

Personalized Web Accessibility using Client-Side Refactoring

According to W3C accessibility standards, most Web applications are neither accessible nor usable for people with disabilities. Developers often solve this problem by building separate accessible applications, but these are seldom usable and typically offer less functionality than the original. Another common solution is to maintain a single application, but create an accessible view by applying on-the-fly transformations to each requested page — a solution that rarely suits all audiences. A third solution is described here: let users improve Web accessibility in their client browsers through interface refactorings, which offer many customized, accessible views of a single application.

**Alejandra Garrido,
Sergio Firmenich,
Gustavo Rossi,
and Julián Grigera**
*Universidad Nacional
de La Plata, Argentina*

Nuria Medina-Medina
Universidad de Granada, Spain

Ivana Harari
*Universidad Nacional
de La Plata, Argentina*

Refactoring was originally conceived as a technique to improve software's internal qualities — such as understandability and maintainability — while preserving semantics.¹ In prior work, we adapted the refactoring approach to improve a Web application's external attributes, such as usability.² These Web refactorings consist of small navigation or interface transformations that enhance perceivable aspects of Web applications, such as user interaction and content presentation, while preserving functionality. Refactorings can also solve accessibility and usability problems for disabled users.³ Still, it's usually impractical to address interface improvements for all audiences because disabilities can vary dramatically in nature (visual, cognitive, or motor), severity (blindness, color blindness, or strabismus) and extent (total or partial). In such

different contexts, “one for all” is barely feasible.

When applying refactoring to improve internal qualities, developers decide which transformations to apply and where, because they're the ones benefiting from the improvement. As Brian Foote and Joseph Yoder put it, “Who better to resolve the forces impinging upon each design issue as it arises, as the person who is going to have to live with these decisions?”⁴ Moreover, different developers might prefer alternative solutions for the same “bad smell” (that is, the design problem that motivated the refactoring¹). Following on this general philosophy, we believe that end users should be able to tailor a website's interface for their own benefit.

We propose empowering users (or close representatives) with the ability to select, in their client browsers, their own interface refactorings for each site



Figure 1. Applying refactoring to Gmail. The Gmail interface (a) before and (b) after applying Distribute Global Menu to address accessibility issues with the checkbox and operations on top. Gmail logo reprinted with permission.

they access. We call our approach *Client-Side Web Refactoring*. CSWR allows for the automatic creation of different, personalized views of the same application to solve the particular bad smells that each user recognizes. (Developers should continue to focus on addressing general usability problems on the server-side, however, and reach the minimum level of accessibility, or “A”).

Here, we describe CSWR and a case study we ran with visually impaired users (though a similar solution could be applied for other disabilities).

Refactoring Example

Figure 1a shows the inbox in Gmail, Google’s email reader. Gmail includes checkboxes on the left that let users select several emails, which is handy for applying an action to all of them. However, for visually impaired people using a screen reader, it’s uncomfortable to have to go back to the checkbox at the line’s beginning to select emails after reading the line, and then go back to the top to apply an operation after selecting the emails; they report this as a “bad accessibility smell” (an accessibility problem that motivates a refactoring).

A refactoring that solves this bad smell is Distribute Global Menu, which distributes a menu of actions affecting a list of elements to each element individually. This eases the local application of an operation because it requires only a single click immediately after the element is read. Figure 1b shows the result of applying this refactoring to the Gmail inbox. The set of actions was removed from the top

and attached to each email in the form of icons (each with an alternative text).

However, some users who report the same bad smell are more comfortable using contextual menus, so the set of actions isn’t read with every email. For them, the Contextualize Global Menu refactoring is more appropriate. Also, experienced users prefer to keep the global menu so they can operate on several emails at once; for them, using the Postpone Selection refactoring will move the checkboxes to the last column.

Client-Side Web Refactoring

As this example shows, a Web refactoring changes a Web application’s navigation structure or look and feel, preserving its content and operations while removing bad usability or accessibility smells. In previous work, we used Web refactorings to enhance navigation and presentation during the development life cycle.^{2,5} We can generate a complete new version of an application with a specific aim – such as a mobile version – by systematically applying and composing refactorings. Here, we propose a similar approach to improve accessibility, where refactoring is applied after deployment and during actual use of the Web application, altering the interface in the browser itself.

