To appear in: LNCS Transactions on Large Scale Data and
Knowledge Centered Systems 1(1), 2009

Facilitating Controlled Tests of Website Design Changes using Aspect-Oriented Software Development and Software Product Lines

Javier Cámara[1] and Alfred Kobsa[2]

[1] Department of Computer Science, University of Málaga
Campus de Teatinos, 29071. Málaga, Spain
jcamara@lcc.uma.es
[2] Dept. of Informatics, University of California, Irvine
Bren School of Information and Computer Sciences. Irvine, CA 92697, USA
kobsa@uci.edu

Abstract. Controlled online experiments in which envisaged changes to a website are first tested live with a small subset of site visitors have proven to predict the effects of these changes quite accurately. However, these experiments often require expensive infrastructure and are costly in terms of development effort. This paper advocates a systematic approach to the design and implementation of such experiments in order to overcome the aforementioned drawbacks by making use of Aspect-Oriented Software Development and Software Product Lines.

1 Introduction

During the past few years, e-commerce on the Internet has experienced a remarkable growth. For online vendors like Amazon, Expedia and many others, creating a user interface that maximizes sales is thereby crucially important. Different studies [11,10] revealed that small changes at the user interface can cause surprisingly large differences in the amount of purchases made, and even minor difference in sales can make a big difference in the long run. Therefore, interface modifications must not be taken lightly but should be carefully planned.

Experience has shown that it is very difficult for interface designers and marketing experts to foresee how users react to small changes in websites. The behavioral difference that users exhibit at Web pages with minimal differences in structure or content quite often deviates considerably from all plausible predictions that designers had initially made [22,30,27]. For this reason, several techniques have been developed by industry that use actual user behavior to measure the benefits of design modifications [17]. These techniques for *controlled online experiments* on the Web can help to anticipate users' reactions without putting a company's revenue at risk. This is achieved by implementing and studying the effects of modifications on a tiny subset of users rather than testing new ideas directly on the complete user base.

Although the theoretical foundations of such experiments have been well established, and interesting practical lessons compiled in the literature [16], the infrastructure required to implement such experiments is expensive in most cases and does not support a systematic approach to experimental variation. Rather, the support for each test is usually crafted for specific situations.

In this work, we advocate a systematic approach to the design and implementation of such experiments based on *Software Product Lines* [7] and *Aspect Oriented Software Development* (AOSD) [12]. Section 2 provides an overview of the different techniques involved in online tests, and Section 3 points out some of their shortcomings. Section 4 describes our systematic approach to the problem, giving a brief introduction to software product lines and AOSD. Section 5 introduces a prototype tool that we developed to test the feasibility of our approach. Section 6 compares our proposal to currently available solutions, and Section 7 presents some conclusions and future work.

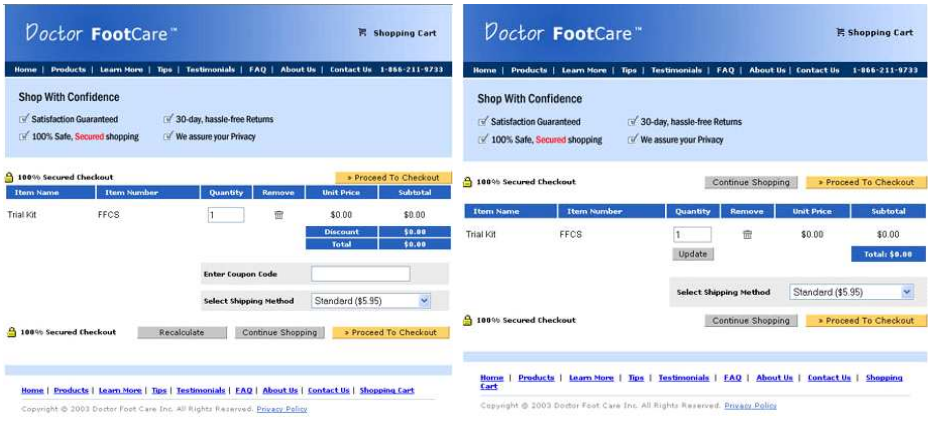

Fig. 1. Checkout screen: variants A (original, left) and B (modified, right)[1]

2 Controlled Online Tests on the Web: an Overview

The underlying idea behind controlled online tests of a Web interface is to create one or more different versions of it by incorporating new or modified features, and to test each version by presenting it to a randomly selected subset of users in order to analyze their reactions. User response is measured along an *overall evaluation criterion* (OEC) or *fitness function*, which indicates the performance of the different versions or *variants*. A simple yet common OEC in e-commerce is

[1] © 2007 ACM, Inc. Included by permission.

the conversion rate, that is, the percentage of site visits that result in a purchase. OECs may however also be very elaborate, and consider different factors of user behavior.

Controlled online experiments can be classified into two major categories, depending on the number of variables involved:

- **A/B, A/B/C, ..., A/../N split testing:** These tests compare one or more variations of a single site element or *factor*, such as a promotional offer. Site developers can quickly see which variation of the factor is most persuasive and yields the highest conversion rates. In the simplest case (A/B test), the original version of the interface is served to 50% of the users (A or *Control Group*), and the modified version is served to the other 50% (B or *Treatment Group*[2]). While A/B tests are simple to conduct, they are often not very informative. For instance, consider Figure 1, which depicts the original version and a variant of a checkout example taken from [11].[3] This variant has been obtained by modifying 9 different factors. While an A/B test tells us which of two alternatives is better, it does not yield reliable information on how combinations of the different factors influence the performance of the variant.
- **Multivariate testing:** A multivariate test can be viewed as a combination of many A/B tests, whereby all factors are systematically varied. Multivariate testing extends the effectiveness of online tests by allowing the impact of interactions between factors to be measured. A multivariate test can, e.g., reveal that two interface elements yield an unexpectedly high conversion rate only when they occur together, or that an element that has a positive effect on conversion loses this effect in the presence of other elements.

The execution of a test can be logically separated into two steps, namely (a) the assignment of users to the test, and to one of the subgroups for each of the interfaces to be tested, and (b) the subsequent selection and presentation of this interface to the user. The implementation of online tests partly blurs the two different steps.

