Refactoring for Usability in Web Applications

Alejandra Garrido and Gustavo Rossi,
Universidad Nacional de La Plata

Damiano Distante, Unitelma Sapienza University

// Refactoring a Web application's design structure can improve its usability. Characterizing each refactoring according to the usability factor it improves and the bad usability smells it targets can further clarify its intent. //

WILLIAM OPDYKE AND Ralph Johnson introduced refactoring in the early 1990s, mainly for restructuring an object-oriented design's class hierarchy.[1] A few years later, Martin Fowler popularized it, defining refactoring as "a change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behavior."[2] Since then, refactoring has been applied to different software artifacts, such as Unified Modeling Language (UML) models,[3] databases,[4] and HTML documents.[5]

Refactoring's basic philosophy—that each refactoring is a small behavior-preserving transformation—has remained the same, but its intent has varied considerably from the original purpose of improving source code readability, extensibility, and maintainability. For example, an HTML refactoring, such as "Turn on autocomplete,"[5] doesn't improve any internal code qualities, but it does make a Web form easier to use and thereby shifts the refactoring's intent toward improving software product usability. A similar case occurs in refactoring APIs, which are intended for external users and so essentially involve external attributes.

We believe it's important to link each refactoring not only to the "bad smells" it can eliminate—borrowing Kent Beck and Fowler's metaphor for symptoms of code problems—but also to the specific quality attributes it aims to improve. To illustrate our claim, we present some refactorings intended to improve usability in Web applications. In general, Web applications are defined by three models corresponding to three design layers: content, navigation, and presentation.[6] Elsewhere, we presented a catalog of usability refactorings for the navigation and presentation models of the Object Oriented Hypermedia Design Method for Web design.[7] The catalog's intent was to improve the usability of applications derived from those models. In this article, we stress the characterization of refactorings according to the design model they apply to and their specific intent. By classifying navigation and presentation refactorings, we can target each model's specific attributes and potential bad smells. We also generalize their mechanics to stereotyped UML diagrams.

Refactoring Web Software for Usability

Refactorings that aren't intended to improve the code's *internal* quality attributes include refactorings to improve database performance[4] or HTML accessibility.[5] These examples indicate a development trend toward applying

0740-7459/11/$26.00 © 2011 IEEE

refactoring to *external* quality attributes. We follow this trend in defining refactorings for Web application models to improve an application's usability without changing its content and functionality.[7]

Let's first consider an example. Figure 1 compares two Facebook interfaces, illustrating a major redesign the company did in 2008. Many widgets are moved around, but let's focus on the tabs designed to ease navigation.

In the original interface (Figure 1a), the homepage of a given account shows personal information, a list of friends, and a mini-feed of news about the account owner and her friends, among other things. The page is cluttered with information and links of different kinds, which require repeated scrolling. In the newer interface (Figure 1b), the homepage has been split into different pages with a new tab row that allows navigation between them. Focusing on this particular change, we can see that the application's basic behavior is preserved. It still supports browsing the same information and accessing the same functionalities. However, the user will perceive a change in the graphical interface and navigation structure, which now balance the weight of the different kinds of information and provide some breathing room for each kind. We catalogued this Web refactoring under the name "Split page."[7]

FIGURE 1. Comparison of Facebook interfaces: (a) original homepage, (b) restructured with tabs. Both interfaces provide the same functionality, but the organization of content is better in (b).

At the implementation level, the mechanics of this refactoring involve listing every change in the source code in charge of building the homepage front end. For example, if we suppose that the site is based on HTML code, implementing this refactoring would entail the following actions:

1. Create new HTML pages, split the original homepage into them, and distribute the HTML code associated with the information, links, and widgets available in the original homepage.
2. To each new page, add the HTML code to implement the tab row and the associated functionalities, which include enabling navigation between the new pages, indicating the current page by distinguishing the active tab from the others, and announcing each page's content by using a meaningful label or icon for each tab.

