REFACTORING FOR USABILITY IN WEB APPLICATIONS

[1]Alejandra Garrido, [2]Gustavo Rossi and [3]Damiano Distante

[1,2]*LIFIA, Fac. de Informática, Univ. Nacional de La Plata and CONICET, Argentina*

{garrido, gustavo}@lifia.info.unlp.edu.ar

[3]*Università Telematica Tel.M.A., Italy*

damiano.distante@unitelma.it

Abstract

Refactoring was originally conceived as a technique for enhancing the design of an existing code base by applying small behavior-preserving transformations to the code. Later research extended the scope of refactoring to other software artifacts, such as design models, and widened its intent to improve additional software quality factors, such as performance. In this paper we discuss how to improve the usability of a Web application by applying refactoring on its design structure. We also classify each refactoring by the specific usability factor it improves and the bad usability smells it targets. Some examples of Web Model Refactorings illustrate our claims.

Keywords: D.2.2 Design Tools and Techniques. D.2.7 Distribution, Maintenance, and Enhancement. D.2.5.t Usability testing.

1. Introduction

Refactoring was originally introduced by Opdyke and Johnson in the early 90's, mainly for restructuring a class hierarchy of an object-oriented design [1]. A few years later, refactoring became popular with Fowler´s book, who defined 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, like Unified Modeling Language (UML) models [3], databases [4] and HTML documents [5]. While in all cases the basic philosophy of refactoring has prevail (i.e., each refactoring is a small behavior-preserving transformation), the intent of refactoring has considerably varied from the original proposal, which was improving readability, extensibility and maintainability of source code. The new intents, however, have not yet been clearly stated nor recognized, which might originate confusion. For example, an HTML refactoring like "Turn on Autocomplete" [5] does not improve any of the internal qualities just listed, but makes a Web form easier to use, thus shifting the intent of refactoring towards improving usability of the software product. A similar case occurs when refactoring an Application Programming Interface (API), which essentially involves external attributes since APIs are intended for others to use.

We believe that it is very important to link each refactoring not only to the bad smells it can eliminate, but also to the specific quality attributes it targets to improve. This paper illustrates our claim with some refactorings for Web applications that are meant to improve usability.

A Web application is generally designed by means of three models corresponding to the following three design layers: content, navigation, and presentation [6]. We have elsewhere presented a catalog of refactorings over the navigation and presentation models of the OOHDM Web design methodology [7]. The refactorings in that catalog are intended to improve the usability of the final application derived from these models. In this article we stress the importance of characterizing refactorings by the design model to which they apply and by their specific intent. Our classification in navigation and presentation refactorings allows us to target the specific attributes of each model and the bad smells that may come up in them. We also generalize their mechanics to stereotyped UML diagrams.

2. Refactoring for Usability in Web Software

We found that, even when it is not clearly stated, there are some refactorings in the literature that are not intended to improve *internal* quality attributes: some database refactorings improve performance [4], and some HTML refactorings improve accessibility [5]. This shows a trend in software development to apply refactoring to improve *external* quality attributes. Following this trend, we have defined refactorings for Web application models that improve the usability of a Web application while keeping its content and functionality unchanged [7].

Let us first see an example. Figure 1 shows a comparison of two Facebook interfaces, illustrating a major redesign they did back in 2008. Many widgets have been moved around, but let us focus on the appearance of tabs to ease navigation.

Figure 1. Comparison of Facebook interfaces

In the old interface (Figure 1 left), the homepage of a given account showed personal information, list of friends, and a mini-feed of news about the account owner and her friends, among other things. This resulted in a homepage cluttered with information and links of different kinds, which required repeated page scrolling. In the newer interface (Figure 1 right), the homepage has been split-up in different pages and a new tab row allows navigation between them: one tab for the mini-feeds page, another for personal information, another for manipulating photos, etc. Focusing on this particular change, we can see that the basic behavior of the application is preserved: it still allows 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 have catalogued this Web refactoring under the name *"Split Page"* [7].

We could discuss the mechanics of this refactoring at different levels. At the implementation or code level, it would amount to list 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 in which we want to split the original homepage and distribute into them the HTML code associated with the information, links, and widgets available in the original homepage.
2) add to each new page the HTML code to implement the tab row and the associated functionalities. These include: enabling navigation between the new pages; indicating the current page by distinguishing the active tab from the others; announcing the content of each page by using a meaningful label or icon for each tab.