The assignment of users to different subgroups is generally randomized, but different methods exist such as:

- **Pseudo-random assignment with caching:** consists in the use of a pseudo-random number generator coupled with some form of caching in order to preserve *consistency* between sessions (i.e., a user should be assigned to the same interface variant on successive visits to the site); and

[2] In reality, the treatment group will only comprise a tiny fraction of the users of a website, so as to keep losses low if the conversion rate of the treatment version should turn out to be poorer than that of the existing version.

[3] Eisenberg reports that Interface A resulted in 90% fewer purchases, probably because potential buyers who had no promotion code were put off by the fact that others could get lower prices.

– **Hash and partitioning:** assigns a unique user identifier that is either stored in a database or in a cookie. The entire set of indentifiers is then partitioned, and each partition is assigned to a variant. This second method is usually preferred due to scalability problems with the first method.

Three implementation methods are being used for the selection and presentation of the interface to the user:

– **Traffic splitting:** In order to generate the different variants, different implementations are created and placed on different physical or virtual servers. Then, by using a proxy or a load balancer which invokes the randomization algorithm, a user's traffic is diverted to the assigned variant.
– **Server-side selection:** All the logic which invokes the randomization algorithm and produces the different variants for users is embedded in the code of the site.
– **Client-side selection:** Assignment and generation of variants is achieved through dynamic modification of each requested page at the client side using JavaScript.

3 Problems with Current Online Test Design and Implementation

The three implementation methods discussed above entail a number of disadvantages, which are a function of the choices made at the architectural level and not of the specific characteristics of an online experiment (such as the chosen OEC or the interface features being modified):

– **Traffic splitting:** Although traffic splitting does not require any changes to the code in order to produce the different user assignments to variants, the implementation of this approach is relatively expensive. The website and the code for the measurement of the OEC have to be replicated n times, where n is the number of tested combinations of different factors (number of possible variants). In addition to the complexity of creating each variant for the test manually by modifying the original website's code (impossible in the case of multivariate tests involving several factors), there is also a problem associated to the hardware required for the execution of the test. If physical servers are used, a fleet of servers will be needed so that each of the variants tested will be hosted on one of them. Likewise, if virtual servers are being used, the amount of system resources required to acommodate the workload will easily exceed the capacity of the physical server, requiring the use of several servers and complicating the supporting infrastructure.
– **Server-side selection:** Extensive code modification is required if interface selection and presentation is performed at the server side. Not only has randomization and user assignment to be embedded in the code, but also a branching logic has to be added in order to produce the different interfaces corresponding to the different combinations of variants. In addition, the code

may become unnecessarily complex, particularly if different combinations of factors are to be considered at the same time when tests are being run concurrently. However, if these problems are solved, server-side selection is a powerful alternative which allows deep modifications to the system and is cheap in terms of supporting infrastructure.
- **Client-side selection:** Although client-side selection is to some extent easier to implement than server-side selection, it suffers from the same shortcomings. In addition, the features subject to experimentation are far more limited (e.g., modifications which go beyond the mere interface are not possible, JavaScript must be enabled in the client browser, execution is error-prone, etc.).

Independent of the chosen form of implementation, substantial support for systematic online experimentation at a framework level is urgently needed. The framework will need to support the definition of the different factors and their possible combinations at the test design stage, and their execution at runtime. Being able to evolve a site safely by keeping track of each of the variants' performance as well as maintaining a record of the different experiments is very desirable when contrasted with the execution of isolated tests on an ad-hoc basis.

4 A Systematic Approach to Online Test Design and Implementation

To overcome the various limitations described in the previous section, we advocate a systematic approach to the development of online experiments. For this purpose, we rely on two different foundations: **(i)** software product lines provide the means to properly model the variability inherent in the design of the experiments, and **(ii)** aspect-oriented software development (AOSD) helps to reduce the effort and cost of implementing the variants of the test by capturing variation factors on aspects. The use of AOSD will also help in presenting variants to users, as well as simplifying user assignment and data collection. By combining these two foundations we aim at supplying developers with the necessary tools to design tests in a systematic manner, enabling the partial automation of variant generation and the complete automation of test deployment and execution.

4.1 Test Design Using Software Product Lines

Software Product Line models describe all requirements or features in the potential variants of a system. In this work, we use a feature-based model similar to the models employed by FODA [13] or FORM [14]. This model takes the form of a lattice of parent-child relationships which is typically quite large. Single systems or variants are then built by selecting a set of features from the model.

Product line models allow the definion of the directly reusable (DR) or mandatory features which are common to all possible variants, and three types of *discriminants* or variation points, namely:

F1(MA) The cart component must include a checkout screen.

- **F1.1(SA)** There must be an additional "Continue Shopping" button present.
  - **F1.1.1(DR)** The button is placed on top of the screen.
  - **F1.1.2(DR)** The button is placed at the bottom of the screen.
- **F1.2(O)** There must be an "Update" button placed under the quantity box.
- **F1.3(SA)** There must be a "Total" present.
  - **F1.3.1(DR)** Text and amount of the "Total" appear in different boxes.
  - **F1.3.2(DR)** Text and amount of the "Total" appear in the same box.
- **F1.4(O)** The screen must provide discount options to the user.
  - **F1.4.1(DR)** There is a "Discount" box present, with amount in a box next to it on top of the "Total" box.
  - **F1.4.2(DR)** There is an "Enter Coupon Code" input box present on top of "Shipping Method".
  - **F1.4.3(DR)** There must be a "Recalculate" button left of "Continue Shopping."

Fig. 2. Feature model fragment corresponding to the checkout screen depicted in Figure 1

- **Single adaptors (SA):** a set of mutually exclusive features from which only one can be chosen when defining a particular system.
- **Multiple adaptors (MA):** a list of alternatives which are not mutually exclusive. At least one must be chosen.
- **Options (O):** a single optional feature that may or may not be included in a system definition.

In order to define the different interface variants that are present in an online test, we specify all common interface features as DR features in a product line model. Varying elements are modeled using discriminants. Different combinations of interface features will result in different interface variants. An example for such a feature model is given in Figure 2, which shows a fragment of a definition of some of the commonalities and discriminants of the two interface variants depicted in Figure 1.