However, if the site generates the HTML pages using a different technology, such as Flash or a Web application framework, the mechanics will differ—in the same way traditional code refactorings depend on the target language. This is one advantage of defining the mechanics at a model level, where they can be generalized as model refactorings on UML-like diagrams. Additionally, Web design models provide a more abstract picture of the application and its different aspects, supporting informed decisions for synchronizing the changes in the different design layers. For example, splitting a page might motivate splitting a table in the database that holds that page's content. Finally, we see model-driven development (MDD) becoming more of a reality as more tools become available to generate code from Web models (for example, UWE4JSF at http://uwe.pst.ifi.lmu.de/toolUWE4JSF.html). In this case, the tool supporting the MDD process would derive the transformed HTML code automatically from the refactored models.

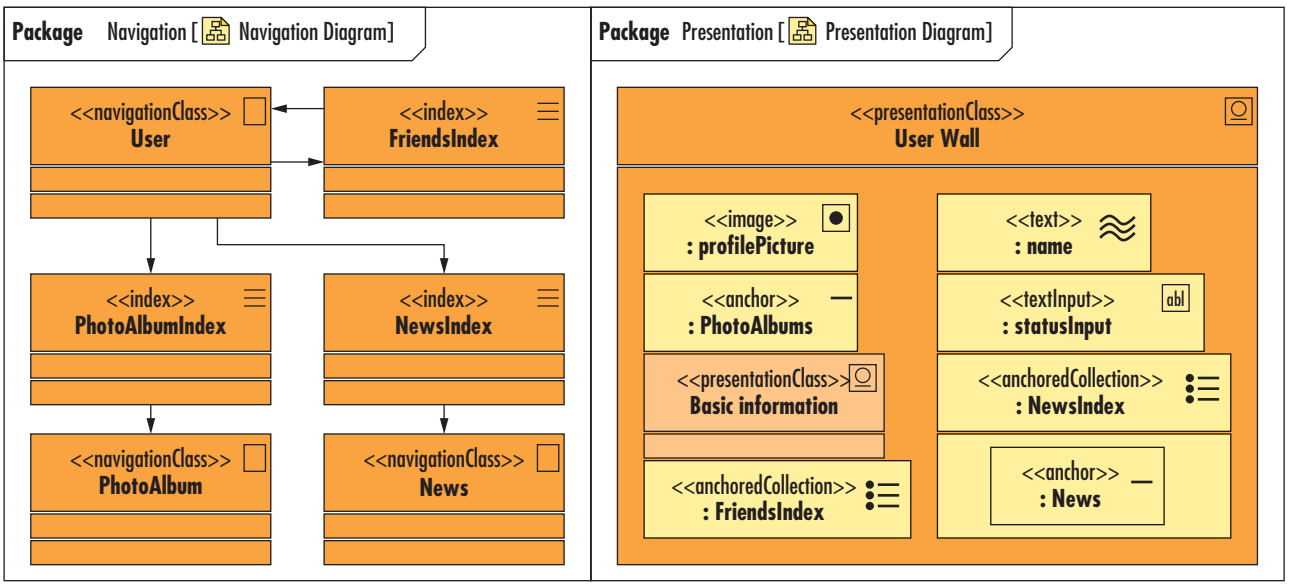


FIGURE 2. Navigation and presentation model diagrams. Names enclosed by guillemets and icons shown in each box distinguish UML stereotypes. For example, the stereotype <<navigationClass>> represents a navigation node, and the stereotype <<anchor>> represents the means for activating a link.

Characterizing Web Model Refactorings

In the three design models that most Web engineering methodologies support,[6] the content model defines the types, attributes, and relationships of the application contents and associated behaviors. Refactorings that apply over this model are traditional refactorings intended to improve internal quality factors.

The navigation model maps the content model's classes to navigation nodes (units of information and behavior perceived by the user) and its associations between content classes to navigation links. Moreover, it organizes the navigation space by mapping associations with a multiplicity greater than one to navigation indexes.

