International Journal of Software Engineering and Knowledge Engineering

World Scientific
www.worldscientific.com

MODELING, DEPLOYING, AND CONTROLLING VOLATILE FUNCTIONALITIES IN WEB APPLICATIONS

MATIAS URBIETA
LIFIA – Facultad de Informática, Universidad Nacional de La Plata
Argentina
matias.urbieta@lifia.info.unlp.edu.ar

GUSTAVO ROSSI
LIFIA – Facultad de Informática, Universidad Nacional de La Plata
Argentina
gustavo@lifia.info.unlp.edu.ar

DAMIANO DISTANTE
Unitelma Sapienza University
Italy
damiano.distante@unitelma.it

JERONIMO GINZBURG
Departamento de Computación, FCEyN, UBA
Argentina
jginzbur@dc.uba.ar

Received (Day Month Year)
Revised (Day Month Year)
Accepted (Day Month Year)

One of the main characteristics of most Web applications is their high dynamism. Once implemented and deployed for the first time, new functionalities are added to meet new or changed requirements. Some of these functionalities may appear on the Web in response to an unexpected event, or phenomena (such as a natural calamity) after which they are removed. Some others are activated periodically, to coincide with a particular date, or period of the year (such as, return to school, Christmas holidays, etc.). Implementing such *volatile functionalities* usually impacts on a number of aspects of a Web application, including content, navigation, presentation, business processes, and user operations. Their cyclic activation/deactivation, which requires repetitive changes in the application code, may be the cause of waste of effort and application quality deterioration, up to incorrect functioning.

In this paper, we present an approach to decouple the design and implementation of volatile functionalities from that of stable ones, i.e., the core functionalities of the application. The approach is instantiated in the context of the Object-Oriented Hypermedia Design Method (OOHDM), but its principles and related techniques are generally applicable to any other Web engineering method. We show how our approach enables the deployment and removal of these functionalities in a cost-effective and safe way and at runtime, thus providing business agility. A framework to classify volatile functionalities and a number of examples are also reported.

Keywords: Web application, Web engineering, volatile functionality, volatile concern, OOHDM.

1. Introduction

One of the most outstanding characteristics of Web applications is their continuous evolution in response to new requirements, or just for the purpose of keeping high their appeal. Functionalities which are implemented and activated once, in connection with an unexpected event and then removed definitely, or which are periodically activated during a particular period of the year, are referred as *Volatile Functionalities*; they correspond to the so called Volatile Concerns in [27]. As will be shown in this paper, a volatile concern might comprise a set of volatile functionalities.

Volatile functionalities are common in most popular Web applications; sometimes they are new features which are experimented for a period of time and then discarded because the user does not find them useful. Some other times they are triggered punctually in response to a specific event or set of conditions. More often they are periodically activated and de-activated in coincidence with specific periods of the year. Very often, the need for these functionalities arises after the application has been implemented and deployed for the first time and, as a consequence, they are not taken into consideration during the application design phase.

In an e-commerce Web site, such as Amazon.com, typical examples of such functionalities are: (*i*) the special offers available at certain periods of the year (e.g., Christmas, St Valentine, etc.) on specific products; (*ii*) the customization of contents for new releases (such as videos from Related Media performances); (*iii*) the functionality for fundraising after a catastrophe, and many others. Similar examples can be found in news sites, such as CNN.com to accommodate discussions on unexpected events, to include new advertisement types (e.g., during presidential elections), etc.

As an example, we show in Fig. 1 the Back-to-School promotion in Amazon.com; Fig. 1.a shows the volatile link in the home page and Fig. 1.b the Back-to-School page, which in turn points at products designed for schools. Some days before the summer holidays end and until the academic activities start, customers can access certain offers, and more specifically, can benefit from free "super-shipping" with certain constraints. After this period, these features are removed until the following year when they will be activated again.

In order to ease the introduction of volatile functionalities into Web applications and thus facilitate their evolution, reducing the risk of introducing errors in working software, we have defined an approach that enables the systematic design, implementation and automatic activation/deactivation of volatile functionalities using a model-driven development style.

This paper, besides presenting in a thorough way the results of our research concerning volatile functionalities, compared to our previous works on the subject [21, 34] introduces the following new contributions:

- It proposes a conceptual framework for characterizing volatile functionalities according to their essential properties and an example instantiation of this framework to characterize a set of volatile functionalities.

- It describes how to deal with volatile functionalities that impact on the business process model implemented by a Web application, by modifying the set of process activities included and their associated workflow.
- It provides guidelines on how to integrate our approach into Web engineering methods different from the Object Oriented Hypermedia Design Method (OOHDM) which we used as base model for our original research.
- It proposes a simple event-based language to specify different patterns of volatile functionalities according to the conceptual framework mentioned above.
- It shows how our volatility pattern specification language and its runtime support are integrated into the Web applications design and development framework supporting the OOHDM method.
- It illustrates additional examples of volatile functionalities and a case study of applying our approach to the design, implementation and management of a volatile functionality in the context of an e-commerce Web site.

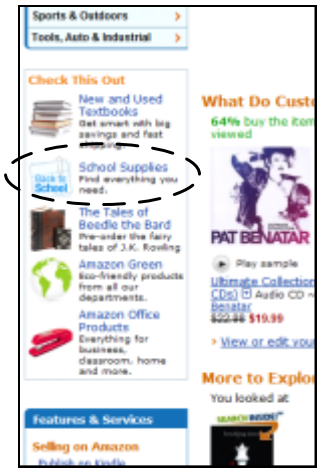


Fig. 1.a. The volatile link to the Back to School offers.

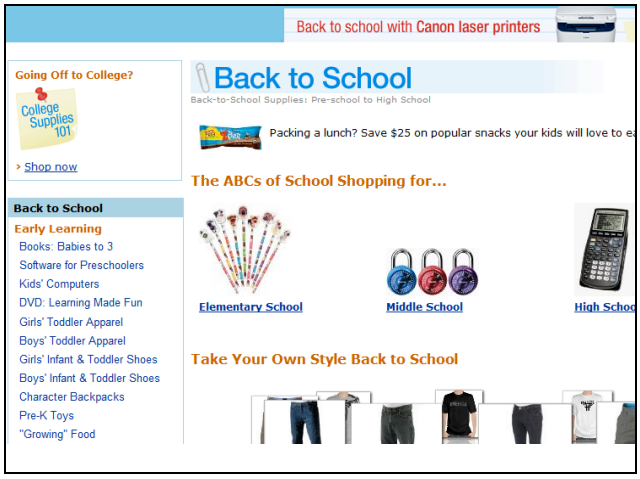


Fig. 1.b. Back to School main page.

The reminder of the paper is organized as follows: Section 2 presents our framework to characterize volatile functionalities; Section 3 discusses some related work; Section 4 synthesizes our model-based approach to deal with volatile functionalities in Web applications; Section 5 describes the life-cycle (activation/deactivation) of volatile functionalities and presents our solutions to automate its management, particularly the volatility pattern specification language; Section 6 evaluates our approach with empirical analysis; finally, Section 7 concludes the paper and introduces some of the future work that we are pursuing.

2. Characterizing Volatile Functionalities

The addition of volatile functionalities in Web applications might be thought as a particular case of Web software evolution and therefore tackled using existing

techniques, such as WebComposition Process Model [17], or more general principles, such as refactoring [15] and patterns [18]. We claim, however, that a specific approach is needed because of the very nature of these functionalities which distinguishes them from other evolution requirements: volatility.

Our research has been aimed at developing a model-driven approach to deal with volatile functionalities that makes it possible (*i*) to treat volatile functionalities similarly to core and stable ones, while keeping their design and implementation separated from the core application, and (*ii*) to automate their activation/deactivation according to a defined volatility pattern.

To reach our goal, we first tried to identify the main characteristics of volatile functionalities by analyzing several examples of them from real-world Web sites and their evolution during the time. From this analysis we found that a volatile functionality can be characterized at least along three dimensions that we defined as *Intent*, *Extent* and *Volatility Pattern*.