However, if the site uses a different technology, such as Flash or a Web application framework to generate the HTML pages, the mechanics will differ, in the same way as traditional code refactorings depend on the target language. This is one of the reasons that make it more appealing to discuss the mechanics at a model level, where they can be generalized as model refactorings on UML-like diagrams. Another reason is that Web design models provide a more abstract picture of the application and its different aspects, so that we can take informed decisions and synchronize the changes in the different design layers. For example, splitting a page may motivate splitting a table in the database that holds the content of that page. A third reason is that we believe that model driven development (MDD) is becoming a reality as more tools appear that can generate code from Web models (e.g. UWE4JSF - http://uwe.pst.ifi.lmu.de/toolUWE4JSF.html). In this case the "new" HTML code would be derived automatically from the transformed models.

3. Characterizing Web Model Refactorings

Most Web engineering methodologies organize the design of a Web application into three layers by means of three design models: content model, navigation model, and presentation model [6]. The *content model* defines the types, attributes, and relationships of the application contents and their associated behavior. Refactorings that apply over this model are traditional refactorings intended to improve internal quality factors. The *navigation model* maps classes of the content model to *navigation nodes* (units of information and behavior perceived by the user), and associations between content classes to *navigation links*. Moreover, associations with a multiplicity greater than one are mapped to *navigation indexes*, to organize the navigation space. The *presentation model* defines the Web application 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 is abstract because it does not specify the exact position of widgets, nor their graphical attributes, but just their type. Both the navigation and presentation models can be represented with stereotyped UML class diagrams [8]. Figure 2 shows simplified versions of a navigation diagram (left) and a presentation diagram (right) for the application presented in Section 2. Classes in the navigation diagram represent nodes or indexes, and classes in the presentation diagram represent pages. For space reasons we have omitted attributes of navigation classes and show a single presentation class corresponding to the "Wall" tab (see Figure 1). Attributes of a presentation class appear in stereotyped boxes where, for example, an "anchored collection" represents a list of anchors for links, mapping an index. Presentation classes can also be nested (e.g. "Basic information").

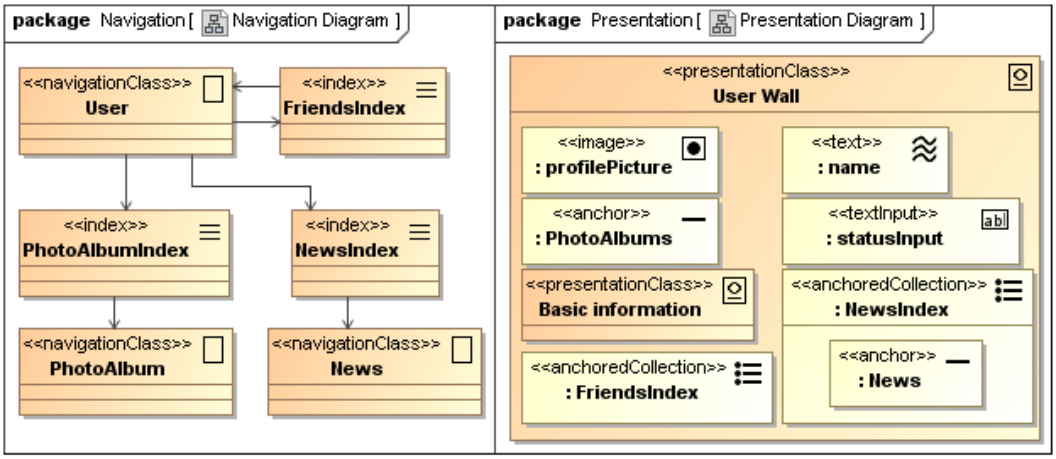


Figure 2. A navigation diagram and presentation diagram

We have defined Web refactorings over the navigation and presentation models, which gives us a powerful abstraction mechanism as compared with the implementation level; we called them *navigation model refactorings* and *presentation model refactorings* respectively [7]. This classification is intended to help developers in choosing the right refactoring depending on the design attributes they need to improve, as further described in Sections 3.1 and 3.2.

3.1 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;
2. the reachability of each operation, by way of a navigation path from the home node.

Following the above 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;
- adding links;
- removing redundant links and links from unreachable nodes.

3.2 Presentation Model Refactorings

The behavior of a Web application, as specified by its presentation model, is given by the set of operations that users may 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 presentation model of a Web application that preserves:

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

Under the above definition, legal presentation model refactorings may:

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

3.3 Target Usability Factors

Additionally 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 the works on Web patterns [9], Web usability [10], and the ISO 9241-11 definition of usability among others, we provide a non-exhaustive list of different factors that contribute to the usability of a Web application and that can be improved through concrete 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 that facilitates the organized and effortless access to the application´s contents.
- **Effectiveness**: an effective Web application provides quick flows to expedite processes for advanced or returning customers.