The presentation model defines the Web application's abstract user interface, which is mainly a collection of pages with their components: widgets that show node attributes, those that trigger node operations, and anchors for navigating over links. It's abstract because it doesn't specify the exact position of widgets, nor their graphical attributes, but just their type.

We can represent both the navigation and presentation models with stereotyped UML class diagrams.[8] Figure 2 shows simplified navigation and presentation diagrams for the Facebook homepage presented in Figure 1. Classes in the navigation diagram represent nodes or indexes, and classes in the presentation diagram represent pages. For space reasons, we omitted navigation class attributes and show a single presentation class corresponding to the "Wall" tab (see Figure 1). Each box inside the presentation class represents an attribute. For example, anchoredCollection represents a list of anchors for links, mapping an index. Presentation classes can also be nested (for example, "Basic information" is also tagged as a <<Presentation class>> and should be expanded in a different box).

Our defined Web refactorings over the navigation and presentation models give us a powerful abstraction mechanism compared with implementation-level refactorings.[7] This classification is intended to help developers choose the right refactoring depending on the design attributes they need to improve.

Navigation Model Refactorings

We define a navigation model refactoring as a change to the navigation model of a Web application that preserves

1. the set of operations made available by all the nodes (considered as a whole) in the model; and
2. the reachability of each operation through a navigation path from the home node.

Following this definition, navigation model refactorings include

- renaming nodes, node attributes, and node operations;
- adding nodes;
- removing unreachable or redundant nodes;
- moving contents or operations among the available nodes;

TABLE 1

Classification of Web refactorings.

Refactoring	Intent	Scope
Convert images to text[5] In webpages, replace any images that contain text with the text they contain, along with the markup and CSS rules that mimic the styling.	Accessibility	Code
Add link[7,11] Shorten the navigation path between two nodes.	Navigability	Navigation model
Turn on autocomplete[5] Save users from wasting time in retyping repetitive content. This is especially helpful to physically impaired users.	Effectiveness, accessibility	Code
Replace unsafe GET with POST[5] Avoid unsafe operations, such as confirming a subscription or placing an order without explicit user request and consent, by performing them only via POST.	Credibility	Code
Allow category changes[7] Add widgets that let users navigate to an item's related subcategories in a separate hierarchy of a hierarchical content organization.	Customization	Presentation model
Provide breadcrumbs[7] Help users keep track of their navigation path up to the current page.	Learnability	Presentation model

- adding links; and
- removing redundant links and links from unreachable nodes.

This list is not exhaustive. We can define more refactorings to improve usability as long as they preserve the Web application's behavior, given by its operations and links to reach those operations.

Presentation Model Refactorings

The presentation model specifies a Web application's behavior in the set of operations that users can trigger in a page and the set of links they can navigate from a page. Therefore, we define a presentation model refactoring as a change to the application's presentation model that preserves

1. the set of operations made available by all the model's pages, considered as a whole, and their semantics; and
2. the availability of an abstract interface for navigation model elements.

Under this definition, legal presentation model refactorings can

- split or merge pages;
- change an abstract widget's type, if the new type preserves the underlying functionality;
- reorganize the arrangement of widgets in a page; and
- add or change the available interface effects.

Note that some navigation model refactorings might signal the application of a presentation model refactoring. For example, moving content from a heavy node to a new one could lead us to split the page of the original node.

Target Usability Factors

In addition to classifying Web model refactorings by the design model to which they apply (scope), we also characterize them by the specific usability factor they aim to improve (intent). Inspired by research on patterns,[9] usability,[10] and the ISO 9241-11 usability definition (among others), we list factors contributing to a Web application's usability, which can be improved by refactorings. By classifying refactorings with their target usability factors, we aim to help developers find the correct refactorings for their problem. Web usability factors are