The *Intent* of a volatile functionality can be identified by answering the following questions: which aspects of the Web application does the volatile functionality impact? For each aspect, which application object types (or classes) have to be added or modified and how?

By taking as reference the design dimensions identified for a Web application by most Web engineering methods [37], aspects that the Intent of a volatile functionality may span include: content, navigation, user interface (or presentation) and behavior. Correspondingly, application objects types include, among others: content types, navigation nodes and paths, interaction widgets, business process workflows, business process activities, and business rules. Names adopted for the mentioned aspects (a.k.a., concerns or layers) and object types (a.k.a., modeling concepts) may vary depending on the considered Web engineering method.

The *Extent* of a volatile functionality identifies the set of application objects (instances of the types identified by the Intent) that it impacts. The Extent of a volatile functionality is indeed determined by answering the following questions: which application objects have to be added or modified for the types determined by the Intent? Is the volatile functionality complete? (i.e., does it affect all instances of the types) or does it apply to some specific instances? For each of the application objects types in the Intent, a way to determine the involved instances has to be specified.

The *Volatility Pattern* deals with describing the life-cycle of the volatile functionality, i.e., the rules governing its activation/deactivation during the time. In fact, this dimension distinguishes a volatile functionality from an evolution requirement. Although both an evolution requirement and a volatile functionality can be unforeseen in the beginning of the application development, a volatile functionality has a well defined *Volatility Pattern* which describes its life-cycle. This dimension of the characterization can be defined by answering the following questions: when has the volatile functionality to be activated? (e.g., after a specific event has occurred; in a fixed date; at a specific day of the week; at a specific time of the day; when certain business rules are satisfied, etc.). How long it has

to remain active? (e.g., a certain number of days, hours, minutes) or when it has to be deactivated? (e.g., when a certain time has passed from the activation; when a certain business rule is satisfied; when a certain day or time comes; etc.). Will it reappear later on? If yes, which is the pattern of repetition?

For each of the characteristics and sub-characteristics described above, we identified a set of possible, but not exhaustive, values it can assume. Overall, we obtained the characterization framework for volatile functionalities reported in Table 1.

Table 1. Characteristics of volatile functionalities with example values.

Characteristic		**Possible values**
Intent	Affected application aspects / objects types	• Contents / content types, views on contents, access structures to contents, etc. • Navigation / navigation nodes, navigation links, navigation paths, etc. • Interface (or presentation) / pages, look and feel of pages, interaction widgets included into pages, etc. • Behavior / user operations, business process workflows, business process activities, business rules, etc.
	Type of possible intervention	• Addition of new application objects types • Modification of existing application objects types
Extent	Affected application objects	• All the instances of the application objects types identified by the Intent • Only some of them, to be specified, e.g. with a query
	Added application objects	• Which new objects of the types identified by the Intent have to be created
Volatility Pattern	When	• At some specific date (e.g. Christmas) • At some fixed time of the day, day of the week, etc. • When some events occur (e.g., some item is out of stock, catastrophes) • According to some business rule (e.g., you want to promote some record; you want to celebrate the 100th customer of the day)
	For how long	• For a fixed period of time (e.g., 15 days) • Up to certain date (e.g., December 20) • Depending on a business rule (e.g., no more than x CDs sold) • According to human intervention
	Pattern of repetition	• E.g., every Monday, every first Sunday of the month, at 8.00am everyday.

It is interesting to highlight some particular characteristics of volatile functionalities shown in Table 1, which reflect the results of our analysis on successful Web sites. First, volatility, which always arises at the requirements level, can affect any application concern (content, navigation, interface and behavior); in some cases, volatile functionalities might span different concerns, e.g., a new content or link implies modifying the navigation and interface layers. Second, additions might be very irregular, e.g., only one or some specific instances of an application object type (content type, content view, navigation node, navigation path, user interface, interaction widget, user

operation, business rule, etc.) might be affected. Finally, volatile insertions might be crosscutting with core functionalities: in Section 4.3.2 we describe a volatile feature (Back to School products) that offers customers a shipping discount when purchasing a Kindle Amazon reader together with school products; this volatile feature (temporarily) modifies a basic business rule on shipping fares for a specific product, therefore crosscutting the core business model of the Amazon system.

In the following section we briefly review the literature on this field and in Section 4 we present the coarse-grained design decisions underlying our design approach for volatile functionalities.

3. Related Work

While software evolution has been studied for years now, volatility is a rather new problem in the literature. The impact of volatile requirements on the costs of a software product has been thoroughly studied [26]. It is now understood that software might vary in unforeseen ways and that the impact of these changes on the overall software costs are affected by an increasing application lifecycle complexity. For example, the Back to School functionality demands the activation/deactivation of several software components, and the execution of a number of software development activities (requirement analysis, design, implementation and testing) both for introducing the change and later removing it.

A first approach to deal with these changes in a modular way was presented in [28]; in this work the authors propose to capture volatile concerns during requirements analysis, model them as (early) aspects and follow existing Aspect Oriented Software Development (AOSD) techniques [14] in the overall application's life-cycle. Even though the approach is fairly general, it lacks the specificity of the interactive Web applications and therefore it does not cover specific features such as enriching a page or temporally changing an interface or navigation path.

As shown later in this paper, the key approach that we propose for dealing with volatile functionality is to achieve transparent plugging and un-plugging of the new features. In the context of Web applications we have found mainly three approaches to solve this problem. A component-based approach is presented in [17]. The Web Composition Process Model is a systematic approach for incorporating evolution as a first-class citizen in a Web application life cycle encouraging reuse through the so called "Web Application Evolution Bus". By profiting from contracts, it simplifies the addition of new functionality in components described using the Web Composition Mark-up Language. Differently from our approach, it does not deal with the mechanics of connecting/de-connecting new (volatile) components; besides, from the abstraction level point of view, it does not consider the problem at the different layers of a Web application model (conceptual, navigation, interface) but attacks mainly the implementation aspects.

The other outstanding trend to deal with transparent evolution is the use of Aspect-Oriented technologies as mentioned some paragraphs above. In the specific area of Web

applications, an interesting approach can be read in [5]. In this work the authors propose to tackle evolution using Aspect-Oriented Design Patterns. The approach is sound and powerful as the authors identify a set of possible types of changes in a Web application and associate an Aspect Pattern to solve each specific situation. However, it is geared towards implementation more than design and therefore it looses part of its power; nevertheless, as discussed throughout this paper, the underlying ideas behind Aspect orientation are key concepts to use with volatile functionality.

Another related work in which we found similar problems and ideas to ours is Web application adaptation. A Web modeling framework with orthogonal facilities for extending functionality in a seamless way is AMACONT [30]. This framework provides means for addressing adaptation in Web applications by implementing AOP concepts. Using aspect-oriented adaptation and semantics-based adaptation for different adaptation granularity, it allows specifying changes in a component-based model. The work provides a useful framework but lacks of means for specifying presentation aspects as well as describing aspects lifecycle.

In [4] the authors use an aspect oriented approach to incorporate adaptive navigation features into Web applications; the approach, built on the UWE method [37], supports most of the well-known adaptive navigation patterns in a modular and transparent way. By using aspects, it guarantees seamless activation and deactivation though limited to the navigation layer. Also, in [6] the authors propose a semantic aspect-oriented approach to adaptation which combines aspect-orientation with the powerful of Semantic Web concepts in the context of the Hera method [37]. In the context of Rich Internet Applications (RIAs), the authors of [20] propose OOH4RIA, an extension of the OO–H method that allows defining personalization adaptations for RIAs' presentation.