Variants can be manually created by the test designer through the selection of the desired interface features in the feature model, or automatically by generating all the possible combinations of feature selections. Automatic generation is especially interesting in the case of multivariate testing. However, it is worth noting that not all combinations of feature selections need to be valid. For instance, if we intend to generate a variant which includes F1.3.1 in our example, that same selection cannot include F1.3.2 (single adaptor).

Likewise, if F1.4 is selected, it is mandatory to include F1.4.1-F1.4.3 in the selection. These restrictions are introduced by the discriminants used in the product line model. If restrictions are not satisfied, we have generated an invalid

variant that should not be presented to users. Therefore, generating all possible feature combinations for a multivariate test is not enough for our purposes. Fortunately, the feature model can be easily translated into a logical expression by using features as atomic propositions and discriminants as logical connectors.

The logical expression of a feature model is the conjunction of the logical expressions for each of the sub-graphs in the lattice and is achieved using logical AND. If G_i and G_j are the logical expressions for two different sub-graphs, then the logical expression for the lattice is:

$$G_i \wedge G_j$$

Parent-child dependency is expressed using a logical AND as well. If a_i is a parent requirement and a_j is a child requirement such that the selection of a_j is dependent on a_i then $a_i \wedge a_j$. If a_i also has other children $a_k \ldots a_z$ then:

$$a_i \wedge (a_k \wedge \ldots \wedge a_z)$$

The logical expression for a single adaptor discriminant is exclusive OR. If a_i and a_j are features such that a_i is mutually exclusive to a_j then $a_i \oplus a_j$. Multiple adaptor discriminants correspond to logical OR. If a_i and a_j are features such that at least one of them must be chosen then $a_i \vee a_j$. The logical expression for an option discriminant is a bi-conditional[4]. If a_i is the parent of another feature a_j then the relationship between the two features is $a_i \leftrightarrow a_j$.

Feature Model Relation	Formal Definition
Sub-graph	$G_i \wedge G_j$
Dependency	$a_i \wedge a_j$
Single adaptor	$a_i \oplus a_j$
Multiple adaptor	$a_i \vee a_j$
Option	$a_i \leftrightarrow a_j$

Table 1. Feature model relations and equivalent formal definitions

Table 1 summarizes the relationships and logical definitions of the model. The general expression for a product line model is $G_1 \wedge G_2 \wedge \ldots \wedge G_n$ where G_i is $a_i \mathcal{R} a_j \mathcal{R} a_k \mathcal{R} \ \ldots \ \mathcal{R} a_n$ and $\mathcal{R}$ is one of $\wedge, \vee, \oplus$, or $\leftrightarrow$. The logical expression for the checkout example feature model shown in Figure 2 is:

$$\begin{array}{lll} F1 \wedge (& F1.1 \wedge (F1.1.1 \oplus F1.1.2) & \vee \\ & F1.2 & \vee \\ & F1.3 \wedge (F1.3.1 \oplus F1.3.2) & \vee \\ & F1.4 \leftrightarrow (F1.4.1 \wedge F1.4.2 \wedge F1.4.3) &) \end{array}$$

By instantiating all the feature variables in the expression to *true* if selected, and *false* if unselected, we can generate the set of possible variants and then test

[4] $a_i \leftrightarrow a_j$ is true when a_i and a_j have the same value.

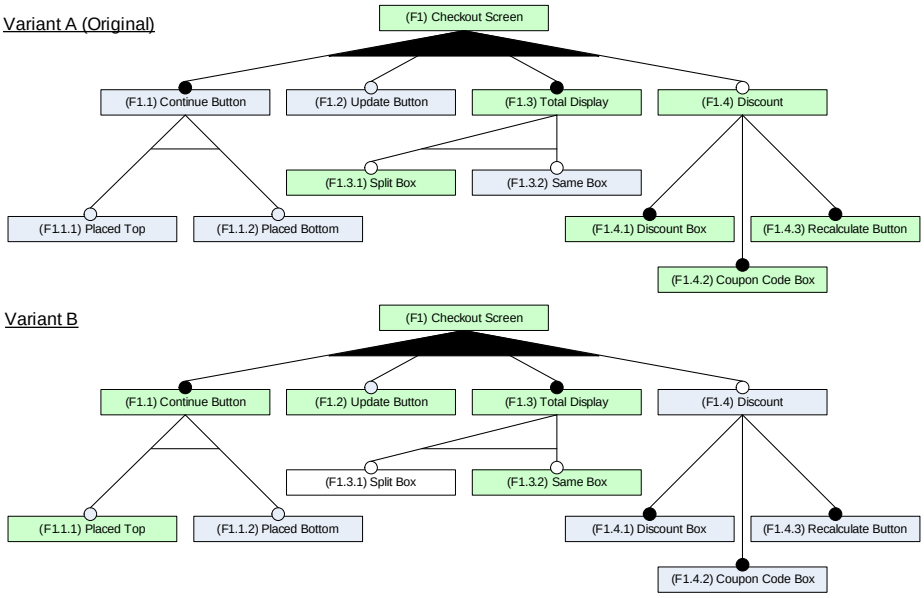

Fig. 3. Feature selections for the generation of variants A and B from Figure 1

their validity using the algorithm described in [21]. A valid variant is one for which the logical expression of the complete feature model evaluates to *true*.

Manual selection can also benefit from this approach since the test administrator can be guided in the process of feature selection by pointing out inconsistencies in the resulting variant as features are selected or unselected. Figure 3 depicts the feature selections for variants A and B of our checkout example. In the feature model, mandatory features are represented with black circles, whereas options are represented with white circles. White triangles express alternative (single adaptors), and black triangles multiple adaptors.

As regards automatic variant generation, we must bear in mind that *full factorial designs* (i.e., testing every possible combination of interface features) provides the greatest amount of information about the individual and joint impacts of the different factors. However, obtaining a statistically meaningful number of cases for this type of experiment takes time, and handling a huge number of variants aggravates this situation. In our approach, the combinatorial explosion in multivariate tests is dealt with by bounding the parts of the hierarchy which descend from an unselected feature. This avoids the generation of all the variations derived from that specific part of the product line.