- *accessibility*: degree to which a Web application can be used by people with physical impairments or assistive technology.
- *navigability*: quality of the navigation structure in facilitating organized, effortless access to the application's contents through links.
- *effectiveness*: extent to which the application provides quick flows to expedite processes for advanced or returning customers.
- *credibility*: the application's capability to encourage trust and support lasting relationships with customers.
- *understandability*: extent to which content organization and layout help a user easily understand what the Web application provides, how to access it, and its current status.
- *customization*: ability to make relevant recommendations, to target user needs on the basis of past behavior or usage context, and to display enough information at any one time to alert interested users but not

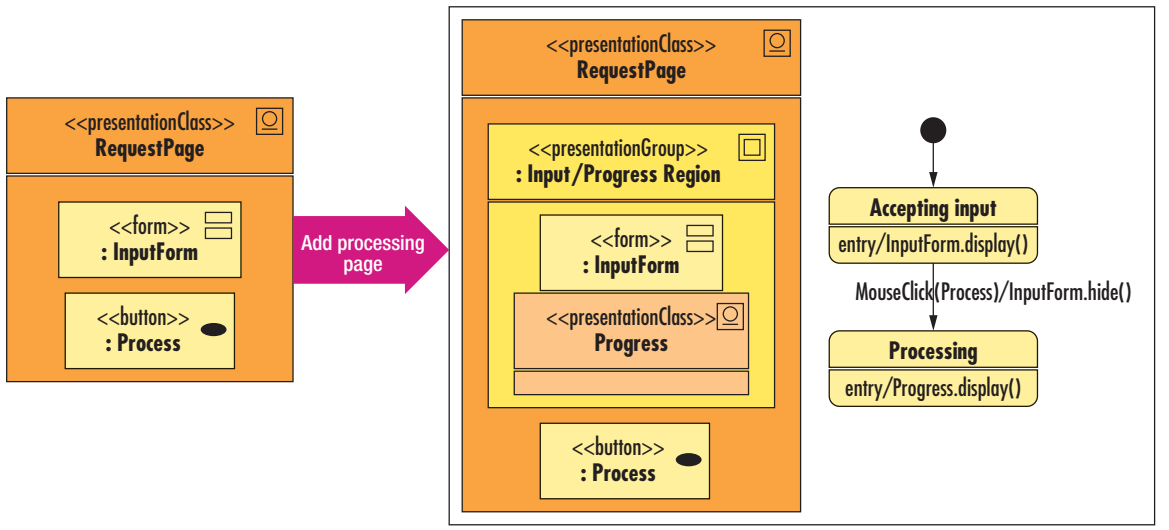


FIGURE 3. "Add processing page" refactoring. The InputForm is replaced by a presentationGroup, which is used to specify alternative components—in this case, InputForm and Progress. The state diagram describes the behavior of the presentationGroup, which shows the InputForm on entry and replaces it by the Progress section when the user clicks the Process button.

overwhelm or distract those who aren't.
- *learnability*: degree to which the application is easy to use or easy to learn through effective user support and guidance.

Table 1 shows some Web refactorings, both code[5] and models,[7,11] classified by their specific intent toward usability and their scope.

The Refactoring Process

We now review two important aspects of the refactoring process: when to refactor and how to measure refactoring benefits.

Detecting Bad Usability Smells

Incrementally detecting and correcting usability bad smells simplifies the process of overall usability evaluation, which application developers must nevertheless perform when they finish an application. Strategies for finding bad usability smells include

- user testing (performed by representatives of real users) or feedback,
- inspection methods (generally performed by experts), and
- Web usage analysis (mining user access logs).

We favor heuristic evaluation, which is the least formal of the inspection methods and fits well with an agile style. Heuristic evaluation analyzes the current system version according to a list of usability principles, reports usability problems (bad smells), and suggests improvements (usability refactorings). Manual processes for finding bad smells depend on the inspector's skill; automated tools can work even at the model level. For example, we can detect the bad smell "Absence of meaningful links" by analyzing a schema such as the one in Figure 2 and automatically applying the corresponding refactoring, "Turn attribute into link."

