Facilitating Controlled Tests of Website Design Changes: a Systematic Approach

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 web site 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 [9,8] revealed that small changes at the user interface can cause surprisingly large differences in the amount of purchases made, and 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 [18,23,21]. For this reason, several techniques have been developed by industry that use actual user behavior to measure the benefits of design modifications [14]. 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 and practical lessons learned from such experiments have been well described [13], there is little systematic support to their design and implementation. In this work, we advocate a systematic approach to the design and implementation of such experiments based on *Software Product Lines* [5] and *Aspect Oriented Software Development* (AOSD) [10]. Section 2 overviews the different techniques involved in online tests and points out their shortcomings. Section 3 describes our approach, briefly introducing software product lines and AOSD. Section 4

introduces a prototype tool that we developed to test the feasibility of our approach. Section 5 compares our proposal with related work, and Section 6 presents some conclusions and perspectives.

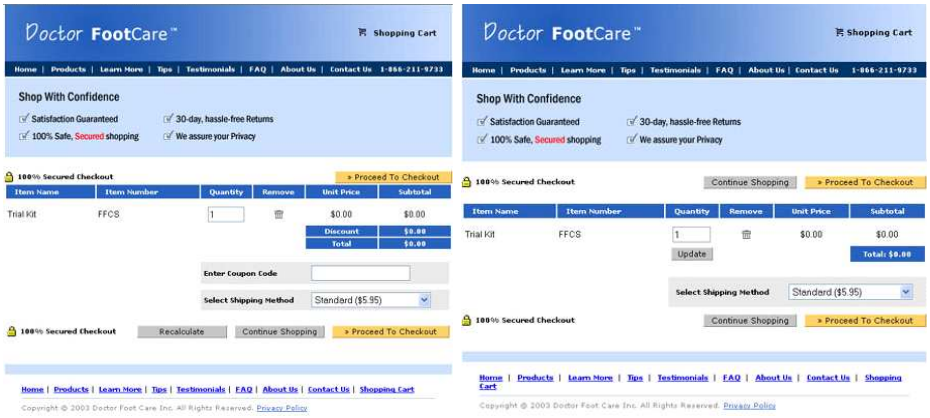

Fig. 1. Checkout screen: variants A (original, left) and B (modified, right)[2]

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 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: **(i) 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 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*). A/B tests are simple, but not very informative. For instance, consider Figure 1, which depicts the original version and a variant of a checkout example taken from [9][1]. 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. **(ii) Multivariate Testing** . A multivariate test can

[1] 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.

[2] © 2007 ACM, Inc. Included by permission.

be viewed as a combination of many A/B tests, whereby all factors are systematically varied. This 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. Specifically, three implementation methods are currently used: **(i) Traffic Splitting.** Different implementations (variants) are manually created and placed on different servers. Then, user traffic is diverted to the assigned variant using a proxy. This approach is expensive, and both website and the code for the measurement of the OEC have to be replicated across (virtual) servers. Moreover, creating each variant for the test manually is impossible in most multivariate tests. **(ii) Server-side Selection.** The logic that produces the different variants for users is embedded in the code of the site. In particular, branching logic has to be added to produce the different interfaces. Code becomes complex and unmanageable if different tests are run concurrently. However, if these problems are solved, server-side selection is a powerful alternative which has the potential to automate variant generation. **(iii) Client-side Selection.** Assignment and generation of variants is achieved through dynamic modification of each requested page at the client side using JavaScript. The drawbacks of this approach are similar to the ones in server-side selection, but in addition, the features subject to experimentation are far more limited (e.g., only superficial modifications are possible, JavaScript must be enabled in the client browser, etc.).

3 Systematic 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.

3.1 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 [11] or FORM [12]. 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 directly reusable (DR) features which are common to all possible variants, and three types of *discriminants* or variation points, namely: **(i) Single adaptors (SA):** a set of mutually exclusive features; **(ii) Multiple adaptors (MA):** a list of alternatives not mutually exclusive (at least one must be selected); and **(iii) Options (O):** a single optional feature.

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 in Figure 1

To define the different interface variants in an online test, we specify common interface features as DR in the feature model. Varying elements are modeled using discriminants. Different combinations of interface features result in different variants. A fragment of such a feature model for our example is given in Figure 2. 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, not all combinations of feature selections are valid. For instance, a single feature selection cannot include both F1.3.1 and 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 feature model. If restrictions are not satisfied, the variant is not valid and should not be presented to users. Feature models can be translated into a logical expression by using features as atomic propositions and discriminants as logical connectors. 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 their validity [17]. A valid variant is one for which the logical expression of the complete feature model evaluates to *true*.

3.2 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.

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 achieved by means of *join points*. These are regions in the dynamic control flow of an application (method calls or executions, 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. When a join point is matched, the program runs code implementing new behavior (*advices*) typically *before, after, instead* of, or *around* (before

and after) the matched join point. To illustrate our approach, we use PHP [20] and phpAspect [3], which provides AspectJ[3] -like syntax and abstractions. However, our proposal is easily adaptable to other platforms.