In addition, our approach does not confine the test designer to a particular selection strategy. It is possible to integrate any optimization method for reducing the complexity of full factorial designs, such as for instance hill climbing strategies like the Taguchi approach [28].

4.2 Case Study: Checkout Screen

Continuing with the checkout screen example described in Section 1, we introduce a simplified implementation of the shopping cart in order to illustrate our approach.

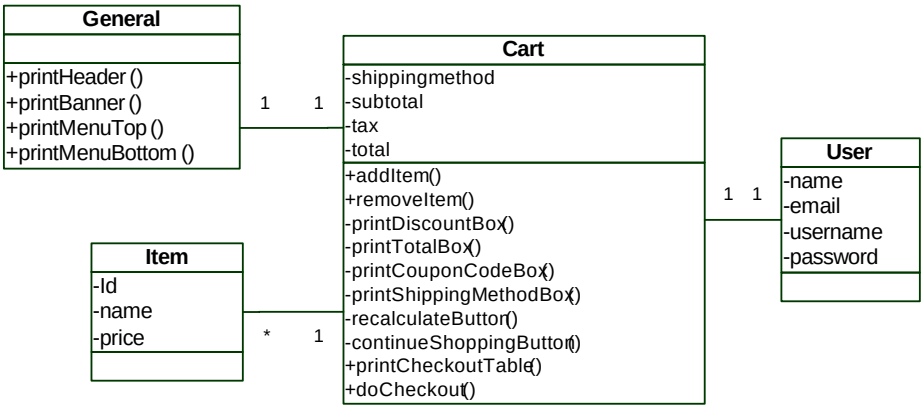

Fig. 4. Classes involved in the shopping cart example

We define a class 'shopping cart' (`Cart`) that allows for the addition and removal of different items (see Figure 4). This class contains a number of methods that render the different elements in the cart at the interface level, such as `printTotalBox()` or `printDiscountBox()`. These are private class methods called from within the public method `printCheckoutTable()`, which is intended to render the main body of our checkout screen. A user's checkout is completed when `doCheckout()` is invoked. On the other hand, the `General` class contains auxiliary functionality, such as representing common elements of the site (e.g., headers, footers and menus).

4.3 Implementing Tests with Aspects

Aspect-Oriented Software Development (AOSD) is based on the idea that systems are better programmed by separately specifying their different concerns (areas of interest), using *aspects* and a description of their relations with the rest of the system. Those specifications are then automatically *woven* (or composed) into a working system. This weaving process can be performed at different stages of the development, ranging from compile-time to run-time (dynamic weaving) [26]. The dynamic approach (Dynamic AOP or d-AOP) implies that the virtual machine or interpreter running the code must be aware of aspects and control the weaving process. This represents a remarkable advantage over static AOP approaches, considering that aspects can be applied and removed at

run-time, modifying application behaviour during the execution of the system in a transparent way.

With conventional programming techniques, programmers have to explicitly call methods available in other component interfaces in order to access their functionality, whereas the AOSD approach offers implicit invocation mechanisms for behavior in code whose writers were unaware of the additional concerns (*obliviousness*). This implicit invocation is achieved by means of *join points*. These are regions in the dynamic control flow of an application (method calls or executions, exception handling, field setting, etc.) which can be intercepted by an aspect-oriented program by using *pointcuts* (predicates which allow the quantification of join points) to match with them. Once a join point has been matched, the program can run the code corresponding to the new behavior (*advices*) typically *before, after, instead* of, or *around* (before and after) the matched join point.

In order to test and illustrate our approach, we use PHP [25], one of the predominant programming languages in Web-based application development. It is an easy to learn language specifically designed for the Web, and has excellent scaling capabilities. Among the variety of AOSD options available for PHP, we have selected phpAspect [4], which is to our knowledge the most mature implementation so far, providing AspectJ[5]-like syntax and abstractions. Although there are other popular languages and platforms available for Web application development (Java Servlets, JSF, etc.), most of them provide similar abstractions and mechanisms. In this sense, our proposal is technology-agnostic and easily adaptable to other platforms.

Aspects are especially suited to overcome many of the issues described in Section 3. They are used for different purposes in our approach that will be described below.

Variant implementation. The different alternatives that have been used so far for variant implementation have important disadvantages, which we discussed in Section 3. These detriments include the need to produce different versions of the system code either by replicating and modifying it across several servers, or using branching logic on the server or client sides.

Using aspects instead of the traditional approaches offers the advantage that the original source code does not need to be modified, since aspects can be applied as needed, resulting in different variants. In our approach, each feature described in the product line is associated to one or more aspects which modify the original system in a particular way. Hence, when a set of features is selected, the appropriate variant is obtained by weaving with the base code[6] the set of aspects associated to the selected features in the variant, modifying the original implementation.

To illustrate how these variations are achieved, consider for instance the features labeled F1.3.1 and F1.3.2 in Figure 2. These two features are mutually

[5] AspectJ [9,15] is the de-facto standard in aspect-oriented programming languages.
[6] That is, the code of the original system.

exclusive and state that in the total box of the checkout screen, text and amount should appear in different boxes rather than in the same box, respectively. In the original implementation (Figure 1.A), text and amount appeared in different boxes, and hence there is no need to modify the behavior if F1.3.1 is selected. When F1.3.2 is selected though, we merely have to replace the behavior that renders the total box (implemented in the method `Cart.printTotalBox()`). We achieve this by associating an appropriate aspect to this feature.

Listing 1 Rendering code replacement aspect.

```
aspect replaceTotalBox{
   pointcut render:exec(Cart::printTotalBox(*));
   around(): render{
      /* Alternative rendering code */
   }
}
```

In Listing 1, by defining a pointcut that intercepts the execution of the total box rendering method, and applying an `around`-type advice, we are able to replace the method through which this particular element is being rendered at the interface.

This approach to the generation of variants results in better code reusability (especially in multivariate testing) as well as reduced costs and efforts, since developers do not have to replicate nor generate complete variant implementations. In addition, this approach is safer and cleaner since the system logic does not have to be temporally (nor manually) modified, thus avoiding the resulting risks in terms of security and reliability.