- **Credibility**: building trust and credibility is essential to establish a lasting relationship with customers.
- **Understandability**: organization and layout of contents that enables users to easily understand what the application provides and how to access it. Also, ability to show current status.
- **Customization**: ability to make relevant recommendations or to target the user needs based on past behavior or context of usage. Also, amount of information shown at once, that should be just enough to tip interested users but not overwhelm or distract the others.
- **Learnability**: ease of use and effective support to operate on a Web application.

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

Refactoring	Intent	Scope
Convert Images to Text [5] In Web pages, 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] Autocompletion helps users avoid wasting time retyping repetitive data. It's especially helpful to physically impaired users.	Effectiveness Accessibility	Code
Replace Unsafe GET with POST [5] Unsafe operations, such as confirming a subscription or placing an order, should be performed only via POST to avoid accidentally taking such actions without explicit user request and consent.	Credibility	Code
Allow Changing Category [7] Add widgets to allow navigating to related subcategories of items in a separate hierarchy of a hierarchical organization of contents.	Customization	Presentation model
Provide Breadcrumbs [7] Help users keep track of their navigation path up to the current page.	Learnability	Presentation model

Table 1. Classification of Web refactorings

4 The Refactoring Process

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

4.1 Detecting Bad Usability Smells

The concept of "bad usability smell" has already emerged as an extension of Beck and Fowler's "bad smells" in code. Incrementally detecting and correcting usability smells simplifies the process of overall usability evaluation, which nevertheless has to be performed when the application is finished. There are different strategies to find bad usability smells: user testing (done by representatives of real users) or users' feedback, inspection methods (generally done by experts), and Web usage analysis (mining logs of users' access). We favor heuristic evaluation, which, being the most informal of inspection approaches, fits well with an agile style. Heuristic evaluation analyzes the current version of the system by using a list of usability principles, reports usability problems (bad smells), and suggests improvements (usability refactorings). Bad smells can be found manually, thus making the process dependent of the inspector's skill, or with automatic tools, which can even work at the model level. For example, the bad

smell "Absence of meaningful links" can be detected by analyzing a schema like the one in Figure 2, and the corresponding refactoring (*"Turn Attribute into Link"*, see Section 5) can be applied automatically.

We categorize bad smells in two coarse groups, navigation and presentation, and tag each bad smell with the usability factors it affects. It must be stressed, however, that the impact of bad smells strongly depends on the application's domain (e.g., e-commerce, e-learning, etc.), the types of users (e.g., impaired users), and also on the site's idiosyncrasies. For example, the bad smell *"Excessive information density"* which may lead to refactorings such as "*Introducing information on Demand*" (see Section 5), is more critical for an e-commerce site than for a software download site. In our growing catalogue of refactorings, we have characterized each refactoring with the bad smells it deodorizes as shown in the examples of Section 5.

4.2 Measuring the Impact of Refactoring

The improvement in usability that can be achieved with Web model refactorings will always depend on the good judgment of the developer to select the most advantageous changes, i.e., in his/her ability to detect the catalogued "bad smells".

Users' feedback can be used both to identify needs or opportunities for refactoring (by considering negative feedback as bad smells) and to evaluate the user satisfaction after applying it. A formal approach to find bad smells, choose the appropriate refactorings and measure the improvement gain in usability is that of applying Web model refactorings within a structured Web quality evaluation framework. We have proposed such an approach to incrementally and systematically improve the usability of a Web application in [12]. That publication also discusses the use of acceptance testing in the process of evaluating the external quality of the final application, 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 asses the users' satisfaction.

5. Examples

This section presents in detail three of our Web model refactorings. Each one has 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 may suggests 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 finally there is an *Example*.

i) Add Processing Page
Scope: Presentation model
Intent: Understandability
Bad smells: Incapacity to know current state in a process.
Motivation: Often users quit 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:

(i) Add a new presentation class to represent a processing page.
(ii) Add widgets into the new page for a progress bar and some text.

(iii) Change the interface effect associated with the widget that triggers the transaction so that it also navigates to the processing page.

Note that this refactoring may also be applied 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, nowadays, in Rich Internet Applications (RIA). To model this kind of behavior, we attach to the presentation diagram a state diagram to describe the dynamic behavior of interface elements in response to user-generated events. Figure 3 shows the mechanics of this version of the refactoring. Note that 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 Process button is clicked.