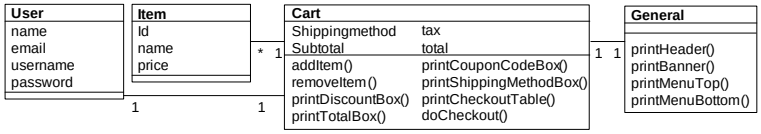

Fig. 3. Classes involved in the shopping cart example

We introduce a simplified implementation of the shopping cart in Section 1 to illustrate our approach: a 'shopping cart' class (`Cart`) allows for the addition and removal of different items. This class contains a number of methods that render the different elements in the cart at the interface, such as `printTotalBox` or `printDiscountBox`. These are private methods called from within the public method `printCheckout-Table`, used to render the main body of our checkout screen. A user's checkout is completed when `doCheckout` is invoked. The `General` class contains auxiliary functions, such as representing common elements of the site (e.g., headers, footers and menus).

Variant implementation. The alternatives used so far for variant implementation have important disadvantages (discussed in Section 2). These 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* (*i.e.,* the original system's code) the set of aspects associated to the selected features in the variant.

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 exclusive and state that in the total box of the checkout screen, text and amount should appear in different boxes rather than the same box, respectively. In the original implementation (Figure 1.A), text and amount appeared in different boxes, 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 method that renders the total box. This is achieved by adding the aspect in Figure 4.A, which defines a pointcut intercepting the execution of `Cart.printTotalBox` and applies an `around`-type advice.

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. Moreover, this approach is safer and cleaner, because the system logic does not have to be temporally (nor manually) modified, with the risks this represents in terms of security and reliability.

[3] AspectJ [7] is the de-facto standard in aspect-oriented programming languages.

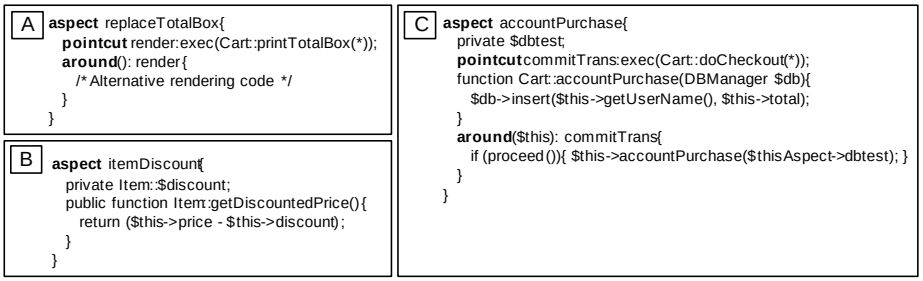

Fig. 4. Aspects: (A) rendering code replacement; (B) item discount inter-type declarations; and (C) data collection

Experimenting with variants may also require 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, etc.) owned by other classes. These are called inter-type members. The aspect on Figure 4.B, introduces an additional `discount` field in our `Item` class, and also a `getDiscountedPrice` method used when the discounted price of an item is to be retrieved.

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 Figure 4.C intercepts the method `Cart.doCheckout` that completes a purchase, 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.

4 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. Moreover, the user can attach aspect code to features. The tool also supports both automatic and manual variant generation, and is able to deploy code which lays out all the necessary infrastructure to perform the designed test on a particular website.

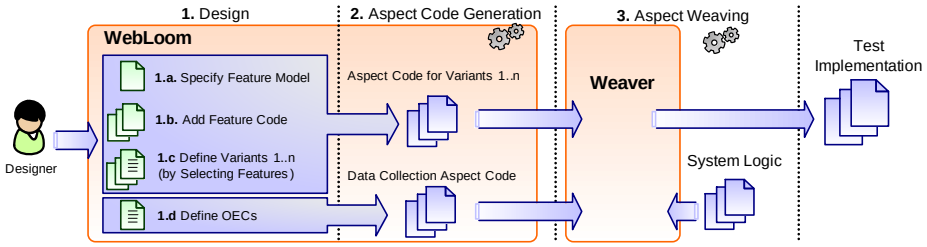

Fig. 5. Operation of WebLoom

In Figure 5 we can observe the way in which our prototype tool works. The user enters a description of the potential modifications to be performed on the website 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 select features to generate any number of variants, which are automatically checked for validity before being stored. Alternatively, the user can ask the tool to generate all the valid variants for the current feature model. 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 aspect code produced by the tool.

5 Related Work

Feature models and AOSD have already been applied in the construction of Web applications in order to achieve significant productivity gains [22,19]. However, these proposals only exploit one of these alternatives and do not pursue a combined approach.