In [19] the authors introduce a software tool called AWAC based on A-OOH (the Adaptive OO-H [22]) for automatic generation of adaptive Web applications. A rule-based architecture for adaptation is presented in [9]; by using Event-Condition-Action (ECA) rules, the rule engine processes Web-generated events and triggers changes over application models to achieve adaptive behaviors. Though containing similar concepts, such as using a rule engine to determine when a change has to be made, the mentioned approaches are best suited to adaptation issues, while ours focuses on a broader spectrum of change patterns and besides it deals with incorporating and eliminating these changes.

Though transparent improvement of conceptual and navigational models has been treated in the literature, we are not aware of any approach supporting oblivious [14] composition of interface design models in such a way that different concerns keep orthogonal. In the XML field, the AspectXML project [3] has ported some concepts of aspect-orientation to XML technology by allowing the specification of point-cuts and advices similarly to Aspect Java. The project is still in a research stage.

Feature Oriented Programming (FOP) provides means for synthesizing software by composing features. A feature increments system functionalities by small refinements introducing classes, resources, dependencies, etc. and is intended to be reused in several applications. FOP and the approach presented in this work share the same underlying

goals: design application functionalities in a modular way where features are treated as first-class entities. In [1] a feature algebra framework for describing feature composition alternatives is presented. Operations such as features introduction and modification are formally described taking into account well-known algebraic properties such as associativity, commutativity, idempotency, among others. As it is a general feature framework, it is not instantiated in a specific application domain, leaving features composition implementation details uncovered. When FOP is combined with the Model Driven Development paradigm (MDD) it results in a new paradigm so called Feature Oriented Model Driven Development (FOMDD) in which models are refined instead of specific platform artifacts. For example, in [41], Trujillo et al. present a FOMDD approach for portlets product-line where porlets' models are refined with transformations that introduce functionalities. The approach does not take into account the volatility aspect of features and therefore it makes feature composition permanent.

4. Integrating Volatile Functionalities into Web Engineering Approaches

In most mature Web design approaches, such as UWE, WebML, Hera, OOWS or OOHDM (see [37] for description and examples of each approach), a Web application is designed with an iterative process comprising at least conceptual and navigational modeling. Some methods also include specific approaches for business process design [11] as well as techniques for requirements elicitation and interface specification. According to the state-of-the-art of model-driven Web engineering techniques [37], most of these design methods produce an implementation-independent model that can be later mapped to different run time platforms. For the sake of clarity we will concentrate on the conceptual, navigational and interface models as they are rather similar in different design approaches. While some comments on volatile requirements modeling are presented below, a thorough discussion on this aspect can be found in [28].

As most of the problems discussed so far apply to all development approaches, we will first describe the philosophy underlying our technical solutions in such a way that it can be reused; next, we will concentrate on the OOHDM design models and will briefly discuss how each part of our approach could be adapted to other methods. A detailed discussion on how to incorporate volatile functionalities in any other specific design method and/or model-driven framework is, of course, outside the scope of this paper.

Our approach is based on the idea that even the simplest volatile functionality (e.g., a video available for a period of time) should be considered as a first-class functionality and, as such, designed accordingly. At same time, their design and implementation have to be taken separated and as much as possible decoupled from that of core and stable functionalities.

Building on the above ideas, our approach can be summarized with the following design guidelines:

- We decouple volatile from core functionalities by introducing a design layer for volatile functionalities (called Volatile Layer) which comprises a requirements model, a conceptual model, a navigational model, and an interface model.

- Volatile requirements are modeled using the same notation used to model core requirements (e.g., use cases, class diagrams, user interaction diagrams, etc.) and separately mapped onto the following models using the heuristics defined by the design approach (See for example [43]). Notice that volatile requirements are not integrated into the core requirements model, therefore leaving their integration to further design activities.
- 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 [18] of the core classes. This strategy applies also to slight variants of business rules (such as adding a price discount to a product). In this case, the decoration is applied at the method level more than at the class level. Notice that this strategy can be applied to any object-oriented method, i.e., any method using a UML-like specification approach. In methods which are based on data modeling constructs, such as WebML, adding new (volatile) information is straightforward given that a precise composition language for entity types is defined. Alternatively, as we discuss in the related work section, an aspect-oriented solution is possible [14].
- Inversion of control is used to achieve obliviousness; i.e., instead of making core conceptual classes aware of their new features, the knowledge relationship is inverted. New classes know the base classes on top of which they are built. Core classes, therefore, have no knowledge about the additions. This also stands for aspect-oriented approaches.
- 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. This principle can be applied in any Web design approach.
- Separate integration specification is used to specify the connection between core and volatile nodes. As we show later in the paper, the integration is achieved at run-time. In other model-driven approaches, the integration can be performed during model transformation by implementing the corresponding transformations.
- The interfaces corresponding to each concern (core and volatile) is designed (and implemented) separately; the interface design of the core classes (described in OOHDM using Abstract Data Views (ADV) [42]) are oblivious with respect to the interface of volatile concerns. As in the navigational layer this principle is independent of the design approach.
- Core and volatile interfaces (at the ADV and implementation layers) are woven by executing an integration specification, which is realized using XSL transformations. Again, the idea of model weaving is generic and therefore the same result can be obtained using other technical solution.

We next explain how these principles have been put into practice in the OOHDM method. For the sake of comprehension and conciseness we focus mainly on those issues which are specific of Web applications (i.e., content, navigation, and presentation) and ignore fine grained changes in business rules which apply only to the conceptual model and do not impact on other models; however, this kind of volatile requirements have been treated thoroughly in the literature (see [28] for example) and can be tackled either using

aspects or object decorations [18]. Conversely, we also address volatile functionalities entailing coarse grained changes in business processes, such as the addition of new process activities.

Our approach supports features aggregation (aka, monotonic module composition [25]) by means of different technologies: (i) object-oriented programming patterns are adopted to model volatile features at the conceptual model level, (ii) affinities are used to enrich nodes in the navigation model, and (iii) XSL transformations are used in the interface models to add new structures in the views. Functionality removal (aka, non-monotonic module composition) and composition properties (e.g., commutativity and associativity [1]) are not covered in this work.

4.1. *Conceptual Design*

The conceptual model of a Web application (a.k.a., application, domain, or content model) is focused on defining the contents of the application with their attributes and associated behavior. When it is defined using the OOHDM method (or others such as UWE), this model is an object-oriented model described with UML and comprised of classes, with their attributes and methods, and associations between classes.

Volatile functionalities may involve brand new content classes (e.g., the class modeling a video content type) or the (volatile) modification of existing content classes and application behaviors. In our approach a volatile functionality is treated as a combination and generalization of *Commands* [18] and *Decorators* [18]. A new functionality is a command because it embodies an application behavior in one class, instead of a method. It can be considered also as a decorator because it allows adding new features (properties and behaviors) to an application in a non intrusive way. In our approach, volatile functionalities might be new behaviors which are added to the conceptual model (and which might encompass many classes) or full-fledged navigation models, containing new nodes, links and even relationships with conceptual classes. Each volatile functionality is treated as a self-contained sub-system and modeled using the OOHDM method. The notation is similar to symmetric approaches for separation of concerns such as the one described in [7].

By using UML package merging [10], it is possible to design models obliviously and seamlessly that, after their design, are merged producing a complex model combining different concerns; UML model elements such as packages and classes are woven. Only few platforms allow a direct translation of this model weaving concept such as built-in C# partial classes, AspectJ (AOP weaver) [2] and Fuji (FOP compiler) [16]. If the underlying platform does not support classes weaving, a programmer must take decisions that will not be traceable because there is no isomorphic relation between models and implementation artifacts.

The same solution can be applied when a business process is modified in a coarse grained way, e.g., when a new activity is injected in the process. This is possible in OOHDM as activities are modeled as first-class objects both in the conceptual and the navigational schema as exemplified in Section 4.2 (See [38]).