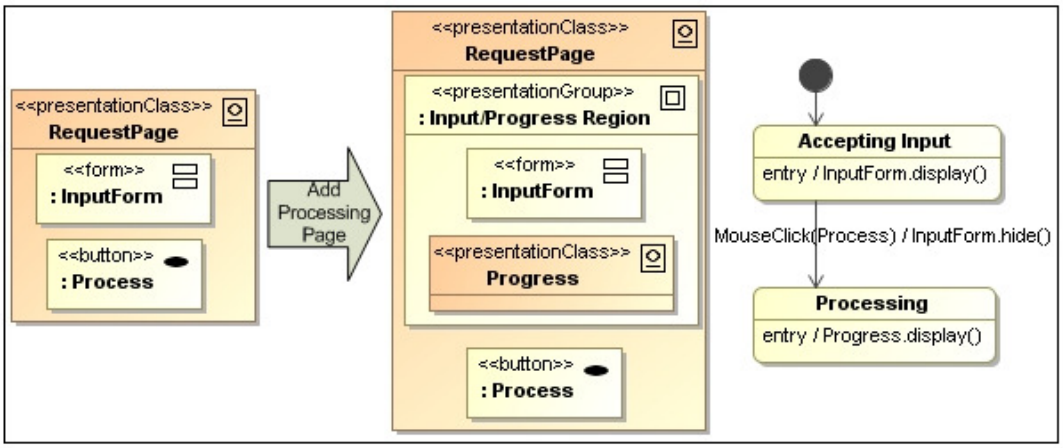


Figure 3. "Add Processing Page" refactoring

Example: This solution is well visible, for example, in practically all airline operator websites or travelling brokers. Also, during the check-out process in websites like Amazon, contacting the credit card service takes some time and a progress bar may provide feedback of the transaction stages like "communicating with bank", "checking card", "getting authorization", etc.

ii) Turn Attribute into Link
Scope: Navigation model
Intent: Navigability
Bad smells: Difficult access to information, Absence of meaningful navigation links.
Motivation: Often, there are page contents which clearly refer to other contents (pages) such as product names, book authors, etc. These contents should provide a navigation link, as suggested by the Web pattern "Embedded Links" [9].
Mechanics: In the navigation diagram, select the attribute of the source node that better distinguishes the target node and perform the next two steps:

(i) Add a new link as an association from the source to the target node.
(ii) Remove the attribute from the source node.

Example: An opportunity for using this refactoring occurs when checking the status of the shopping cart during the process of buying some products in an e-commerce website. This refactoring may be used to add links from product names in the shopping cart to the product detail pages. In Cuspide.com (an online bookstore), we may 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 Web page (either manually or automatically) with the new model.

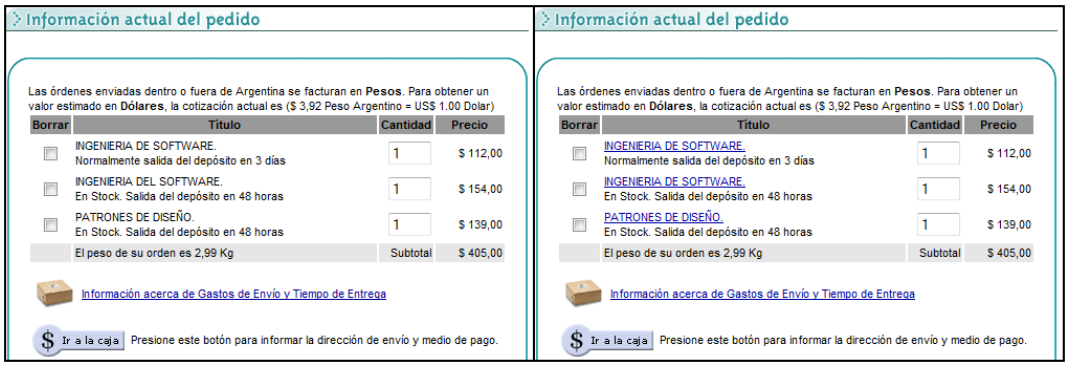


Figure 4. Shopping cart for an online bookstore without links (left) and with links (right) after applying "Turn Attribute into Link"

iii) Introduce Information on Demand

Scope: Presentation model

Intent: Navigability, Customization

Bad smells: Excessive information density, Cluttered interface, Lack of interface space.

Motivation: Many times we 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 a user choice, performed on some active object.

Mechanics: In the presentation diagram, select the page which will be affected by this refactoring and:

(i) Add or select the objects that will activate the presentation of the different content (e.g., a menu).