Finally, not only interface modifications such as the ones depicted in Figure 1, but also backend modifications are easier to perform, since aspect technology allows a behavior to be changed even if it is scattered throughout the system code. The practical implications of using AOP for this purpose can be easily seen in an example. Consider for instance Amazon's recommendation algorithm, which is invoked in many places throughout the website such as its general catalog pages, its shopping cart, etc. Assume that Amazon's development team wonders whether an alternative algorithm that they developed would perform better than the original. With traditional approaches they could modify the source code only by *(i)* replicating the code on a different server and replacing all the calls[7] made to the recommendation algorithm, or *(ii)* including a condition contingent on the variant that is being executed in each call to the algorithm. Using aspects instead enables us to write a simple statement (pointcut) to intercept every call

[7] In the simplest case, only the algorithm's implementation would be replaced. However, modifications on each of the calls may also be required, e.g., due to differences in the signature with respect to the original algorithm's implementation,.

to the recommendation algorithm throughout the site, and replace it with the call to the new algorithm.

Experimenting with variants may require going beyond mere behavior replacement though. This means that any given variant may require for its implementation the modification of data structures or method additions to some classes. Consider for instance a test in which developers want to monitor how customers react to discounts on products in a catalog. Assume that discounts can be different for each product and that the site has not initially been designed to include any information on discounts, i.e., this information needs to be introduced somewhere in the code. To solve this problem we can use inter-type declarations. Aspects can declare members (fields, methods, and constructors) that are owned by other classes. These are called inter-type members. As can be observed in Listing 2, we introduce an additional `discount` field in our `item` class, and also a `getDiscountedPrice()` method which will be used whenever the discounted price of an item is to be retrieved. Note that we need to introduce a new method, because it should still be possible to retrieve the original, non-discounted price.

Listing 2 Item discount inter-type declarations.

```
aspect itemDiscount{
   private Item::$discount;
   public function Item::getDiscountedPrice(){
      return ($this->price - $this->discount);
   }
}
```

Data Collection and User Interaction. The code in charge of measuring and collecting data for the experiment can also be written as aspects in a concise manner. Consider a new experiment with our checkout example in which we want to calculate how much customers spend on average when they visit our site. To this end, we need to add up the amount of money spent on each purchase. One way to implement this functionality is again inter-type declarations.

When the aspect in Listing 3 intercepts the method that completes a purchase (`Cart.doCheckout()`), the associated advice inserts the sales amount into a database that collects the results from the experiment (but only if the execution of the intercepted method succeeds, which is represented by `proceed()` in the advice). It is worth noting that while the database reference belongs to the aspect, the method used to insert the data belongs to the `Cart` class.

Aspects permit the easy and consistent modification of the methods that collect, measure, and synthesize the OEC from the gathered data to be presented to the test administrator in order to be analyzed. Moreover, data collection

Listing 3 Data collection aspect.

```
aspect accountPurchase{
   private $dbtest;
   pointcut commitTrans:exec(Cart::doCheckout(*));
   function Cart::accountPurchase(DBManager $db){
      $db->insert($this->getUserName(),
                  $this->total);
   }
   around($this): commitTrans{
      if (proceed()){
      $this->accountPurchase($thisAspect->dbtest);
      }
   }
}
```

procedures do not need to be replicated across the different variants, since the system will weave this functionality across all of them.

User Assignment. Rather than implementing user assignment in a proxy or load balancer that routes requests to different servers, or including it in the implementation of the base system, we experimented with two different alternatives of aspect-based server-side selection:

- **Dynamic aspect weaving:** A user routing module acts as an entry point to the base system. This module assigns the user to a particular variant by looking up what aspects have to be woven to produce the particular variant to which the current user had been assigned. The module then incorporates these aspects dynamically upon each request received by the server, flexibly producing variants in accordance with the user's assignment. Although this approach is elegant and minimizes storage requirements, it does not scale well. Having to weave a set of aspects (even if they are only a few) on the base system upon each request to the server is very demanding in computational terms, and prone to errors in the process.
- **Static aspect weaving:** The different variants are computed offline, and each of them is uploaded to the server. In this case the routing module just forwards the user to the corresponding variant stored on the server (the base system is treated just like another variant for the purpose of the experiment). This method does not slow down the operation of the server and is a much more robust approach to the problem. The only downside of this alternative is that the code corresponding to the different variants has to be stored temporarily on the server (although this is a minor inconvenience since usually the amount of space required is negligible compared to the average server storage capacity). Furthermore, this alternative is cheaper than traffic splitting, since it does not require the use of a fleet of servers

nor the modification of the system's logic. This approach still allows one to spread the different variants across several servers in case of high traffic load.

5 Tool Support

The approach for online experiments on websites that we presented in this article has been implemented in a prototype tool, called WebLoom. It includes a graphical user interface, to build and visualize feature models that can be used as the structure upon which controlled experiments on a website can be defined. In addition, the user can write aspect code which can be attached to the different features. Once the feature model and associated code have been built, the tool supports both automatic and manual variant generation, and is able to deploy aspect code which lays out all the necessary infrastructure to perform the designed test on a particular website. The prototype has been implemented in Python, using the wxWidgets toolkit technology for the development of the user interface. It both imports and exports simple feature models described in an XML format specific to the tool.