Our CSWR approach has two key benefits:

- *Simpler maintenance* – developers maintain a single core application, applying Web usability refactorings that address a general audience, while different refactored versions can be created by and for different users.

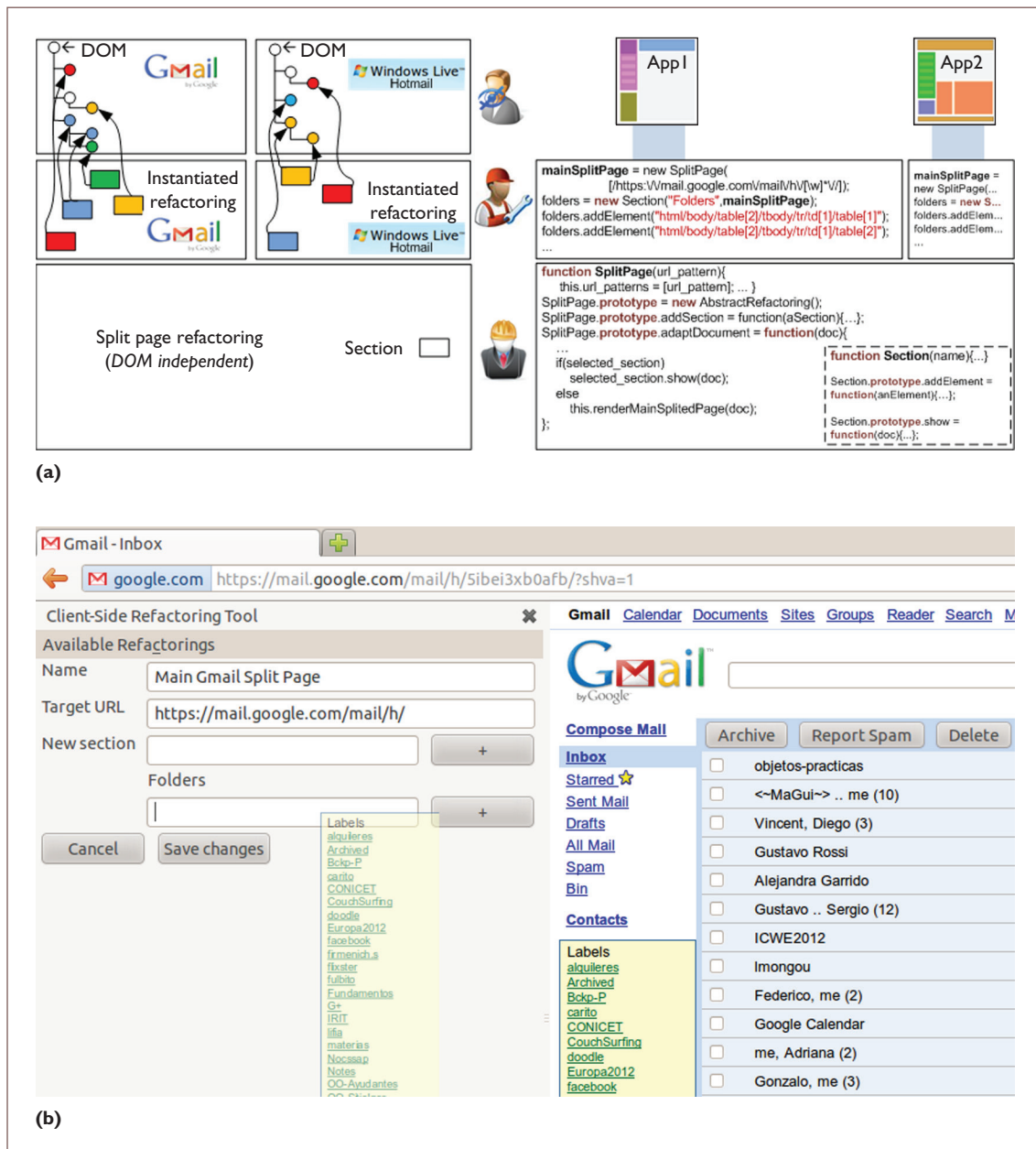


Figure 2. Client-Side Web Refactoring. a) Split Page refactoring levels: the generic implementation at the bottom, two instances (for Gmail and Hotmail) in the middle, and the result on each DOM at the top. b) Split Page instantiation: the intermediary drags Gmail's Labels to create a new Folders section.