(ii) Enclose the widgets that display the different content into a "presentation group" to specify that they are alternative components.

(iii) Attach a state diagram to specify the appearance of each alternative component in response to a mouse generated event (e.g., mouse-on or mouse-click).

Example: Figure 5 shows an example of how this kind of refactoring is reflected in the context of an online music store like Amazon. By using RIA technologies to synchronize the interface with the changes in the model, a typical arrangement of CDs is replaced with an overlapping arrangement that can accommodate more CDs in the same space. When the mouse hovers over one of the CDs its details are shown.

Figure 5: "Introduce Information on Demand" refactoring

6. Concluding Remarks

In this paper we showed how to use the concept of refactoring to improve the external quality of Web applications, specifically their usability.

Using refactoring, we may apply small changes to an existing Web navigation or presentation model, and achieve a more usable and accessible application, without changing its behavior. The refactorings included in our catalogue are characterized by scope (the target design model), intent (the usability factors to improve), and bad smells (the usability problems they face). For simplicity, we have exemplified the paper with well-known coarse-grained usability factors; however, the intent can become finer grained if we consider, for example, accessibility factors for visual-impaired or motion-impaired people. Note that our catalogue of Web model refactorings is not complete, but we believe that our characterization will be useful in suggesting many others; we hope that others will be willing to contribute to build 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, since all refactorings can be explained by showing how they affect the corresponding Web page. Additionally, the transformations can be applied at different levels of abstraction, such as the modeling level in a model-driven approach, or the implementation level in a Java or HTML code base. However, thinking about Web applications and Web refactorings from a model perspective grants more power than focusing on the implementation, for the reasons described in Section 2. We are now working to provide tool support to automate the refactoring process and to incorporate refactoring in different model-driven Web development approaches.

References

[1] W. Opdyke and R. Johnson, "Creating abstract superclasses by refactoring". *Proc. 1993 ACM Conference on Computer Science* (CSC '93), ACM, 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*. *Int. Conf. NetObjectDays* (NODe 2002), 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, O. Pastor, D. Schwabe, L. Olsina, eds., *Web Engineering. Modelling and Implementing Web Applications*. Springer, HCI, 2008.

[7] A. Garrido, G. Rossi and D. Distante. "Systematic Improvement of Web Application Design". *Journal of Web Engineering*, 8 (4), 2009, pp. 371-404.

[8] N. Koch, A. Knapp, G. Zhang and H. Baumeister, "UML-based Web Engineering: An Approach based on Standards", *Web Engineering: Modelling and Implementing Web Applications*, Rossi, Pastor, Schwabe and Olsina, eds., Springer, HCI, 2008, pp. 157-191.

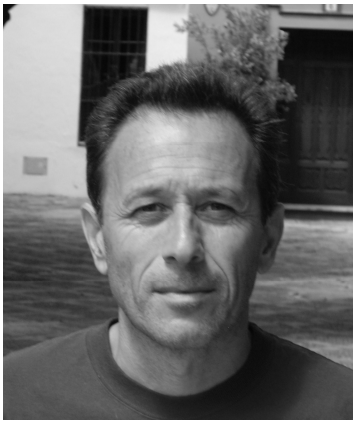
[9] D.K. van Duyne, J.A. Landay, and J.I. Hong, *The Design of Sites: Patterns for Creating Winning Web Sites* 2nd ed., Prentice Hall PTR, 2006.

[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.Conf. on Web Engineering* (ICWE 08), IEEE CS Press, 2008, pp.75-85.

[12] L. Olsina, A. Garrido, G. Rossi, D. Distante and G. Canfora, "Web Application Evaluation And Refactoring: A Quality-Oriented Improvement Approach", *Journal on Web Engineering*, 7 (4), 2008, pp. 258-280.

Alejandra Garrido is an Assistant Professor at University of La Plata, Argentina, and a researcher at CONICET. She has a PhD in Computer Science from the University of Illinois at Urbana-Champaign. Her research interests include refactoring and Web engineering.



Gustavo Rossi is a Full Professor at University of La Plata, Argentina, and a researcher at CONICET. He holds a PhD from PUC-Rio, Brazil. His research interests include Web engineering.

Damiano Distante is an Assistant Professor at the Università Telematica Tel.M.A., Italy. He holds a PhD in Information Engineering from the University of Salento, Italy. His research interests include Web engineering, conceptual modeling, and software evolution. He is a member of the Steering Committee for the IEEE International Symposium on Web Systems Evolution (WSE).