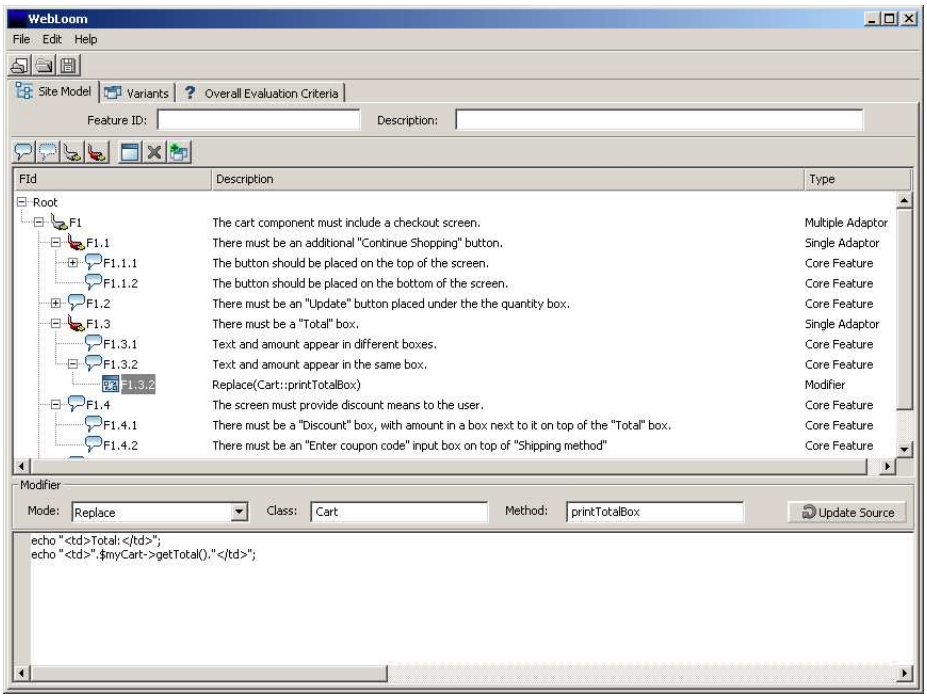

Fig. 5. WebLoom displaying the product line model depicted in Figure 2

The prototype tool's graphical user interface is divided into three main working areas:

- **Feature model.** This is the main working area where the feature model can be specified (see Figure 5). It includes a toolbar for the creation and modification of discriminants and a code editor for associated modifications. This area also allows the selection of features in order to generate variants.
- **Variant management.** Variants generated on the site model area can be added or removed from the current test, renamed or inspected. A compilation of the description of all features contained in a variant is automatically presented to the user based on feature selections when the variant is selected (Figure 6, bottom).

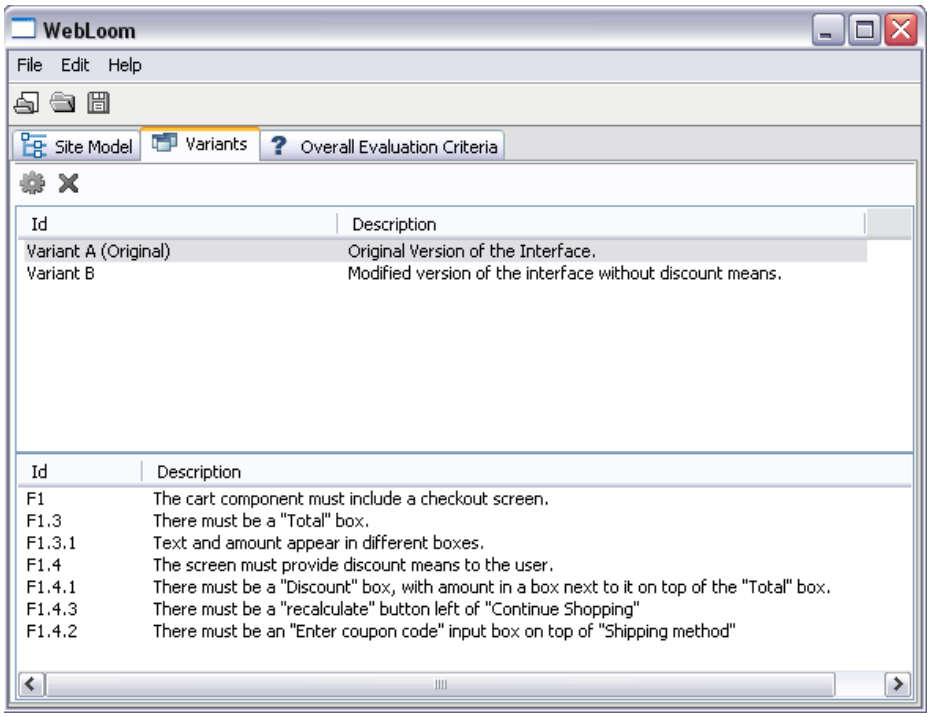

Fig. 6. Variant management screen in WebLoom

- **Overall Estimation Criteria.** One or more OEC to measure on the experiments can be defined in this section. Each of the OEC are labeled in order to be identified later on, and the associated code for gathering and processing data is directly defined by the test administrator.

In Figure 7, we can observe the interaction with our prototype tool. The user enters a description of the potential modifications to be performed on the web-

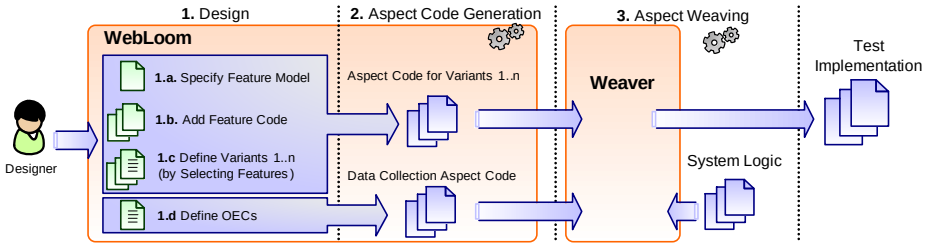

Fig. 7. Operation of WebLoom

site, in order to produce the different variants under WebLoom's guidance. This results in a basic feature model structure, which is then enriched with code associated to the aforementioned modifications (aspects). Once the feature model is complete, the user can freely select a number of features using the interface, and take snapshots of the current selections in order to generate variants. These variants are automatically checked for validity before being incorporated into the variant collection. Alternatively, the user can ask the tool to generate all the valid variants for the current feature model and then remove the ones which are not interesting for the experiment.

Once all necessary input has been received, the tool gathers the code for each particular variant to be tested in the experiment, by collecting all the aspects associated with the features that were selected for the variant. It then invokes the weaver to produce the actual variant code for the designed test by weaving the original system code with the collection of aspects produced by the tool.