- *Architecture independence* – developing a CSWR requires little (if any) knowledge of the target application's underlying architecture.

The engine behind CSWRs uses a client-side adaptation framework that aims to adapt existing applications by changing their DOM structure.⁶ CSWRs are implemented by specializing the class `AbstractRefactoring` (provided in our framework) and redefining the method `adaptDocument()` with the refactoring's mechanics.

For example, consider the Split Page refactoring, which solves the problem of a saturated, complex webpage by dividing it into a set of simpler pages or sections.² In this case, the method `adaptDocument()` receives the DOM elements that represent disjoint page sections as parameters and creates a new page for each section, replacing the original page's contents with an index to the new pages (see the bottom of Figure 2a).

To apply the Split Page refactoring to a specific page, you must first create an instance

of `SplitPage` (such as the two instances in the middle level of Figure 2a), passing as parameters:

- a URL identifying the target page or a URI pattern for a set of pages of the same site (such as all Gmail pages in Figure 2a's code), and
- instances of the class `Section`, which contain DOM elements specified through XPath expressions. Because DOM elements might not always be identified through XPath queries based on attributes, absolute XPath expressions from the DOM root might be required.

The three levels in Figure 2a correspond to the three steps involved in the refactoring process, which can involve three user roles (pictured in the middle column):

- The *JavaScript (JS) programmer* creates new refactorings by writing a parameterized script that reuses components offered by our framework's refactoring engine.
- An *intermediary* instantiates refactorings for a specific website. (The JS programmer can play this role as well). The refactorings are instantiated either by writing code (as in Figure 2a) or using our refactoring tool (as in Figure 2b). This graphical tool lets the intermediary point-and-click on the target page to select the components that act as values for each refactoring parameter. Figure 2b, for example, shows an instantiation of `Split Page`, where the group of email labels is dragged to a `Folders` section that will go into a new page.
- *End users* install instantiated refactorings by choosing them from an accessible menu in their Web browsers; from then on, they can access the refactored website configured to their needs. Unlike traditional refactoring, we added this new step (separating it from instantiation) so that handicapped users unable to code a script or use a graphical refactoring tool can still be part of the process by choosing their own refactorings.

CSWR's power comes not only from letting end users choose specific refactorings, but also from letting intermediaries compose refactorings in different ways. CSWR composition is done at instantiation time, and requires special handling because refactorings can interfere

with each other. Similar to code refactorings, an applied refactoring might invalidate the next refactoring's preconditions. In this case, the preconditions of CSWRs are the existence of the DOM elements specified as parameters (XPath expressions). Thus, an applied refactoring might invalidate subsequent refactorings if it changes the xPaths that identify their target elements. For example, if `Split Page` and `Distribute Global Menu` are both instantiated on the original DOM's elements, and `Split Page` is applied first, the `Distributed Global Menu` won't find its target elements in their original XPath location.

Our refactoring tool helps intermediary users correctly compose CSWRs so that they can create and distribute a complete, accessible version of an application as a composition of CSWR instances. When an intermediary user selects several refactorings to compose, the tool creates and suggests a possible sequence, first by placing structural refactorings (such as `Split Page`), then refactorings that adapt specific DOM elements (such as `Distribute Global Menu`), and finally by placing DOM-independent refactorings (such as `Replace Image with their Alt Text`). CSWRs are thus instantiated in order, and users can specify certain noninterfering CSWRs to be independent. With this information, the tool creates the menu of optional CSWRs, so that when end users install a composed set of CSWRs, they can still choose to activate or deactivate independent CSWRs individually.

Case Study: Accessible Gmail

We carried out a case study on the HTML version of Gmail with visually impaired users to validate our claim that refactorings offer a better experience when they're customized for and by end users to suit their own expertise, screen-reader of choice, personal preferences, and so on. We played the roles of JS programmers and intermediaries.

Preliminary Study

We first conducted a study with seven potential Gmail users to test our preliminary hypothesis: our refactored Gmail version is more accessible and usable than the original. Of the seven users, six were blind and one had a severe sight deficiency. The test users had various computer and Internet skill levels; most used email clients and Web browsers, although only one had previously used webmail tools.