When the process workflow is simple (e.g., when it is just a sequence of atomic steps), the order of activities can be indicated by relating the corresponding classes in the conceptual model. These relations are later reflected in the linking structure of the navigational model (though, as explained in Section 4.2.3, the semantics associated with these links is slightly different from that of conventional links). When the process workflow is more complex, it is defined using a UML activity diagram and therefore we also need to indicate how the diagram is modified when the new activity in inserted. A thorough explanation of oblivious modifications (e.g., using aspects) of activity diagrams is outside the scope of the paper. Our approach is a simplification of the technique presented in Section 4.3.2 for behavioral weaving of user interfaces.

4.2. *Navigational Design*

The navigational design of a Web application is aimed at defining views, access structures and navigation paths to contents in order to enable the user easily accessing and navigating them. Most of Web engineering methods base their navigational model on two modeling primitives, namely *Node* and *Link*. OOHDM is no exception to this, as it defines nodes as logical views on application model classes and links as the hypermedia realization of application model associations.

At the navigational layer, volatile and core navigational components are connected using an integration specification which indicates, for example, if the volatile features are "inserted" in the core node or if they are connected with a hyperlink (as in Fig. 1). This specification also includes a query indicating which core nodes will contain the extension. Nodes matching the query are affected by (or enhanced with) the volatile functionality and represent the *Affinity* of the volatile functionality. The name we use was inspired by [29].

It is possible to define one or more affinities for the same volatile functionality, i.e., the same functionality might be incorporated in different parts of the application, by following different rules.

4.2.1. *Affinity Definition*

The affinities of a volatile functionality are specified with the same query language used in OOHDM to define nodes [37]. The language is based on object queries. Using this query language the definition of an affinity assumes the following form:

```
AFFINITY: AffinityName
FROM C1…Ci
WHERE Predicate
INTEGRATION: Extension | Linkage((V1…Vi))
```

In it, *AffinityName* is the name associated to the affinity, $C_1 \ldots C_i$ indicate core node classes involved in the query, *Predicate* is a logical expression defined in terms of properties of model objects which determines the instances of the core node classes $C_1 \ldots C_i$

that will be affected by the volatile functionality, and `Extension` / `Linkage` indicate the way the volatile functionality is integrated into core nodes through the volatile nodes $V_1 \ldots V_i$. An extension indicates that the core nodes are enhanced to contain the new functionality information (and operations). In a linkage integration, the core nodes "just" allow navigation towards the volatile nodes $V_1 \ldots V_i$ which actually contain the volatile functionality, and therefore does not support new behaviors. In the case of linkage integration, we can also specify additional features such as attributes or anchors that have to be added to the extended node (e.g., to make navigation more clear).

The requirements related with the Back to School volatile concern shown in Fig. 1 make it necessary to introduce a new node called BackToSchoolNode which acts as an index for the categories of products related to school. For the new volatile node to become easily accessible to users we need to define at least two linkage affinities from a core node to it: the first will link the Amazon home page to the index and the second will link each product node tagged as 'school' to the index.

The specification of these two affinities would look, respectively, as follows:

```
AFFINITY: Back to School affinity  - Home Page
FROM: HomePageNode
INTEGRATION: Linkage (BackToSchoolNode)

AFFINITY: Back to school affinity  - Generic link
FROM: ProductNode
WHERE: hasTag('School')
INTEGRATION: Linkage (BackToSchoolNode)
```

4.2.2. *Query Execution*

To improve flexibility, queries are intended to be executed at run-time; this allows to support irregular extensions, i.e., volatile functionalities which apply only to some specific nodes of a class, as the previous examples show. In this way, the weaving of volatile navigational functionalities does not occur during model compilation, but during application execution. Query execution and weaving of nodes in the context of the OOHDM method is supported by an application framework named Cazon which will be described later in this paper.

4.2.3. *Volatility in Business Processes*

A particular case of volatile insertion in the navigational model arises when a new activity is injected in a business process, as mentioned in Section 4.1. Suppose, for example, that the checkout process in an e-commerce Web site consists of six activities, namely: "login", "consolidate order", "confirm address", "select way of payment", "specify shipping options", and "confirmation". Suppose that during Christmas time we want to add the option of adding a "custom wrapping" for products after selecting the shipping options. In OOHDM, the navigational counterpart of activity objects are *activity*

nodes [37], hence, each of the activities in the checkout process mentioned above is associated to a corresponding node in the navigational model of the application. As a consequence, the injection of the new "custom wrapping" volatile activity consists in: (*i*) defining a new activity node in the volatile navigation model (correspondingly, an activity class is to be added in the volatile conceptual model), (*ii*) expressing the relationships between the new volatile activity node and the other core nodes of the application, and (*iii*) specifying the integration as described in Section 4.2.1.

OOHDM activity nodes are linked with each others by means of *activity links* whose semantics is defined by the relations from which they derivate in the conceptual model. Therefore, while the semantics of the *Linkage* integration type still holds (i.e., the new "custom wrapping" volatile node is linked from the core "shipping options" node), the exact semantics of the navigation path is defined in the conceptual model where the workflow constraints are defined. Then, when the user navigates through the activity link, the next activity node to be open will result from the corresponding volatile insertion in the conceptual model which relates the "custom wrapping" conceptual object with the previous activity object ("specify shipping options") and the "next" one ("confirmation"). The complete flow of this kind of volatile customization can be also derived from a technique presented earlier in [36].

4.3. *Abstract Interface Design*

In OOHDM, the user interface is specified using Abstract Data Views (ADVs) [37] which support an object-oriented model for interface objects. An ADV is defined for each node class to indicate how each node attribute or sub-node (if it is a composite node) will be presented to the user. An ADV can be seen as an Observer [18] of the node expressing its perception properties, in general, as nested ADVs or primitive types (e.g. buttons). Using a configuration diagram [42] we express how these properties relate with the node attributes and operations.

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 are specified using ADV-charts [42], a kind of statecharts representing states and state transitions for a given ADV. ADV-charts are useful when we need to model rich interface behaviors such as that of Rich Internet Applications (RIA) [42].
ADV-Charts are state machines diagrams[*] that allow expressing interface transformations occurring as the result of the user interaction on a given ADV. ADV-Charts describe interface behaviors through Event-Condition-Action rules. A complete description of ADV-charts and their use to specify interface behaviors can be found in [42].

4.3.1. *Structural Weaving of Volatile Functionality*

As a consequence of inserting volatile functionalities into the conceptual model or the navigational model, new interface elements must be added into the interface model,

[*] Or statechart diagrams if we refer to UML 1.x instead of UML 2.0.

therefore introducing new fields with data or control interface objects (anchors, buttons, etc.). Though we described this process in [21], we briefly review it here for completeness and readability reasons.

Each concern (core and volatile) will comprise ADVs for its corresponding nodes. During the interface design stage and when a node should exhibit some volatile functionality, we indicate the look and feel of the final page by specifying how the volatile interface will be inserted into the core ADV. More specifically, we indicate the relative position of the added interface objects with respect to the core interface objects. To express the integration, we have defined a simple specification language which allows indicating point-cuts and insertions at the abstract interface level, i.e. the position where the volatile ADV has to be inserted in the core ADV.

The specification generalizes the idea of point-cuts in aspect-orientation to the two dimensional space of Web interfaces. A point-cut and the corresponding insertion are specified using the following template:

```
Integration: IntegrationName
Target: ADVTargetName
Add: ADVSourceName | InsertionSpecification
Relative to: ADV name
Position: [above | bottom | left | right]
```

The field `Integration` is an identification for this specification. It may refer or not to a navigational affinity, since the same User Interface (UI) integration specification can be used with many navigational affinities. The field `Target` indicates the names of the ADVs (one or more) which will host the volatile interface code. Inner ADVs may be specified using a "." notation. As an example, Product.Reviews indicates that the insertion will take place in the ADV Reviews, which is a part of the Product ADV.

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, the insertion position is pointed put by using the `Relative` and `Position` fields.

It is worth to notice that the specification is still "abstract", thus leaving space to fine tuning during implementation.