We categorize bad smells in two coarse groups, navigation and presentation, and tag each bad smell with the usability factors it affects. We stress, however, that the impact of bad smells strongly depends on the application domain (for example, e-commerce and e-learning), the types of users (for example, impaired users), and the site's idiosyncrasies. For example, the bad smell "Excessive information density," which might lead to refactorings such as "Introducing information on demand," is more critical for an e-commerce than a software download site.

In our growing catalogue of refactorings, we characterize each refactoring with the bad smells it deodorizes.

Measuring the Impact of Refactoring

The usability improvement that Web model refactorings can achieve will always depend on the developer's good judgment in selecting the most advantageous changes—that is, in his or her ability to detect the catalogued bad smells.

Developers can employ user feedback both to identify needs or opportunities for refactoring (by considering negative feedback as bad smells) and to evaluate user satisfaction after applying it. A formal approach to find bad smells, choose the appropriate refactorings, and measure the usability improvement gain is to apply Web model refactorings within a structured Web quality evaluation framework. We've

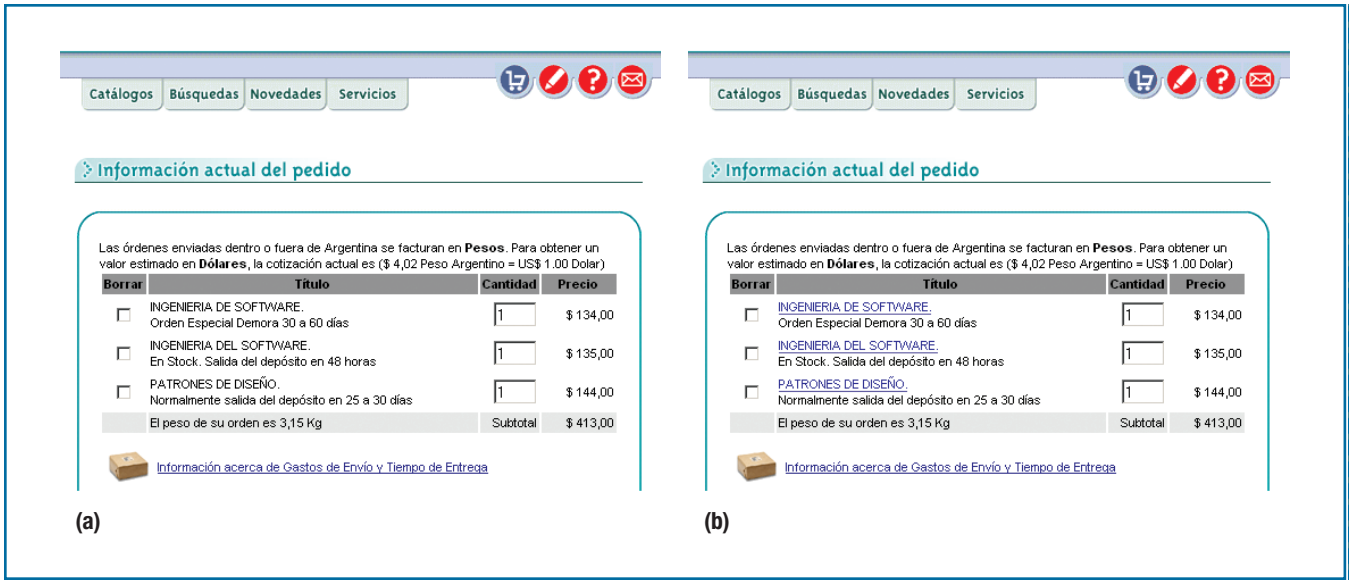


FIGURE 4. Shopping cart for an online bookstore: (a) without links and (b) with links after applying "Turn attribute into link" refactoring. The links let a user navigate from the book titles to the pages containing book details—for instance, to review the difference between the first and second titles in the cart.

already proposed such an approach to incrementally and systematically improve an application's usability.[12] This prior work also discusses the use of acceptance testing in the process of evaluating the final application's external quality, much as unit testing is used in traditional refactoring to evaluate the preservation of functionality. In this sense, as the application is transformed, we will need to adapt and apply the corresponding unit tests and assess the users' satisfaction.