We gave users the following tasks:

1. Read and reply to an email.
2. Compose an email and send it to several people.
3. Search and delete a specific sent message.

Users had to complete the tasks first in the original Gmail, so we could check for ignored bad smells, and then in a refactored version, to detect unsolved bad smells and determine how this first refactoring attempt improved or spoiled the user experience.

The selected refactorings were Split Page (to partition the email list, tags list, and the main Google menu), Distribute Global Menu (for email operations), Reorganize into a List (for sets of unnumbered actions), Remove Redundant Operation (to remove the global menu of email operations at the top and leave the one at the bottom of the email list), and Postpone Selection (to move the checkbox column to the end of the each email's row).

During the study, we used observation and questionnaires to gather various feedback. These led to several findings:

- Different skill levels in screen reader use led to different complications; for example, refactorings aimed at simplifying the structure were useful mostly for novice users, but were burdensome for experienced ones.
- Users suggested new refactorings, including the use of context menus as an alternative to distributed menus.
- Users new to webmail clients couldn't complete the tasks in the original Gmail, but could complete them with external aid in the refactored version (which proves our hypothesis with new users).
- Users had positive feedback on the content's organization and functionality. Using a ratings scale of *very good*, *good*, *average*, and *poor*, the feedback included five *good* qualifications and one *average* for organization/functionality, but ease of use scored only two *goods* and four *averages*, which helped us gather other bad smells and improve the tools for intermediaries and final users.

Actual Experiment

Following our preliminary study, we conducted an experiment with new users. Of the 10 blind

users in this study, three were experienced in operating Gmail and the other seven were experienced in Web browsing, but not with Gmail. This time, our hypothesis was that a personalized version of Gmail is better than our completely refactored version. We conducted the test with one user at a time, going through simplified tasks that covered the same basic ground as in our previous study:

1. Delete all emails from a specific sender.
2. Find a specific deleted email in the Trash and put it back in the Inbox.
3. Answer the email recovered in task 2.

Before the actual experiment, we asked users to answer a specific email (as in task 3) on the original Gmail. This had a twofold purpose: it gave us a sense of their expertise with the tools (browser and screen reader) and reduced the bias that might occur for users without previous Gmail experience.

For the main part of the experiment, we devised an optimal set of four refactorings based on the experience from the previous study and applied them to Gmail:

- Split Page, to reduce each page's contents and thus ease content access.
- Distribute Menu, to simplify the tasks applied to each item on a list (such as emails).
- Contextualize Menu, to present actions over an item as a contextual menu.
- Postpone Selection, to let users read the email subjects and check them immediately afterward to apply an action to several selected emails.

Before we asked the subjects to complete the tasks, we explained each refactoring. Once the users were finished with the tasks in the completely refactored version, we had them arrange their own set of refactorings using the menu options in their browsers. They then performed the affected tasks – those related to the selected refactorings – again for further comparison.

Results

The main measurement we gathered was task completion time, comparing the times from the completely refactored site with those repeated in the personalized version (see Table 1). Out of 10 users, five preferred Contextualize Menu to

Table 1. Results comparing completely refactored vs. personalized versions of Gmail.

User	Selected refactorings	Completely refactored (secs)	Personalized (secs)	Drop rate (%)
1	SplitPage, ContextualizeMenu, Postpone Selection	180	160	11.11%
2	SplitPage, ContextualizeMenu, Postpone Selection	300	200	33.33%
3	SplitPage, ContextualizeMenu, Postpone Selection	143	43	69.93%
4	SplitPage, ContextualizeMenu, Postpone Selection	91	52	42.86%
5	SplitPage, ContextualizeMenu, Postpone Selection	68	66	2.94%
			Partial	32.03%
6	DistributeMenu, Postpone Selection	90	65	27.78%
7	DistributeMenu, Postpone Selection	180	97	46.11%
			Partial	36.94%
			Overall	33.44%

Distribute Menu, two discarded Split Page, and the rest preferred the refactored site as it was.

The overall completion times decreased by an average of 33.44 percent, with 32.03 percent for the subjects who chose Contextualize Menu and 36.94 percent for the group with no Split Page refactoring.