As an example, let us consider again the Back to School volatile functionality example. The specification below (Back to School integration) indicates that the Amazon Home page ADV has to be enriched with an ADV with the behavior of an anchor (as UI component) to the Back to School page.

```
Integration: Back to School integration – Home page
Target: ADV Home page
Add: Anchor(ADV BackToSchool)
Relative to: ADVHomePage.CheckThisOut.NewAndUsedTextBooks
Position: Below
```

The specification indicates that the *Home page ADV* is to be modified with a new anchor defined at runtime to the *BackToSchool ADV*. The join-point is defined by the `Relative to` property which describes the path to the target pivot component. Finally the `Position` points out which place the new link must take. This specification tries to keep the ADV document model as simple as possible, by avoiding the introduction of technology dependent information. In Fig. 1, the link that was appended to the page is highlighted with a dashed oval, without any intrusive code editing.

In order to add a link from any product tagged as "school" to the "Back to School" page, associated to the second navigation affinity reported as example in Section 4.2.1, we can use the following integration specification:

```
Integration: Back to School integration – Generic link
Target: ADV Product
Add: Anchor(ADV BackToSchool)
Relative to: ADV AddToShoppingCart
Position: below
```

As it can be easily read, the integration specification increments the *Product ADV* with a link to the *BackToSchool ADV* to be positioned *below* the *AddToShoppingCart ADV* (which basically corresponds to a button UI component).

4.3.2. *Behavioral Weaving*

An interesting and certainly challenging situation arises when a volatile functionality crosscuts with core functionalities, e.g., when its insertion modifies existing behaviors in the core model. Crosscutting might manifest in different ways and, according to our characterization in Table 1, it might affect the conceptual, navigation and interface models. Crosscutting in the conceptual model can be solved in a straightforward way using well-known software engineering techniques, such as aspect-orientation, or with the correct use of certain patterns, as shown in our approach. Meanwhile, crosscutting in the navigational model strongly depends on the modeling style, and a thorough analysis of existing approaches shows that complex behaviors are not usual in this model as most core behaviors are allocated in the conceptual model. Therefore, and for the sake of conciseness, we focus on the interface model since, with the growing popularity or RIA, more and more sophisticated behaviors are presented in the user interface and can be affected by volatile insertions.

Continuing with our Back to School example, we want now to implement a new Shipping Promotion requirement so that when a product tagged as "School" is added to the shopping cart, the "*addToShoppingCart*" button pops-up a suggestion to add a product promotion (e.g., a hypothetical offer of a Kindle reader with free shipping promotion) as shown in Fig. 2.a and its corresponding ADV shown in Fig. 2.b; dotted arrows were used to describe relationships between abstract and concrete interface

elements. In order to introduce the new behavior, we need to change the interface behavior of an existing interface object (the *AddToShoppingCartButton*). Additionally, we need to trigger the addition of the Kindle item and apply the shipping discount if the user accepts the offer.

We have developed an extension of the ADV-chart notation to allow the specification of behavioral weaving in an oblivious way.

Fig. 3 shows how the behavioral weaving of the Kindle ADV with the Product ADV is specified, in this case a book, by means of ADV-charts, once the structural weaving has been performed. Basically, the ADV-chart describes the following behavior for the resulting interface:

- Intercept the "*mouseClick*" event of the *addToShoppingCartButton* (using the *catch* keyword).
- Enable the pop-up interface to be in the state *On*, and block the underlying page.
- End, returning control to the original ADV-chart (using the proceed keyword at the event 2) where the product will be appended to the cart in the first fragment of transition 2 or,
- Trigger the event which causes the product to be added to the cart together with a Kindle reader. After that, the business process goes on as expected using the keyword *proceed*.

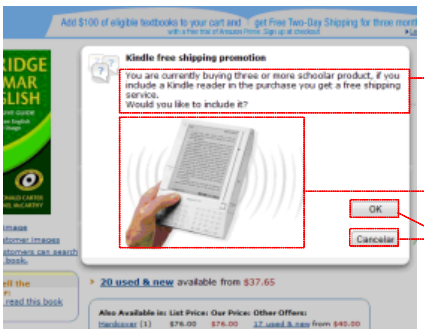


Fig. 2.a. Popup shown after AddToShoppingCart button was clicked.

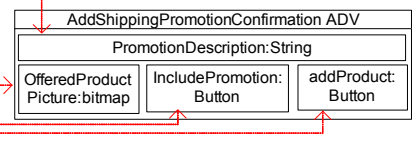
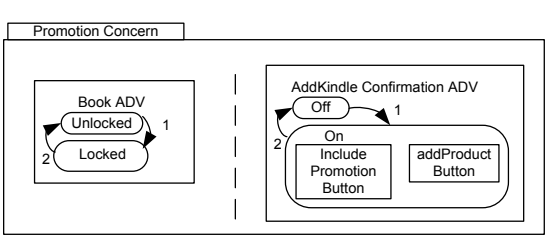


Fig. 2.b. Include Kindle Confirmation ADV.




```
1:
 Event: catch (mouseClick)
 Pre-Cond: AddToShoppingCart.hasFocus()
 Post-Cond: perCont=perCont+
            AddShippingPromotionConfirmation
2:
 Event: MouseClick
 Pre-Cond: addProductButton.hasFocus()
 Post-Cond: proceed()

 Event: MouseClick
 Pre-Cond:  IncludePromotionButton.hasFocus()
 Post-Cond: perCont=perCont-
            AddShippingPromotionConfirmation &&
            && owner.includesShippingPromotion()
            && proceed()
```

Fig. 3. The promotion confirmation ADV-Chart.

For the processing to be complete, two methods in the corresponding volatile node need to be added, namely *includesShippingPromotion* and *addShippingPromotion*. The former checks if the shopping cart registers a Shipping Promotion and the latter routine is used to register a promotion in the shopping cart. More details will be given in Section 5.2

Our approach takes advantage of well-known aspect oriented concepts in order to keep the core interface (Product ADV) oblivious from the Shipping Promotion requirement. In a few words, the solution is achieved by processing the original Product page and introducing the new behavior by means of a weaver. The weaving process takes the Shipping Promotion specification and binds its join-point definition (the catch of the mouse click event) with the target page components (AddToShoppingCart button).

5. Lifecycle Management of Volatile Functionalities

In this section we briefly describe how to put the modeling concepts into work by automating the process of activation/deactivation of volatile functionalities. We first present a simple model for reasoning on the life cycle of these functionalities, then we describe a rule-based language for simplifying the specification of activation/deactivation conditions.

5.1. *Life-Cycle Model*

In the context of our OOHDM-based approach and following the analysis presented in Section 2, we can reason on the life-cycle of a volatile functionality to better understand when it has to be connected into the application, when it has to be disconnected or re-connected in other pages, which design models it affects, etc..

Initially, the volatile functionality is introduced in the system in a passive state. When a specific event arises, like a fixed date arrives or an application event occurs, the functionality "wakes up" and turns into an "active" state where it starts participating by introducing its new behaviors through the weaving machinery. Later, other events may cause the functionality to be removed; this pair (active-passive) of transitions may happen several times depending on the functionality's requirements.

After a volatile functionality ends its expected life-cycle and enters into passive state, it can be kept in the application to be inserted again in the future, discarded immediately or eventually preserved to provide recorded information when needed, without affecting the implementation of the core Web application components.

While our model-based approach allows dealing with volatile functionalities modularly and decoupled from core functionalities, the process of connecting and disconnecting them should be also treated in a model-based way since, as explained in Section 2, volatile features can follow some predictable patterns which can help to automate their activation/deactivation. Therefore, we extended the integration specification language described in Section 4 by introducing production rules [12] to express the complex activation rules for volatile functionality.

The event conditions that trigger the activation/deactivation of a functionality are expressed using event patterns [44] which allow expressing rich event conditions, correlation and possibly spanning time windows. Complex event conditions can be specified combining different event patterns with first order logic.