Example Web Application Refactorings

We present three of our Web model refactorings here, with details in six sections. *Scope* is the model to which it applies, *intent* refers to the specific usability factors it pursues, *bad smells* label the targeted problem that might suggest its application, *motivation* is the story behind the bad smells, *mechanics* is the list of steps to apply the refactoring on stereotyped UML diagrams, and *example* is self-explanatory.

Add Processing Page

Scope: Presentation model
Intent: Understandability
Bad smells: No way of knowing current state in a process
Motivation: Users often leave a website in the middle of a transaction after waiting some time without receiving feedback that their transaction is in progress. A processing page can alleviate this problem.
Mechanics: In the presentation diagram, perform the following three basic steps:

1. Add a new presentation class to represent a processing page.
2. Add widgets into the new page for a progress bar and some text.
3. Change the interface effect associated with the widget that triggers the transaction so that it also navigates to the processing page.

Note that you can also apply this refactoring by replacing a section in the source page—probably an input form—with a progress bar, instead of navigating to a new page with the progress bar. This solution is common in rich Internet applications (RIAs). To model this kind of behavior, we attach a state diagram to the presentation diagram. The state diagram describes the dynamic behavior of interface elements in response to user-generated events. Figure 3 shows the mechanics of this version of the refactoring.
Example: The result of this refactoring is visible in practically all airline operator websites or travel brokers. It could be useful to apply it during the checkout process on websites like Amazon, when a progress bar indicates the transaction stages while the site contacts a credit card service: communicating with bank, checking card, getting authorization, and so on.

Turn Attribute into Link

Scope: Navigation model
Intent: Navigability
Bad smells: Difficult access to information, absence of meaningful navigation links
Motivation: Often, some page content clearly refers to other content (pages), such as product names and book authors. This page content should provide a navigation link, as suggested by the Embedded Links Web pattern.[9]
Mechanics: In the navigation diagram, select the attribute of the source node that better distinguishes the target

FIGURE 5. "Introduce information on demand" refactoring. (a) The original user interface presenting a set of CDs in an online music store, and (b) the interface after applying the refactoring using RIA technologies.

node and perform the next two steps:

1. Add a new link as an association from the source to the target node.
2. Remove the attribute from the source node.

Example: An opportunity to use this refactoring occurs when a customer checks the shopping cart's status during the purchase of products from an e-commerce website. You can use this refactoring to add links from product names in the cart to the product detail pages. In Cuspide.com, an online bookstore, we can apply this refactoring to add links from book titles in the shopping cart to the book page. Figure 4 shows how the shopping cart changes after applying this refactoring to the navigation model and synchronizing the webpage (either manually or automatically) with the new model.

Introduce Information on Demand

Scope: Presentation model
Intent: Navigability, customization
Bad smells: Excessive information density, cluttered interface, lack of interface space
Motivation: We often have plenty of information to show and a small area to accommodate it. One solution is to add a scrollbar to the available area. A better solution is to use the same screen space to show different content according to what a user chooses for an active object.
Mechanics: In the presentation diagram, select the page that will be affected by this refactoring and

1. Add or select the objects that will activate the presentation of the different content (for example, a menu).
2. Enclose the widgets that display the different content into a "presentation group" to specify that they're alternative components.
3. Attach a state diagram to specify the appearance of each alternative component in response to a mouse-generated event.

Example: Figure 5 shows this kind of refactoring in the context of an online music store, such as Amazon.com. By using RIA technologies to synchronize the interface with the model changes, this refactoring replaces a typical arrangement of CDs with an overlapping arrangement that can accommodate more CDs in the same space. When the mouse hovers over one of the CDs, the application shows its details.