From this second study, we gathered new feedback, which led to the following findings:

- Novice users found it easier to navigate using Split Page, but this wasn't the case for experienced users because the split structure requires additional navigation steps for some tasks (such as when folders were moved to a folder index in a separate page).
- Some novice users suggested splitting the pages in a new way, so that some of the features were always present; others wanted to easily hide the main menu when desired.
- Users' habits directly interfere with the results. For example, experienced users didn't appreciate the benefits of Distribute Menu until after they tried it and used it for a while, because they were used to dealing with the global menu.

These results clearly show the importance of personalization: because experienced users can move quickly through a page with keyboard combinations, they prefer loaded pages and shorter navigation paths – a solution that frustrates inexperienced users because it demands going through lots of content every time the page reloads.

Discussion

In previous work,⁶ we developed tools that let users create conceptual models and then define adaptations in terms of these models, based on the idea of ModdingInterface.⁷ We're now planning to adapt this conceptual layer specifically for use with CSWR. This new abstraction level would let developers define concepts (and their properties) over DOM elements and define the adaptations in terms of concepts, instead of manipulating DOM elements directly with XPath expressions. Thus, if two Web applications manage the same concepts – and thereby form an application family that shares the same abstract model – the CSWRs defined in terms of abstract concepts can be applied to both applications.

For example, webmail applications that share the same abstract model (Inbox, Folder, Email, and so on) can use the same set of CSWRs defined in terms of these concepts. This approach not only allows more CSWR reusability, but it might improve script resilience. Such resilience is one of the most important drawbacks for client-side scripting, and our approach isn't exempt: when webpage DOMs change, scripts might stop working. If the development uses an agile process, server-side refactorings might update the DOM often. Another common constraint in this type of technology is that it's not applicable to all websites; for example, sites developed using technologies such as Flash might present problems.

Although we propose CSWR to improve accessibility for unsighted users, developers can easily apply the same approach to create different

Related Work in Improving Web Accessibility

Ideally, accessibility should be contemplated early, during Web application design, and webpages should follow existing standards or guidelines such as Web Content Accessibility Guidelines (WCAG). Such guidelines could, for example,^{1,2} be incorporated in the Web engineering life cycle. Other key approaches to ensure or enforce accessibility include systematically assessing compliance with guidelines³ and automatically detecting accessibility problems in webpages.^{4,5} Despite these research efforts, however, most Web applications aren't yet fully accessible, and the problem must be tackled with more dynamic approaches.

A well-known technique to transform existing webpages to be accessible is *transcoding*,⁶ which applies transformations on the fly, based on semantic annotations manually added by developers or automatically derived from Web design models. Transcodings can be applied on the server, the client, or a proxy.⁶ Our approach shares the philosophy behind transcoding, but most of the existing transcoding systems for accessibility lack extensibility and personalization:

- All transcoding methods (including Text Magnification and Content Reorder)⁶ are predefined by their developers, and it isn't possible to add new transcoding methods. Most such systems are only extensible – in terms of which webpages will be transcoded – if volunteers are allowed to annotate a website and apply the transcodings for future visitors. Our Client-Side Web Refactoring (CSWR) approach allows for a new type of volunteer (JavaScript programmer) who can add new refactorings in response to new bad smells or tackle the same bad smell in a different way.
- Transcoding-aware annotations have the same impact for all users regardless of their special capacities. Because transcodings are considered transparent from the users' viewpoint, it isn't possible to fine tune them for a specific webpage according to each user. With CSWR, each user can select a different set of refactorings for each website.
- Transcodings don't necessarily preserve behavior; they can remove some operations, such as when they aim to simplify content. In contrast, refactorings were conceived as behavior-preserving transformations,⁷ which, in the case of Web applications, means preserving content and functionality.⁸
- Transcodings don't necessarily compose, and they might even interfere with each other.⁶ We propose CSWR composition as an additional way of customizing a website, letting users apply a sequence of refactorings incrementally.

Interest is growing in client-side scripting⁹ for customizing existing pages, as proven by large communities using GreaseMonkey (www.greasespot.net), a popular tool for client-side scripting that allows any kind of change over a webpage's DOM.

Specific tools such as WebAdapt2Me (http://www-03.ibm.com/able/accessibility_services/WebAdapt2Me.html) and AccessMonkey¹⁰ focus on accessibility. However, both tools only let users make basic changes to style, such as font size or color, and basic changes to content order.