Also, in order to allow a clear understanding of the activation rules by non-technical stakeholders, we express the conditions and actions using an appropriate Domain Specific Language (DSL). The rules written in DSL are then interpreted by a production rules engine with support for Complex Event Processing (CEP) [44], as we will see in the next section.

An activation rule looks as follows:

```
WHEN
    (Event_Pattern_Expression)
THEN
    (CONNECT | DISCONNECT)
    Concern concern_Name
    NAV_Affinity Affinity_Name
    UI_Integration Integration_Name
```

In the Mother's day example, the volatile functionality may be activated/deactivated at fixed dates using a time event. In the following expression volatile functionality is being activated a month before the Mother's day:

```
WHEN
    Time is *-Apr-14 00:00
THEN CONNECT Concern MothersDay
  NAV_Affinity MothersDay,UI_Integration MothersDayHomePage
```

For disconnecting the volatile functionality, another date-based condition is used in which the functionality is removed at Mother's Day end:

```
WHEN
    Time is *-May-14 23:59
THEN
    DISCONNECT Concern MothersDay
    NAV_Affinity MothersDay,UI_Integration MothersDayHomePage
```

By combining event-based and time-based queries, we are able to describe complex life cycles for volatile requirements.

5.2. *From Models to Running Applications. The Cazon Framework*

In order to support our approach to deal with volatile functionalities in the context of OOHDM, we have implemented a framework, called Cazon, on top of Struts [40]. The

framework supports the semi-automatic translation of OOHDM models into running applications and the introduction of volatile functionalities, working at the model level. Cazon allows mapping the affinities and integration specifications described in Section 4 into XML documents and supporting both the execution of queries during page construction and the weaving of interfaces. Cazon basically wraps the Struts request/response cycle by introducing volatile support.

A thorough description of the Cazon framework can be found in [21,35]; here we will focus on novel features related with the activation/deactivation of volatile functionalities.

A Cazon based application demands the definition of OOHDM module and the VService module. The former deals with the definition of navigational constructors (basically nodes and links) and can be used with or without the latter. The latter, meanwhile, decorates the OOHDM module and is in charge of augmenting the application nodes with volatile functionalities according to the navigational affinities associated to each node.

Each VService is the implementation counterpart of a volatile functionality and is also an OOHDM model. It contains all the conceptual classes, nodes definitions and the UI Stylesheets that implement the volatile ADVs. The VService objects may have references to some core application components, but the VService is oblivious to the navigational and UI integration specifications which are kept in external files. These files are configured as properties of the VService, as we will see in the following.

The navigational affinity specifications are mapped almost directly into XML files [35]. The UI integration is performed by means of XSL transformations. The 2D point-cuts where the volatile concern is added in the core interfaces are indicated by XPath expressions [34]. A complete description of the basic implementation is described in [21].

In order to introduce runtime activation/deactivation and configuration of volatile functionalities, each VService is managed by a JMX managed bean (MBean) [27]. A MBean is a Java object that represents a manageable resource, such as a volatile behavior. Through the MBean interface, all the functionality's properties, such as activation/deactivation life-cycle, navigational affinity and UI integration XML files, can be set at run-time. Each VService is separately packaged into a service archive file (SAR) and can be hot deployed on a JEE application server such as JBoss[24].

The VService's life-cycle is specified in a file using production rules which indicates the activation/deactivation conditions. The activation rules file is then interpreted by Drools [12], an open source production rules engine with CEP support.

As we have mentioned before, we defined a DSL that allows writing the activation rules in an almost natural language. Activation conditions written in DSL are mapped as event patterns. The events are represented as simple bean classes.

Once the activation rules are deployed, a stateful session of the rules engine will be listening for application events flow which can come from arbitrary channels. When the event condition of a rule is satisfied, the engine fires the consequence of the rule. In this case, the `connect()` or `disconnect()` method is invoked with the functionality's name, a navigational affinity XML file, and the XSLT file for UI integration, as

parameters. Finally the VServiceManager executes an operation on the VService JMX interface (retrieved by JNDI). This operation could be starting or stopping the volatile functionality or changing its integration configuration files.

The activation rules can be changed at run-time through a Web interface, allowing a quick change of the application when the business requires it (activation rules are also volatile).

6. Evaluating the Approach

This work promotes handling volatile concerns in Web applications as first class requirements and proposes a strategy to properly design, implement and deploy the corresponding volatile functionalities. The approach also enables controlling the activation and de-activation of these functionalities according to the application's business rules. As Web applications comprise several components and artifacts that are designed and implemented using different kinds of tools, the impact of introducing volatile concerns poses different problems depending on the affected application components.

In order to validate our approach and assess its benefits, we applied it to introduce two unforeseen concerns into an e-commerce application that is used as running example[†]. The two concerns are: *geolocalization*, which provides longitude and latitude to business objects including a suitable presentation of new data in a map, and *commentable*, which allows users to add comments to specific business objects.

After applying our approach using the Cazon framework in several different systems, we can summarize some lessons learned. Next we present a twofold analysis: a high level impact analysis where it is provided a detailed explanation of code changes, and a source code analysis where some well-known object oriented metrics are used.

6.1. *Impact Analysis*

In the following, for each Web application layer, we outline the potential changes that could be introduced when implementing a volatile requirement using a conventional object oriented approach, and compare them with that introduced by our approach:

- **Application (or domain) layer**
  - *Volatile instance variables and methods*: When using a "bare" object oriented (OO) approach to implement volatile functionalities, new elements such as variables, relationships, and accessors must be coded inside existing classes. Meanwhile, using our approach, new elements are encapsulated in decorators, and core classes are not modified.
  - *Volatile classes:* In a conventional OO approach, the introduction of volatile classes produces no significant impact into existing code. However, when removing the new classes, the application must be versioned again. In our

[†] Source code is available at: http://www.lifia.info.unlp.edu.ar/~murbieta/.

approach, new classes are packaged in a volatile component avoiding removing volatile requirement's code after it is deprecated and no longer needed.

- *Volatile crosscutting behavior:* To introduce volatile crosscutting behavior using an OO approach, already existing behavior must be updated changing stable and tested code. With our approach instead, by isolating volatile concerns, using decorator objects or a classical aspect-oriented (AO) approach, new behavior is introduced easily. The volatile functionality enriches core behavior with different strategies (i.e. pre-processing and post-processing).

- **Navigational layer**
  - *Volatile navigational nodes*: In conventional approaches, similarly to adding volatile classes in the application layer, adding new nodes in the navigational layer produces neither tangled nor scattered behavior, thus its impact is minimal to the core application. However, deactivating the volatile feature requires removing the new nodes and modifying again the application. Conversely, our approach promotes the packaging of new nodes in such a way that they are easily plugged and unplugged when implemented using Cazon.
  - *Navigational node operations*: In MVC Web frameworks such as Struts that Controller layer is based on command pattern [18], new operations can be encapsulated as new classes, thus avoiding the need to modify any page of the application. This solution produces previously mentioned new class's impact as it also introduces a new class.
- **Interface layer**
  - *User interface widgets*: When dealing with volatile functionalities with conventional OO approaches, new interface features are introduced in user interfaces components producing more complex code where the new feature is tangled with core features. In contrast, in our approach, new structural features, such as widgets or layout configuration, are specified using transformations and later weaved by a transformation engine. Therefore, the base user interface remains unchanged.
  - *RIA behavior*: In conventional approaches, behavior declaration is scattered among different objects, e.g., using JavaScript functions, code defined anonymously in widgets, or object methods. Code complexity becomes high since user interface definition is affected in several points by the new RIA behavior. Alternatively, our approach encapsulates the behavior in HTML documents and JavaScript artifacts complemented with an integration specification that contains the target of such behavior and how it should be introduced.

