the same volatile concern, therefore allowing great flexibility in the resulting nodes without polluting core classes. For example, once the CD is not 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 re-located (during a period of time) in the home page.

In order to support our model-driven approach and to simplify the process of weaving volatile functionality into core nodes, we have implemented a framework (CAZON), on top of Apache Struts. CAZON supports semi-automatic translation of core and volatile OOHDM models into XML specifications, and manages the instan-

tiation of Web pages from the OOHDM navigational schema; the integration between core and volatile services is performed by executing the queries which specify the service affinities. A service manager evaluates affinity queries (stored in XML files) on the actual node which is 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 [19].

In the following sections we explain how we managed to apply the ideas of oblivious composition of volatile features in the user interface, both at the design and implementation levels. Though we focus on our implementation in 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 Improving Web Interface Composition

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 code which describes the interface of core components is polluted with tags belonging to volatile functionality and therefore both concerns (core and volatile) get tangled.

As an example, the JSP page that implements the CD interface (see Figure 1) will have knowledge on both St. Valentine store and on the Fishbowl component as shown in Figure 4.

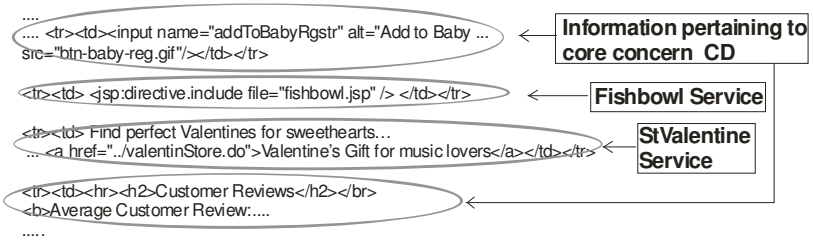

Figure 4: Code tangling in the user interface

In Figure 4, we can see two blocks of code which refer to the Fishbowl and 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's 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 (such as JSP, JSF, etc.) or user interface de-

scription languages (such as usiXML [15], UIML [18], 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 made oblivious interface code insertion) or on point-cuts such as in aspect-orientation. The *include* primitive, typical of scripting languages, does not guarantee obliviousness because it yields an explicit invocation.

It is not 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.

3.1 Composing Web Interface Designs

In OOHDM, the user interface is specified using Abstract Data Views (ADVs) [7], which support an object-oriented model for interface objects. 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. An ADV can be seen as an Observer [9] of the node, expressing its perception properties, in general as nested ADVs or primitive types (e.g. buttons). Using a configuration diagram [21] we 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 the result of user-generated events. These behavioral aspects, which are specified using ADV-charts [7] (a kind of Statechart), are outside the scope of the paper; we will only focus on structural interface aspects.

We have slightly modified the ADV notation in such a way that the positions of nested objects in the ADV reflect the look and feel of the interface as shown in Figure 5. This notation which is inspired in a similar one for UWE [12], allows improving discussions with different stakeholders, though it can not be processed automatically by standard ADV-based tools.

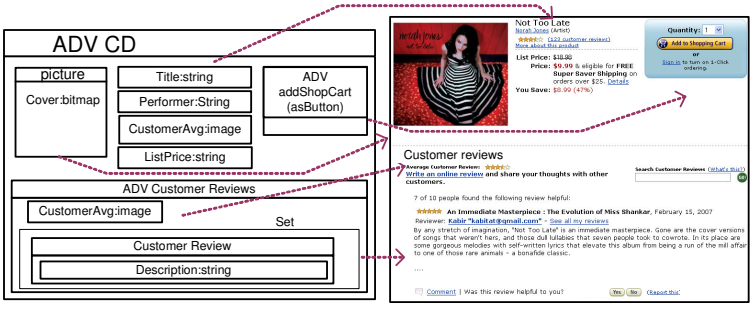

Figure 5: The ADV corresponding to the CD Node

As explained in Section 2, 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.