When it comes to accessibility improvements, current client-side tools are too primitive. Generic tools such as GreaseMonkey hardly provide mechanisms for script compatibility; when different scripts are applied over the same pages, the execution of one script can spoil the previous one's changes or invalidate the execution of the script that follows. Besides, while GreaseMonkey lets users adapt a specific page, it doesn't let them generalize – that is, they can't apply the same change on different pages if changes depend on the DOM's structure. Although GreaseMonkey is excellent as a weaver, it doesn't offer facilities for accessibility. In contrast, our tool is a weaver that further provides mechanisms for refactoring definition, composition, and installation. Tools designed specifically for accessibility based on client-side scripting, such as AccessMonkey, also have several limitations, mainly because they're focused on basic style changes, which are usually insufficient to solve problems such as user disorientation or long navigation chains.

References

1. V. Luque Centeno et al., "Web Composition with WCAG in Mind," *Proc. Int'l Cross-Disciplinary Workshop on Web Accessibility (W4A)*, ACM, 2005, pp. 38–45.
2. P. Plessers et al., "Accessibility: A Web Engineering Approach," *Proc. 14th Int'l Conf. World Wide Web*, ACM, 2005, pp. 353–362.
3. J. Vanderdonck, A. Beirekdar, and M. Noirhomme-Fraiture, "Automated Evaluation of Web Usability and Accessibility by Guideline Review," *Proc. 4th Int'l Conf. Web Engineering*, LNCS 3140, Springer, 2004, pp. 17–30.
4. C. Benavidez et al., "Semi-Automatic Evaluation of Web Accessibility with HERA 2.0," *Proc. Int'l Conf. Computers Helping People with Special Needs*, LNCS 4061, Springer, 2006, pp. 199–206.
5. *TAW3: Tool for the Analysis of Websites*, Fundación CTIC, Spanish Ministry of Employment and Social Affairs (IMERSO) Online Web Accessibility Test; www.tawdis.net.
6. C. Asakawa and H. Takagi, "Transcoding," *Web Accessibility: A Foundation for Research*, S. Harper and Y. Yesilada, eds., Springer, 2008, pp. 231–261.
7. M. Fowler, *Refactoring: Improving the Design of Existing Code*, Addison Wesley, 1999.
8. A. Garrido, G. Rossi, and D. Distanto, "Refactoring for Usability in Web Applications," *IEEE Software*, vol. 3, no. 28, 2011, pp. 60–67.
9. O. Diaz, C. Arellano, and J. Iturriz, "Layman Tuning of Websites: Facing Change Resilience," *Proc. 17th Int'l Conf. World Wide Web (WWW)*, 2008, ACM, pp. 127–128.
10. J. Bigham and R. Ladner, "Accessmonkey: A Collaborative Scripting Framework for Web Users and Developers," *Proc. Int'l Cross-Disciplinary Conf. Web Accessibility (W4A)*, ACM, 2007, pp. 25–34.

views of a Web application targeted to improve other external qualities or to create, for example, a mobile version. Note that W3C guidelines for both accessibility (Web Content Accessibility Guidelines; www.w3.org/TR/WCAG10) and mobile (Mobile Web Best Practices; www.w3.org/TR/mobile-bp) have several similarities, which can be implemented as CSWR if they aren't contemplated originally by Web applications.

Refactoring is a powerful and essential tool that lets developers improve running applications based on feedback. This feedback might come from bad smells in the code identified by developers, or from bad smells in usability and accessibility experienced by users. However, for developers to correct bad smells based on user feedback typically takes a long time – especially for bad accessibility smells, which generally aren't a priority. We thus put refactoring in the hands of users, who know better what they actually need. This not only lets users customize specific interaction improvements, but also removes such improvements from the main development cycle of the applications themselves, which reduces cost and effort.

Web refactorings are a technically compelling way to dynamically improve users' experience, as they are composable and let users create different application versions without any knowledge of the internal design. This is a huge benefit because it also allows a crowdsourcing approach to making CSWRs and their compositions available. Indeed, our future work includes building a crowdsourcing tool for volunteers to upload new generic refactorings or instantiate existing refactorings for a particular website, which could come as a package of composed refactorings to create a completely new version of a site. We propose hosting crowdsourced CSWRs to spread their adoption with the least possible burden for end users. Moreover, to overcome the existence of different versions of refactorings in response to webpages' DOM evolution, our crowdsourcing tool's architecture would automatically select the latest versions of a given CSWR or CSWR set. 