These examples show how we can use refactoring to apply small changes to progressively improve the external quality of an existing Web application. For simplicity, we've presented examples with well-known coarse-grained usability factors. However, the intent can be finer grained—for example, you could define accessibility factors for visually impaired or motion-impaired people. Our catalog of Web model refactorings isn't complete, but we believe our characterization will be useful in suggesting many others. We hope other people will contribute to building a robust set of refactorings over time, plus tools that reify the model transformations.

Our approach is agnostic with respect to design methods and implementation technologies used for application development. All refactorings can be explained by showing how they affect the corresponding webpage. Additionally, the transformations can be applied at different abstraction levels, such as the modeling level in a model-driven approach or the implementation level in a Java or HTML code base. However, we do maintain that thinking about Web applications and refactorings from a model perspective is a more powerful refactoring approach than focusing on the implementation. We're now working to automate the refactoring process and incorporate refactoring in different model-driven Web development approaches.

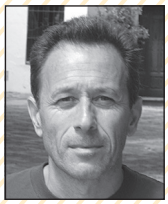
References

1. W. Opdyke and R. Johnson, "Creating Abstract Superclasses by Refactoring," *Proc. 1993 ACM Conf. Computer Science* (CSC 93), ACM Press, 1993, pp. 66–73.
2. M. Fowler, *Refactoring: Improving the Design of Existing Code*, Addison-Wesley, 1999.
3. M. Boger, T. Sturm, and P. Fragemann, "Refactoring Browser for UML," *Objects, Components, Architectures, Services, and Applications for a Networked World*, LNCS 2591, Springer, 2003, pp. 366–377.

4. S.W. Ambler and P.J. Sadalage, *Refactoring Databases: Evolutionary Database Design*, Addison-Wesley, 2006.
5. E.R. Harold, *Refactoring HTML: Improving the Design of Existing Web Applications*, Addison-Wesley, 2008.
6. G. Rossi et al., eds., *Web Engineering. Modelling and Implementing Web Applications*, Springer, 2008.
7. A. Garrido, G. Rossi, and D. Distante, "Systematic Improvement of Web Application Design," *J. Web Eng.*, vol. 8, no. 4, 2009, pp. 371–404.
8. N. Koch et al., "UML-Based Web Engineering: An Approach Based on Standards," *Web Engineering: Modelling and Implementing Web Applications*, G. Rossi et al., eds., Springer, 2008, pp. 157–191.
9. D.K. van Duyne, J.A. Landay, and J.I. Hong, *The Design of Sites: Patterns for Creating Winning Web Sites*, 2nd ed., Prentice Hall PTR, 2006.
10. J. Nielsen, *Designing Web Usability*, New Riders, 1999.
11. J. Cabot and C. Gomez, "A Catalogue of Refactorings for Navigation Models," *Proc. 8th Int'l Conf. Web Eng.* (ICWE 08), IEEE CS Press, 2008, pp. 75–85.
12. L. Olsina et al., "Web Application Evaluation and Refactoring: A Quality-Oriented Improvement Approach," *J. Web Eng.*, vol. 7, no. 4, 2008, pp. 258–280.

ABOUT THE AUTHORS

ALEJANDRA GARRIDO is an assistant professor at the Universidad Nacional de La Plata, Argentina, and a researcher at Conicet, Argentina's National Scientific and Technical Research Council. Her research interests include refactoring and Web engineering. Garrido has a PhD in computer science from the University of Illinois at Urbana-Champaign. Contact her at garrido@lifia.info.unlp.edu.ar.



GUSTAVO ROSSI is a full professor at Universidad Nacional de La Plata, Argentina, and a researcher at Conicet, Argentina's National Scientific and Technical Research Council. His research interests include Web engineering. Rossi has a PhD from Pontifícia Universidade Católica do Rio de Janeiro (PUC-Rio). Contact him at gustavo@lifia.info.unlp.edu.ar.

DAMIANO DISTANTE is an assistant professor at Unitelma Sapienza University, Italy. His research interests include Web engineering, conceptual modeling, and software evolution. Distante has a PhD in information engineering from the University of Salento, Italy. Contact him at damiano.distante@unitelma.it.