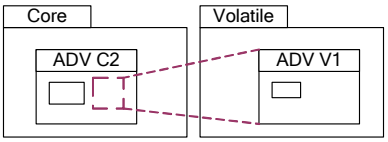

Figure 6: ADV V1 woven into ADV C2

To express the integration, we have defined a simple specification language which 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:

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

The field "IntegrationFor" refers to the navigational affinity as described in Section 2; the name of the affinity is necessary only when there is more than one affinity in the same concern. When the affinity is satisfied (at the navigational level) the interfaces must be composed according to the specification. The field "Target" indicates the name of the ADV (or ADVs) which 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 must be inserted in the target, either an ADV or an immediate specification, which is used when the inserted field is simple enough to avoid the specification of another (auxiliary) ADV. Finally we indicate the insertion position by using the "Relative" and "Position" fields. Notice that the specification is still "abstract", leaving place to fine tuning during implementation.

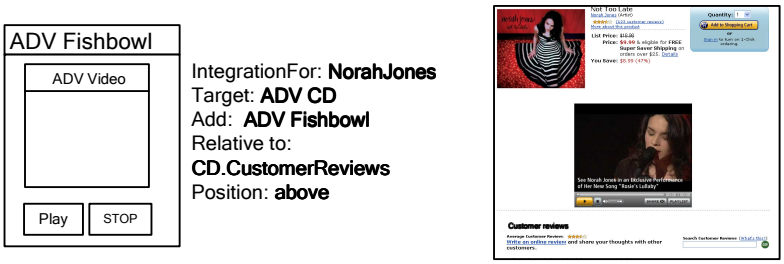

Figure 7: Integrating Fishbowl onto the ADV CD

In Figure 7 we show (on the left) the ADV for the Fishbowl volatile functionality and the integration specification which corresponds to the abstract interface of Figure 5. The result of the weaving process in the concrete interface is shown on the right of Figure 7. As shown, the ADV Fishbowl is placed between the CD's core information and the inner ADV Customer Review.

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 (left), we show the ADV Reminder which is used during the integration process of the Valentine's store. This ADV does not need to have an underlying navigational node and it provides an anchor to the St. Valentine's store which is realized in the CD node after the navigational weaving. The result of weaving the ADV Reminder into the ADV CD according to the specification in the right of Figure 8, gives as a result the ADV in Figure 9 (left) with a concrete interface shown in Figure 9 (right).

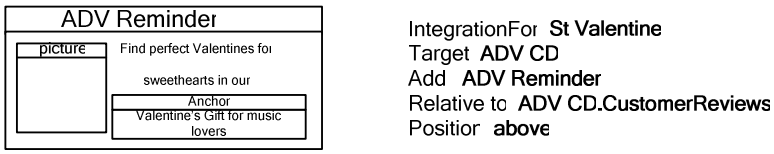

Figure 8: ADV Reminder and integration specification

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

A further research subject (see Section 5) is to analyze the impact of the order in which integrations are specified (in fact "executed") in the interface look and feel, for example in the case of the volatile concerns Fishbowl and St. Valentine.

As shown in [21], ADVs can be mapped systematically into concrete interface specifications in different running environments. Next, we show our approach to achieve obliviousness of interface code in the implementation stage.

3.2 Using transformations to compose XML documents.

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

The core of our solution is to use XSL [25] transformations to compose volatile and core interfaces, and XPATH [24] 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 which matches the template, indicates the point-cut where the advice is inserted. An XSL engine (e.g., Xalan[23], Saxon[20]) does the weaving process.

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

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

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

Pointcut

Advice

Figure 10: Transformation that inserts Fishbowl component

For example, in order to add the Fishbowl component to the CD Node, we can apply the XSL transformation shown in Figure 10 over the CD interface. The XPATH expression (point-cut) *"//tr[contains(.,'Customer Reviews')]"* refers to the row containing the text "Customer Reviews" (see Figure 4). The template (advice) leaves the existing elements of that row unchanged and inserts above a new one with the Fishbowl 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, this kind of transformations could be applied statically to incorporate the tags with volatile functionality without polluting the source code, as shown in Figure 11.