Overall, one of the major drawbacks of using conventional approaches (e.g., OO approaches) for handling volatile concerns is the fact that their life-cycle has to be supported in an ad-hoc fashion, by hard-coding the associated logic. This prevents reusability and demands the application to be modified and re-deployed again without

volatile requirements when they "expire". In our approach, life-cycle management of volatile concerns is supported by means of business rules combined with CEP which allows specifying time and business-based events responsible of enabling and disabling volatile features.

When using a traditional OO approach, volatile functionality development does not only compromise the coding task, but it also demands additional effort on testing since the affected features must be tested twice, both for volatile requirement introduction and removal. In a traditional OO approach, volatile requirements introduce changes on Web application functionality (e.g. method crosscutting behavior, instance variable, or RIA behavior) requiring regression tests in order to ensure that base features still work. When volatile functionality is not longer necessary, its code must be eradicated by means of manual, error-prone, tasks and so testing is needed. Meanwhile, with our approach, core behavior is not modified and thus regression testing is not needed. Instead, integration test is mandatory to verify that the woven concerns behave as expected. Only volatile requirements activation demands testing because their deactivation brings the application to a previous and already tested version.

6.2. *Source Code Analysis*

In order to assess how our ideas impact applications implementation, we have analyzed different applications source code measuring a variety of aspects using well-known metrics of Object-Oriented Programming [8]. In this analysis we will focus on (*i*) Source Line Of Code (SLOC or LOC) metric, which measures the size of the source code without comment lines; (*ii*) Lack of COhesion Metric (LCOM), which measures where class' features are not related to its modularization; (*iii*) class complexity, defined as the size of a class in terms of line of sentences; and (*iv*) code duplication, which detects duplication of code sentences.

We used the Sonar [39] source code quality tool for analyzing code automatically. This tool allows managing source code quality analyzing code complexity, design, coding rules, duplications, and potential bugs, among others. By using this tool, we compared how code changes when introducing volatile features using a conventional OO approach against using our proposed approach.

Before introducing analysis results, we must remark that the lack of modularization of crosscutting concerns increases application complexity when the application is evolving and growing, affecting different application source code aspects. First we will analyze how, when using an OO approach the application reacts to new features, and then we will provide a brief description of how our approach keeps modules simples.

The SLOC metric may be used as an indicator for predicting defect density [33]. The analysis showed up that when introducing volatile functionality using either a conventional OO approach or our approach, the SLOC metric increases. This is not surprising as, in both cases, volatile features are introduced by means of new objects and new object's state and behavior appended to object definitions, which ultimately correspond to new lines of code. Even though our approach promotes modularization,

new features must still be implemented and thus its SLOC adds up to the overall amount. Differently from a conventional OO approach, instead, our approach promotes the separation of concerns, and thus keeps classes smaller by having different artifacts for each feature instead of having a single class with both core and volatile functionality tangled. This kind of modularization has shown to prevent defects [13].

Application testability is compromised, at least, by two factors: the SLOC increment in an existing artifact, such as a class, and the increasing of LCOM metric. When new sentences are introduced in a class, its complexity increases demanding new test suites for testing the new feature. On the other hand, a lack of cohesion (LCOM) produces classes that encapsulate different features; this problem is known as the tyranny of dominant decomposition [31], which does not modularize concerns that are not framed by the main decomposition criterion. In some cases, this issue was registered as a code duplication metric increase.

The consequence of volatile functionality elimination is an error prone task because of its intrinsic crosscutting nature. Although we have not assessed the effort of manually removing a volatile functionality, several works have shown that the more changes a component has, more risk for a presence of bug exists [23]. Instead, using our approach, there is no chance of introducing a bug because no code is modified for introducing a volatile functionality thus a task for removing it is not needed.

7. Conclusions and Future Work

In this paper we have presented an integrated approach for dealing with volatile functionalities in Web applications. The approach is based on two main ideas: 1) treating volatile functionalities similarly to core and stable application's features (thus designing them prior to implementation), and 2) keeping their design and implementation separated and decoupled from core and stable features.

The proposed approach provides solutions to cover the whole life-cycle of volatile functionalities, from design to implementation, to deployment and run-time state management (activation/deactivation, according to their volatility pattern). It addresses volatility at the different application layers it can impact, including conceptual, navigational and user interface, and makes it possible to seamlessly integrate and manage volatile functionalities without the need for any modification to the application's core (stable) components.

Our proposal makes it possible to introduce new volatile functionalities in Web applications "on the fly", and enable non-technical people to control their activation rules at runtime, thus providing business agility to the application.

The approach has been derived from a conceptual framework, also presented in the paper, that can be used to characterize volatile functionalities and to reason on this kind of temporal-framed features which are nowadays typical in Web software.

We have shown, with simple but archetypical examples, how we model volatile functionalities and their weaving at the conceptual, navigational and interface layer. We have also shown how to specify their volatility pattern, i.e., the events/conditions under

which the new functionality has to be activated and deactivated, and the way in which we implemented this process.

We have described the overall structure of a run-time framework, named Cazon, that we developed for dealing with volatile functionalities in the context of the OOHDM model-driven Web engineering method.

We have presented a comprehensive and detailed example of application of our approach and the Cazon supporting framework to deal with a real-world volatile functionality in the context of an e-commerce Web application, and have discussed how our proposal could be applied to other Web engineering methods, different from OOHDM.

By treating volatile functionalities similarly to first class core functionalities, our approach simplifies the application's evolution and provides a framework for reusing both components and activation/deactivation rules. Our approach and our conceptual framework to deal with volatile functionalities can ease designers' tasks in: 1) evolving the application more seamlessly when unforeseen volatile functionality arises, and 2) understanding which type of volatile functionality might become "stable" and therefore will be integrated in the core application.

We are currently extending our research in different directions. First we are studying how to extend our semi-automatic integration approach to the conceptual layer for the particular case in which the new functionality only concerns this layer; while the process of dynamic page construction and deployment can be easily intercepted, as we did with Cazon, the same solution can not be applied automatically to the connection among application objects. We are studying techniques for dynamic weaving as in [32].

Correctness of Web concern compositions needs to be studied as it was done in [25] for Feature compositions to guarantee correctness of the resulting application. A language for describing concerns' dependencies (constraints) as well as expected hooks in different models (conceptual, navigational, and interface models) should be defined. The model checking task would be automated for easing designer tasks. The length restrictions of the Journal prevent us to further discuss this issue which requires much space. Additionally this discussion is somewhat out of the present scope of the paper.

We are also working on the integration of our approach in other model-driven Web engineering methods and particularly analyzing the integration at the meta-model level; by analyzing existing ideas to bridge and/or unify methods [37] we can find a way to express volatility in a higher abstraction level.

Finally, we are constantly analyzing and assessing Web applications in order to obtain additional feedback for our conceptual framework related to this kind of Web application evolution.

References

[1] S. Apel, C. Lengauer, B. Möller, and Christian Kästner. An Algebraic Foundation for Automatic Feature-Based Program Synthesis. Science of Computer Programming (SCP), 75(11):1022-1047, November 2010