Regarding the combined use of both approaches, Lee et al. [15] and Loughran and Rashid [16] present some guidelines on how feature-oriented analysis and aspects can be combined. Other approaches such as [24] aim at implementing variability, and the management and tracing of requirements to implementation by integrating model-driven and aspect-oriented software development. The AMPLE project [1] takes this approach along the software lifecycle, aiming at traceability during product line evolution. Although both combination approaches and our own proposal employ software product lines and aspects, 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 creation of different versions of a Web application with a limited lifespan 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 noticing that aspects are only involved as a means to generate system variants, but not necessarily present in the original system implementation.

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 [4,2]. These firms offer ad-hoc studies of web retail sites with the goal of achieving higher conversion rates. Some of them use proprietary technology usually focused on the statistical aspects of the experiments, requiring significant code refactoring for test implementation.

6 Concluding Remarks

We believe that the benefits of our approach are especially valuable for the problem domain that we address. On one hand, testing is performed on a regular basis for websites to continuously improve their conversion rates. On the other hand, a high percentage of the tested modifications are discarded since they do not improve the site's performance. Therefore, a lot of effort is lost in the process. We believe that WebLoom will save developers time and effort, reducing the amount of work they have to put into the design and implementation of online tests.

A more detailed description of our work can be found in [6]. Regarding future work, we aim 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 web site.

References

1. Ample project. http://www.ample-project.net/.
2. Optimost. http://www.optimost.com/.
3. phpAspect: Aspect oriented programming for PHP. http://phpaspect.org/.
4. Vertster. http://www.vertster.com/.
5. *Software product lines: practices and patterns*. Addison-Wesley Longman Publishing Co, Boston, MA, USA, 2001.
6. J. Cámara, and A. Kobsa. *Facilitating Controlled Tests of Website Design Changes using Aspect Oriented Programming and Software Product Lines*. Transactions on Large Scale Data and Knowledge Centered Systems 1(1). Springer, 2009.
7. A. Colyer, A. Clement, G. Harley, and M. Webster. *Eclipse AspectJ: Aspect-Oriented Programming with AspectJ and the Eclipse AspectJ Development Tools*. Pearson Education, 2005.
8. B. Eisenberg. How to decrease sales by 90 percent. http://www.clickz.com/1588161.
9. B. Eisenberg. How to increase conversion rate 1,000 percent. http://www.clickz.com/showPage.html?page=1756031.
10. R.E. Filman, T. Elrad, S. Clarke, and M. Aksit, editors. *Aspect-Oriented Software Development*. Adisson-Wesley, 2004.
11. K. Kang, S. Cohen, J. Hess, W. Novak, and S. Peterson. Feature-oriented domain analysis (FODA) feasibility study. TR. CMU/SEI-90-TR-21, SEI, 1990.
12. K.C. Kang, S. Kim, J. Lee, K. Kim, E. Shin, and M. Huh. FORM: A feature-oriented reuse method with domain-specific reference architectures. *Ann. Software Eng*, 5, 1998.
13. R. Kohavi, R.M. Henne, and D. Sommerfield. Practical guide to controlled experiments on the web: listen to your customers not to the hippo. In *Proc. of KDD '07*. ACM, 2007.
14. R. Kohavi and M. Round. Front Line Internet Analytics at Amazon.com, 2004. http://ai.stanford.edu/∼ronnyk/emetricsAmazon.pdf.
15. K. Lee, K.C. Kang, M. Kim, and S. Park. Combining feature-oriented analysis and aspect-oriented programming for product line asset development. In Proc. of *SPLC '06*. IEEE, 2006.
16. N. Loughran and A. Rashid. Framed aspects: Supporting variability and configurability for AOP. In *ICSR*, LNCS 3107. Springer, 2004.
17. M. Mannion and J. Cámara. Theorem proving for product line model verification. In *Proc. of PFE-5*, LNCS 3014. Springer, 2004.
18. F. McGlaughlin, B. Alt, and N. Usborne. The power of small changes tested, 2006. http://www.marketingexperiments.com/improving-website-conversion/power-small-change.html.
19. U. Pettersson and S. Jarzabek. Industrial experience with building a web portal product line using a lightweight, reactive approach. In *Proc. of ESEC/SIGSOFT FSE*. ACM, 2005.
20. PHP: Hypertext preprocessor. http://www.php.net/.
21. S. Roy. 10 factors to test that could increase the conversion rate of your landing pages, 2007. http://www.wilsonweb.com/conversion/sumantra-landing-pages.htm.
22. S. Trujillo, D.S. Batory, and O. Díaz. Feature oriented model driven development: A case study for portlets. In *ICSE*, pages 44–53. IEEE, 2007.
23. N. Usborne. Design choices can cripple a website, 2005. http://alistapart.com/articles/designcancripple.
24. M. Voelter and I. Groher. Product line implementation using aspect-oriented and model-driven software development. In *Proc. of SPLC '07*. IEEE, 2007.