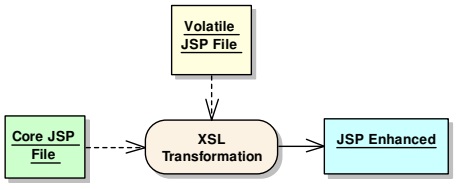

Figure 11: Statically weaving of volatile interfaces

A problem with this simple solution is that when volatile functionality only affects some instances of a class (e.g., those products which are recommended to be St. Valentine presents), some kind of conditional structure should be included in the tags, polluting the resulting code. As applying XSLT transformations in bare JSP during 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.

3.3 Our Approach in a nutshell

In CAZON for each node type we define a style sheet which transforms its XML representation 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 sheets 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's 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) 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) it yields a concrete interface, like the one in Figure 9, but without the reference to the St. Valentine's store.

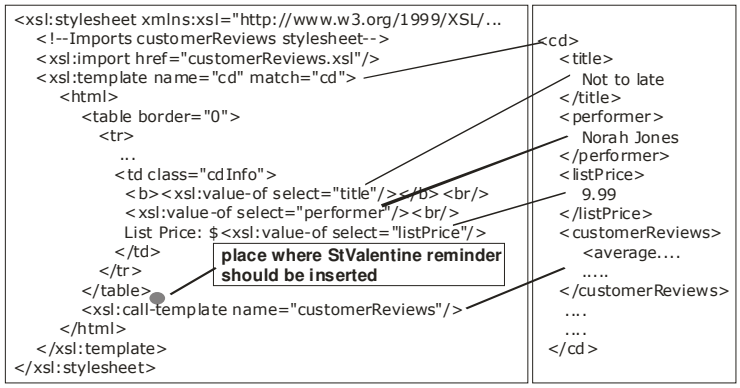

Figure 12: CD stylesheet and CDNode instance

As explained in Section 2, when a node's instance satisfies an affinity query corresponding to a volatile service, the corresponding node (in fact its XML representation) is augmented according to the indication of 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 a XSL transformation which applied over the style sheet corresponding to the core node, inserts the newly added elements. The transformation associated with the St. Valentine' Reminder integration specification (Figure 8) is shown in Figure 13.

```
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
   <!--Imports a transformation which copies all the elements-->
   <xsl:import href="defaultTemplate.xsl"/>
   <xsl:template match="//xsl:call-template[@name='customerReviews']">
      <xsl:copy-of select="document('reminder.xsl')//xsl:template[@name='reminder']/table"/>
      <xsl:copy-of select="."/>
   </xsl:template>
</xsl:stylesheet>
```

Pointcut

Advice

Reference to Valentine's reminder stylesheet

Figure 13: Transformation that inserts Valentine's reminder

We have realized these ideas in CAZON using Stxx [22], an extension of Struts which allows action classes to return XML data documents, to be transformed using different technologies into appropriated presentation formats.

To allow dynamic weaving of style sheets during run time, we created a sub-class of the Stxx AbstractXSLTransformer such that the method *transform()* collaborates with CAZON's ServiceManager to get the list of volatile services which had affinities with the actual node. For each of these volatile services, its associated integration transformation is obtained and applied over the base interface style sheet. Finally, the transformed style sheet is applied to its XML document (which has been previously augmented) and the final user interface is obtained, as shown in Figure 9 (right).

In figure 14 we show a UML activity diagram which summarizes the core steps of our implementation.