[2] AspectJ, http://eclipse.org/aspectj/
[3] AspectXML, The AspectXML home page. in www.aspectxml.org.
[4] H. Baumeister, A. Knapp, N. Koch, and G. Zhang, Modelling Adaptivity with Aspects, in Proc. of the 5th International Conf. on Web Engineering (ICWE'05), Lecture Notes in Computer Science, Springer Verlag, 2005.
[5] M. Bebjak, V. Vranic, and P. Dolog, Evolution of Web Applications with Aspect-Oriented Design Patterns, in Proc. of the 2nd International Workshop on Adaptation and Evolution in Web Systems Engineering (AEWSE'07). CEUR Workshop Proceedings, ISSN 1613-0073. http://CEUR-WS.org/Vol-267/, 2007.
[6] S. Casteleyn, W. Van Woensel, and G. J. Houben, A semantics-based aspect-oriented approach to adaptation in web engineering, in Proc. of the 18th ACM Conf. on Hypertext and Hypermedia (HT07; September 10-12 2007; Manchester, UK;), ISBN 978-1-59593-820-6. pp. 189-198, ACM 2007.
[7] S. Clarke and E. Baniassad, Aspect-Oriented Analysis and Design, The Theme Approach, Addison-Wesley, Object Technology Series, 2005. ISBN: 0-321-24674-8.
[8] S. H. Chidamber and C. F. Kemerer , A Metrics Suite for Object Oriented Design. IEEE Trans. Software Eng. (TSE) 20(6):476-493, 1994.
[9] F. Daniel, M. Matera, A. Morandi, M. Mortari, and G. Pozzi, Active Rules for Runtime Adaptivity Management, in Proc. of the 2nd Int. Workshop on Adaptation and Evolution in Web Systems Engineering (AEWSE'07). CEUR Workshop Proceedings, ISSN 1613-0073, http://CEUR-WS.org/Vol-267/, 2007.
[10] J. Dingel, Z. Diskin, and A. Zito, Understanding and Improving UML Package Merge, Understanding and improving UML package merge, Software & Systems Modeling, Vol. 7, No. 4. (4 October 2008), pp. 443-467.
[11] D. Distante, G. Rossi, G. Canfora, and S. Tilley, A Comprehensive Design Model for Integrating Business Processes in Web Applications, in Int. Journal of Web Engineering and Technology (IJWET), Vol. 3, Issue 1, 2007, pp 43-72. Inderscience Publishers, 2007.
[12] Drools, at https://www.jboss.org/drools/ .
[13] K. El Emam, S. Benlarbi, N. Goel, W. L. Melo, H. Lounis, S. N. Rai, The Optimal Class Size for Object-Oriented Software. IEEE Trans. Software Eng. (TSE) 28(5):494-509, 2002.
[14] R. Filman, T. Elrad, S. Clarke, and M, Aksit, Aspect-Oriented Software Development. Addison-Wesley, 2005.
[15] M. Fowler, Refactoring: Improving the Design of Existing Code, Addison-Wesley, 2000.
[16] Fuji: A Compiler for Feature-Oriented Programming in Java, http://fosd.de/fuji.
[17] M. Gaedke and G. Graf, Development and Evolution of Web-Applications using the WebComposition Process Model, in Web Engineering, Software Engineering and Web Application Development. LNCS 2016 Springer 2001, pp 58-76.
[18] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, Design Patterns. Elements of Reusable Object-Oriented Software, Addison-Wesley, 1995.
[19] I. Garrigós, C. Cruz, and J. Gómez, A Prototype Tool for the Automatic Generation of Adaptive Websites, in Proc. of the 2nd Int. Workshop on Adaptation and Evolution in Web Systems Engineering (AEWSE'07). CEUR Workshop Proceedings, ISSN 1613-0073. http://CEUR-WS.org/Vol-267/, 2007.
[20] I. Garrigós, S. Meliá, and S. Casteleyn, Personalizing the Interface in Rich Internet Applications, in Proc. of the 9th Int. Conf. on Web Engineering (ICWE 2009), LNCS 5648, Springer-Verlag, pp. 292-299, 2009.
[21] J. Ginzburg, D. Distante, G. Rossi, and M. Urbieta, Oblivious Integration of Volatile Functionality in Web Application interfaces, in Journal Web Engineering. Vol. 8, N.1, pp. 25-4, Rinton Press, 2009.
[22] J. Gómez, C. Cachero, and O. Pastor , Conceptual Modelling of Device-Independent Web Applications, IEEE Multimedia, Special Issue on Web Engineering, pp. 26–39, 2001.
[23] A. E. Hassan, Predicting faults using the complexity of code changes. ICSE 2009:78-88.
[24] JBoss application server at http://www.jboss.org/jbossas .

[25] M. Kuhlemann, D. Batory, and C. Kästner, Safe composition of non-monotonic features. in Proc. of the 8th international conference on Generative programming and component engineering (GPCE '09). ACM, New York, NY, USA, 177-186.
[26] G. P. Kulk, and C. Verhoef, Quantifying requirement volatiliy effects. Science of Computer Programming 72 (2008), pp 136-175
[27] MBean JBoss microkernel, http://docs.jboss.org/jbossas/jboss4guide/r4/html/
[28] A. Moreira, J. Araujo, and J. Whittle, Modeling Volatile Concerns as Aspects, in Proc. of the 18th Conf. on Advanced Information Systems Engineering (CAiSE 2006). LNCS 4001 Springer, 2006.
[29] M. Nanard, J. Nanard, and P. King, IUHM: A Hypermedia-based Model for Integrating Open Services, Data and Metadata, in Proc of 14th ACM Conference on Hypertext and Hypermedia (HT03; Nottingham, UK; 26-30 August 2003). ACM Press (2003), pp. 128-137.
[30] M. Niederhausen, K. Van Der Sluijs, J. Hidders, E. Leonardi, G. J. Houben, and K. Meißner, Harnessing the Power of Semantics-Based, Aspect-Oriented Adaptation for AMACONT, in Proc. of the 9th Int. Conf. on Web Engineering (ICWE 2009). LNCS, Springer Berlin / Heidelberg, 2009, pp. 106-120.
[31] D. L Parnas, On the Criteria To Be Used in Decomposing Systems into Modules, Commun, ACM (CACM) 15(12):1053-1058, 1972.
[32] A. Popovici, T. R. Gross, and G. Alonso, Dynamic weaving for aspect-oriented programming, in Proc. of the 1st Int. Conf. on Aspect-Oriented Software Development, AOSD 2002, ACM 2002, pp.141-147.
[33] C. Rahmani, D. Khazanchi, A Study on Defect Density of Open Source Software. ACIS-ICIS 2010:679-683, 2010.
[34] G. Rossi, J. Ginzburg, M. Urbieta, and D. Distante,Transparent Interface Composition in Web Applications, in Proc. of the 7th Int. Conf. on Web Engineering (ICWE2007). LNCS 4607 Springer 2007.
[35] G. Rossi, A. Nieto, L. Mengoni, N. Lofeudo, L.S. Nuño, and D. Distante, Model-Based Design of Volatile Functionality in Web Applications, in Proc. of the 4th Latin American Web Congress (LA-WEB 2006). IEEE Computer Society, 2006.
[36] G. Rossi, H. Schmid, and F. Lyardet, Customizing Business Processes in Web Applications, in Proceedings the 4th Int. Conf. on E-Commerce and Web Technologies (EC-Web-2003).LNCS 2738 pp 359-368, Springer 2003.
[37] G. Rossi, O. Pastor, D. Schwabe, and L. Olsina, Web Engineering: Modelling and Implementing Web Applications. Human-Computer Interaction Series. Springer, London, 2008.
[38] H. Schmid, and G. Rossi, Modeling and Designing Processes in E-Commerce Applications, in IEEE Internet Computing 8(1): 19-27,2004.
[39] Sonar, http://www.sonarsource.org/, Accessed 4 June 2011.
[40] Struts Framework, http://struts.apache.org/ .
[41] S. Trujillo, D. Batory, and O. Diaz, Feature Oriented Model Driven Development: A Case Study for Portlets, In proc. of the 29th international conference on Software Engineering, Vol. 0 (ICSE '07), pp. 44-53.
[42] M. Urbieta, G. Rossi, J. Ginzburg, and D. Schwabe, Designing the Interface of Rich Internet Applications, in Proc. of LA-WEB 07, Chile, IEEE Press, 2007.
[43] P. Vilain, D. Schwabe, and C. Sieckenius De Souza, A Diagrammatic Tool for Representing User Interaction in UML, in Proc. of UML 2000, Lecture Notes in Computer Science 1939, pp 133-147, Springer 2000.
[44] D. Zimmer, and R. Unland, On the semantics of complex events in active database management systems, in Proc. of the 15th Int. Conf. on Data Engineering, (ICDE 1999). pp. 392-399, 1999.