6 Related Work

Software product lines and feature-oriented design and programming have already been successfully applied in the development of Web applications, to significantly boost productivity by exploiting commonalities and reusing as many assets (including code) as possible. For instance, Trujillo et al. [29] present a case study of Feature Oriented Model Driven Design (FOMDD) on a product line of portlets (Web portal components). In this work, the authors expressed variations in portlet functionality as features, and synthesized portlet specifications by composing them conveniently. Likewise, Petersson and Jarzabek [24] present an industrial case study in which their reuse technique XVCL was incrementally applied to generate a Web architecture from the initial code base of a Web portal. The authors describe the process that led to the development of the Web Portal product line.

Likewise, aspect-oriented software development has been previously applied to the development of Web applications. Valderas et al. present in [31] an approach for dealing with crosscutting concerns in Web applications from requirements to design. Their approach aims at decoupling requirements that belong

to different concerns. These are separately modeled and specified using the task-based notation, and later integrated into a unified requirements model that is the source of a model-to-model and model-to-code generation process yielding Web application prototypes that are built from task descriptions.

Although the aforementioned approaches meet their purpose of boosting productivity by taking advantage of commonalities, and of easing maintenance by properly encapsulating crosscutting concerns, they do not jointly exploit the advantages of both approaches. Moreover, although they are situated in the context of Web application development, they are not well suited to the specific characteristics of online test design and implementation which have been described in previous sections.

The idea of combining software product lines and aspect-oriented software development techniques does already have some tradition in software engineering. In fact, Lee et al. [18] present some guidelines on how feature-oriented analysis and aspects can be combined. Likewise, Loughran and Rashid [19] propose framed aspects as a technique and methodology that combines AOSD, frame technology, and feature-oriented domain analysis in order to provide a framework for implementing fine-grained variability. In [20], they extend this work to support product line evolution using this technique. Other approaches such as [32] aim at implementing variability, and the management and tracing of requirements for implementation by integrating model-driven and aspect-oriented software development. The AMPLE project [1] takes this approach one step further along the software lifecycle and maintenance, aiming at traceability during product line evolution. In the particular context of Web applications, Alférez and Suesaowaluk [8] introduce an aspect-oriented product line framework to support the development of software product lines of Web applications. This framework is similarly aimed at identifying, specifying, and managing variability from requirements to implementation.

Although both the aforementioned approaches and our own proposal employ software product lines and aspects, there is a key difference in the way these elements are used. First, the earlier approaches are concerned with the general process of system construction by identifying and reusing aspect-oriented components, whereas our approach deals with the specific problem of online test design and implementation, where different versions of a Web application with a limited lifespan are generated to test user behavioral response. Hence, our framework is intended to generate lightweight aspects which are used as a convenient means for the transient modification of parts of the system. In this sense, it is worth noting that system and test designs and implementations are completely independent of each other, and that aspects are only involved as a means to generate system variants, but not necessarily present in the original system design. In addition, our approach provides automatic support for the generation of all valid variants within the product line, and does not require the modification of the underlying system which stays online throughout the whole online test process.

To the extent of our knowledge, no research has so far been reported on treating online test design and implementation in a systematic manner. A number of consulting firms already specialized on analyzing companies' Web presence [2,6,3]. These firms offer ad-hoc studies of Web retail sites with the goal of achieving higher conversion rates. Some of them use proprietary technology that is usually focused on the statistical aspects of the experiments, requiring significant code refactoring for test implementation[8].

Finally, SiteSpect [5] is a software package which takes a proxy-based approach to online testing. When a Web client makes a request to the Web server, it is first received by the software and then forwarded to the server (this is used to track user behavior). Likewise, responses with content are also routed through the software, which injects the HTML code modifications and forwards the modified responses to the client. Although the manufacturers claim that it does not matter whether content is generated dynamically or statically by the server since modifications are performed by replacing pieces of the generated HTML code, we find this approach adequate for trivial changes to a site only, and not very suitable for user data collection and measurement. Moreover, no modifications can be applied to the logic of the application. These shortcomings severely impair this method which is not able to go beyond simple visual changes to the site.

7 Concluding Remarks

In this paper, we presented a novel and systematic approach to the development of controlled online tests for the effects of webpage variants on users, based on software product lines and aspect oriented software development. We also described how the drawbacks of traditional approaches, such as high costs and development effort, can be overcome with our approach. We believe that its benefits are especially valuable for the specific problem domain that we address. On one hand, testing is performed on a regular basis for websites in order to continuously improve their conversion rates. On the other hand, a very high percentage of the tested modifications are usually discarded since they do not improve the site performance. As a consequence, a lot of effort is lost in the process. We believe that WebLoom will save Web developers time and effort by reducing the amount of work they have to put into the design and implementation of online tests.

Although there is a wide range of choices available for the implementation of Web systems, our approach is technology-agnostic and most likely deployable to different platforms and languages. However, we observed that in order to fully exploit the benefits of this approach, a website should first be tested whether its implementation meets the modularity principle. This is of special interest at the presentation layer, where user interface component placement, user interface

[8] It is however not easy to thoroughly compare these techniques from an implementation point of view, since firms tend to be quite secretive about them.

style elements, event declarations and application logic traditionally tend to be mixed up [23].

Regarding future work, a first perspective aims at enhancing our basic prototype with additional WYSIWYG extensions for its graphical user interface. Specifically, developers should be enabled to immediately see the effects that code modifications and feature selections will have on the appearance of their website. This is intended to help them deal with variant generation in a more effective and intuitive manner. A second perspective is refining the variant validation process so that variation points in feature models that are likely to cause significant design variations can be identified, thus reducing the variability.