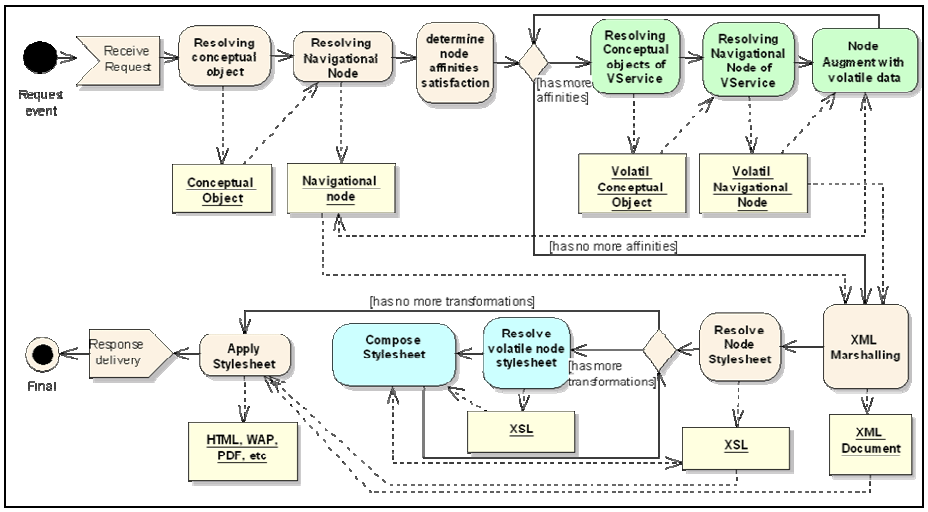

Figure 14: Overview of request processing in CAZON

We have finally defined a set of heuristics to allow semi-automatic translation of interface integration definitions (as those presented in Section 3.1) to XSL templates and XPATH specifications, assuming that the XSL style sheets which correspond to the ADVs have been manually codified.

4 Related Work

Volatile Requirements have been a research focus in the requirement engineering community for some years. In [16], the authors present an aspect-oriented approach for representing and composing volatile requirements using composition patterns [5]. Though we deal with the problem on another level of abstraction (design and implementation), our approach complements the ideas in [16] 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 [4]. Both volatile and core models are modeled using the same concepts, while in asymmetric approaches crosscutting concerns are modeled with a different primitive: the aspect.

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, in [2] aspect oriented concepts are used to deal with adaptivity. Though we don't use aspects to deal with volatile functionality, the approaches are somewhat equivalent.

In [12] 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 [12] to allow volatile concerns. One just needs to append transformations steps into the Cocoon pipeline.

Composition of interfaces has also been addressed in [14] by using operators of the tree algebra with the UsiXML description language [15]. 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.

So far, we are not aware of any approach supporting oblivious composition of interface design models; meanwhile, in the XML field, the AspectXML project [1] has ported some concepts of aspect-orientation to XML technology, by allowing the specification of point-cut and advices similarly to Aspect Java. The project is still in a research stage.

In [6] a J2EE framework (named AspectJ2EE) which 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.

5 Concluding Remarks and Further Work

We have presented an original approach for seamless and oblivious composition of Web applications' 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 which encompass volatile functionality, i.e. the kind of functionality

that can not be guaranteed to be stable. Using the compositional approach that we described in the paper, 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 our approach, those in the OOHDM design framework). By using a very simple syntax, we indicate the way in which interface designs are composed, and using XSL transformations we are able to weave the corresponding XML files. In this way we don't need to pollute the design model and the implementation code with references to components that may be eliminated (requiring newer code editions). We have realized these ideas in the context of CAZON, an OOHDM-based framework which automates dynamic weaving of volatile functionality into core application's modules, both at the navigation and interface levels. The ideas in this paper can be used for weaving any kind of concern into the application's core in those cases in which we want both (the concern and the core functionality) to evolve separately and obliviously.

We are now working on several research areas: first we are building tools to automate the translation of ADVs into XSL files, and point-cut specifications into XSL transformations to improve model-driven support in CAZON. 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. We are also studying the problem of conflicts among volatile models, and the impact which 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 [8]. Finally, we are researching on the process of building prototypes from requirement specifications which encompass separated concerns using early aspects approaches such as [10].