References

1. Ample project. http://www.ample-project.net/.
2. Offermatica. http://www.offermatica.com/.
3. Optimost. http://www.optimost.com/.
4. phpAspect: Aspect oriented programming for PHP. http://phpaspect.org/.
5. Sitespect. http://www.sitespect.com.
6. Vertster. http://www.vertster.com/.
7. *Software product lines: practices and patterns*. Addison-Wesley Longman Publishing Co, Boston, MA, USA, 2001.
8. G.H. Alférez and Poonphon Suesaowaluk. An aspect-oriented product line framework to support the development of software product lines of web applications. In *SEARCC '07: Proceedings of the 2nd South East Asia Regional Computer Conference*, 2007.
9. Adrian Colyer, Andy Clement, George Harley, and Matthew Webster. *Eclipse AspectJ: Aspect-Oriented Programming with AspectJ and the Eclipse AspectJ Development Tools*. Pearson Education, Upper Saddle River, NJ, 2005.
10. Bryan Eisenberg. How to decrease sales by 90 percent. Available at: http://www.clickz.com/1588161.
11. Bryan Eisenberg. How to increase conversion rate 1,000 percent. Available at: http://www.clickz.com/showPage.html?page=1756031.
12. Robert E. Filman, Tzilla Elrad, Siobhan Clarke, and Mehmet Aksit, editors. *Aspect-Oriented Software Development*. Adisson-Wesley, 2004.
13. K. Kang, S. Cohen, J. Hess, W. Novak, and S. Peterson. Feature-oriented domain analysis (FODA) feasibility study. Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University, November 1990.
14. Kyo Chul Kang, Sajoong Kim, Jaejoon Lee, Kijoo Kim, Euiseob Shin, and Moonhang Huh. FORM: A feature-oriented reuse method with domain-specific reference architectures. *Ann. Software Eng*, 5:143–168, 1998.
15. Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An Overview of AspectJ. In Jørgen Lindskov Knudsen, editor, *ECOOP 2001 Object-Oriented Programming*, volume 2072 of *Lecture Notes in Computer Science*, pages 327–353, 2001.
16. Ron Kohavi, Randal M. Henne, and Dan Sommerfield. Practical Guide to Controlled Experiments on the Web: Listen to your Customers not to the HIPPO. In Pavel Berkhin, Rich Caruana, and Xindong Wu, editors, *Proceedings of the 13th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, San Jose, California, USA, 2007*, pages 959–967. ACM, 2007.

17. Ron Kohavi and Matt Round. Front Line Internet Analytics at Amazon.com, 2004. Available at: http://ai.stanford.edu/ ronnyk/emetricsAmazon.pdf.
18. Kwanwoo Lee, Kyo C. Kang, Minseong Kim, and Sooyong Park. Combining feature-oriented analysis and aspect-oriented programming for product line asset development. In *SPLC '06: Proceedings of the 10th International on Software Product Line Conference*, pages 103–112, Washington, DC, USA, 2006. IEEE Computer Society.
19. Neil Loughran and Awais Rashid. Framed aspects: Supporting variability and configurability for AOP. In Jan Bosch and Charles Krueger, editors, *Software Reuse: Methods, Techniques and Tools. 8th International Conference, ICSR 2004, Madrid, Spain*, volume 3107 of *Lecture Notes in Computer Science*, pages 127–140. Springer, 2004.
20. Neil Loughran, Awais Rashid, Weishan Zhang, and Stan Jarzabek. Supporting product line evolution with framed aspects. In David H. Lorenz and Yvonne Coady, editors, *ACP4IS: Aspects, Components, and Patterns for Infrastructure Software*, pages 22–26, March.
21. Mike Mannion and Javier Cámara. Theorem proving for product line model verification. In Frank van der Linden, editor, *Software Product-Family Engineering: 5th International Workshop, PFE 2003, Siena, Italy*, volume 3014 of *Lecture Notes in Computer Science*, pages 211–224, Siena, Italy, 2003. Springer.
22. Flint McGlaughlin, Brian Alt, and Nick Usborne. The power of small changes tested, 2006. Available at: http://www.marketingexperiments.com/improving-website-conversion/power-small-change.html.
23. Tommi Mikkonen and Antero Taivalsaari. Web applications – spaghetti code for the 21st century. In Walter Dosch, Roger Y. Lee, Petr Tuma, and Thierry Coupaye, editors, *Proceedings of the 6th ACIS International Conference on Software Engineering Research, Management and Applications, SERA 2008, Prague, Czech Republic*, pages 319–328. IEEE Computer Society, 2008.
24. Ulf Pettersson and Stan Jarzabek. Industrial experience with building a web portal product line using a lightweight, reactive approach. In Michel Wermelinger and Harald Gall, editors, *Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering, Lisbon, Portugal, 2005*, pages 326–335. ACM, 2005.
25. PHP: Hypertext preprocessor. http://www.php.net/.
26. Andrei Popovici, Andreas Frei, and Gustavo Alonso. A Proactive Middleware Platform for Mobile Computing. In Markus Endler and Douglas Schmidt, editors, *Middleware 2003: ACM/IFIP/USENIX International Middleware Conference Rio de Janeiro, Brazil*, Lecture Notes in Computer Science. Springer, 2003.
27. Sumantra Roy. 10 Factors to Test that Could Increase the Conversion Rate of your Landing Pages, 2007. Available at: http://www.wilsonweb.com/conversion/sumantra-landing-pages.htm.
28. Genichi Taguchi. The role of quality engineering (Taguchi Methods) in developing automatic flexible manufacturing systems. In *Proceedings of the Japan/USA Flexible Automation Symposium, Kyoto, Japan, July 9-13*, pages 883–86, 1990.
29. Salvador Trujillo, Don S. Batory, and Oscar Díaz. Feature oriented model driven development: A case study for portlets. In *Proceedings of the 30th International Conference on Software Engineering (ICSE'07), Leipzig, Germany*, pages 44–53. IEEE Computer Society, 2007.
30. Nick Usborne. Design choices can cripple a website, 2005. Available at: http://alistapart.com/articles/designcancripple.

31. Pedro Valderas, Vicente Pelechano, Gustavo Rossi, and Silvia E. Gordillo. From crosscutting concerns to web systems models. In Boualem Benatallah, Fabio Casati, Dimitrios Georgakopoulos, Claudio Bartolini, Wasim Sadiq, and Claude Godart, editors, *Proceedings of Web Information Systems Engineering - WISE 2007, 8th International Conference on Web Information Systems Engineering, Nancy, France*, volume 4831 of *Lecture Notes in Computer Science*, pages 573–582. Springer, 2007.
32. Markus Voelter and Iris Groher. Product line implementation using aspect-oriented and model-driven software development. In *SPLC '07: Proceedings of the 11th International Software Product Line Conference*, pages 233–242, Washington, DC, USA, 2007. IEEE Computer Society.