References

1. AspectXML. The AspectXML home page. In www.aspectxml.org
2. Baumeister, H., Knapp, A., Koch, N. and Zhang, G. Modelling Adaptivity with Aspects. 5th International Conference on Web Engineering (ICWE'05). Springer Verlag, Lecture Notes in Computer Science.
3. 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 June (2000)
4. Clarke, S., Baniassad, E. Aspect-Oriented Analysis and Design. The Theme Approach. Addison-Wesley, Object Technology Series, 2005. ISBN: 0-321-24674-8
5. Clarke, S., Walker, R. Composition patterns: an approach to designing reusable aspects in Proceedings of the 23 nd International Conference on Software Engineering. (Toronto, Canada, May 2001). ACM Press, 5-14.
6. Cohen, T. and (Yossi) Gil, J. AspectJ2EE = AOP + J2EE Towards an Aspect Based, Programmable and Extensible Middleware Framework. In Proceedings of ECOOP 2004. LNCS 3086, 219—243.
7. Cowan, D. Pereira de Lucena, C.: Abstract Data Views: An Interface Specification Concept to Enhance Design for Reuse. IEEE Trans. Software Eng. 21(3): 229-243 (1995)

8. Filman, R., Elrad, T., Clarke, S., Aksit, M. (eds.). Aspect-Oriented Software Development. Addison-Wesley, 2004.
9. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns. Elements of reusable object-oriented software, Addison Wesley 1995.
10. 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: 25-31
11. Harrison, W., Ossher, H, Tarr, P.: General Composition of Software Artifacts. Software Composition 2006: 194-210
12. Koch, N., Kraus, A., and Hennicker R.: The Authoring Process of UML-based Web Engineering Approach. In Proceedings of the 1st International Workshop on Web-Oriented Software Construction (IWWOST 02), Valencia, Spain (2001) 105-119
13. Kraus, A. and Koch, N.. Generation of Web Applications from UML Design Models using an XML Publishing Framework. Integrated Design and Process Technology Conference (IDPT'2002), June 2002
14. Lepreux, S., Vanderdonckt, J., Towards Supporting User Interface Design by Composition Rules, Proceedings of 6th Int. Conf. on Computer-Aided Design of User Interfaces CADUI'2006 (Bucharest, 6-8 June 2006), Chapter 19, Springer-Verlag, Berlin, 2006
15. Limbourg, Q., Vanderdonckt, J., Michotte, B., Bouillon, L., López-Jaquero, V., UsiXML: a Language Supporting Multi-Path Development of User Interfaces, Proceedings of 9th IFIP Working Conference on EHCI-DSVIS'2004
16. Moreira, A., Araujo, J., Whittle, J.: Modeling Volatile Concerns as Aspects, Proceedings of CAiSE 2006, Luxemburg, June 2006, pp 544, 558.
17. Pastor, O., Abrahão, S., Fons, J.: An Object-Oriented Approach to Automate Web Applications Development. Proceedings of EC-Web 2001: 16-28
18. Phanouriou, C.: "UIML : A Device-Independent User Interface Markup Language". Ph. D. Thesis, Virginia University, 2000
19. Rossi, G., Nieto, A., Mengoni, L., Lofeudo, N., Distante, D., "Model-Based Design of Volatile Functionality in Web Applications", Proceedings of LA-WEB 2006, Mexico 2006, IEEE Press, pp 179-188.
20. Saxon. In http://saxon.sourceforge.net/, 2007
21. 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, October, 1998, 207-225.
22. Stxx. Struts for Transforming XML with XSL. In http://stxx.sourceforge.net, 2005
23. Xalan. In http://xalan.apache.org/, 2007
24. XPATH. XML Path Language. In http://www.w3.org/TR/xpath, 2007
25. XSL.The Extensible Stylesheet Language Family. In http://www.w3.org/Style/XSL/, 2007.