References

1. M. Fowler, *Refactoring: Improving the Design of Existing Code*, Addison Wesley, 1999.

2. A. Garrido, G. Rossi, and D. Distanto, "Refactoring for Usability in Web Applications," *IEEE Software*, vol. 3, no. 28, 2011, pp. 60–67.
3. N. Medina-Medina et al., "Refactoring for Accessibility in Web Applications," *Proc. 11th Int'l Conf. Interacción Persona-Ordenador*, Assoc. Interacción Persona-Ordenador, 2012, pp. 427–430; www.aipo.es/items.php?id=364.
4. B. Foote and J. Yoder, "Big Balls of Mud," *Pattern Languages of Programs*, Addison Wesley, 2000, pp. //FORTHCOMING//
5. N. Medina-Medina et al., "An Incremental Approach for Building Accessible and Usable Web Applications," *Proc. 11th Int'l Conf. Web Information System Eng. (WISE)*, Springer, 2010, pp. 564–577.
6. S. Firmenich et al., "A Crowdsourced Approach for Concern-Sensitive Integration of Information across the Web," *J. Web Engineering*, vol. 10, no. 4, 2011, pp. 289–315.
7. O. Diaz, C. Arellano, and J. Iturrioz, "Layman Tuning of Websites: Facing Change Resilience," *Proc. 17th Int'l Conf. World Wide Web (WWW)*, ACM, 2008, pp. 127–128.

Alejandra Garrido is an assistant professor at Facultad de Informática, Universidad Nacional de La Plata, Argentina, where she's a member of the Research and Development in Advanced IT Lab (LIFIA). She is also a researcher at Argentina's National Scientific and Technical Research Council (CONICET). Her research interests include refactoring and Web engineering, focusing on design patterns, frameworks, refactoring for the C language, and refactoring for usability. Garrido has a PhD in computer science from the University of Illinois at Urbana-Champaign. She's a member of the Hillside Group. Contact her at garrido@lifia.info.unlp.edu.ar.

Sergio Firmenich is a teaching assistant at Facultad de Informática, Universidad Nacional de La Plata, Argentina, and a member of the Research and Development in Advanced IT Lab (LIFIA). His research interests focus on Web application adaptability – specifically, on engineering the adaptation of existing applications. Firmenich has a PhD in computer science from Universidad Nacional de La Plata. Contact him at firmenich@lifia.info.unlp.edu.ar.

Gustavo Rossi is a professor at Facultad de Informática, Universidad Nacional de La Plata, Argentina, and the director of the Research and Development in Advanced IT Lab (LIFIA). He is also a researcher at CONICET. His research interests include Web application design and agile approaches. Rossi has a PhD in informatics from

the Pontifical Catholic University of Rio de Janeiro, Brazil. He's one of the developers of the Object-Oriented Hypermedia Design Method (OOHDM) and is a member of IEEE and ACM. Contact him at gustavo@lifa.info.unlp.edu.ar.

Julián Grigera is a PhD student at Facultad de Informática, Universidad Nacional de La Plata, Argentina. His research interests are in Web development and agile methodologies, and he's previously worked on context-aware systems architecture and sensing mechanisms, and usability and accessibility for Web applications and mobile devices. Grigera has a Licentiate degree in informatics from the University of La Plata. Contact him at juliang@lifa.info.unlp.edu.ar.

Nuria Medina-Medina is an associate professor and researcher in the Department of Computer Languages and Systems at the University of Granada, where she's a member of the Group on Specification, Development and Evolution of Software (GEDES). Her research interests

include hypermedia systems, user modeling, user adaptation, and software evolution, as well as Web browsing, refactoring for the visually impaired, and bioinformatics. Medina-Medina has a PhD in computer science from the University of Granada. Contact her at nmedina@ugr.es.

Ivana Harari is an assistant professor and Director of Web Accessibility at the Facultad de Informática, Universidad Nacional de La Plata, Argentina. Her research interests include human-computer interaction, mobile user interface design, and Web accessibility, as well as usability engineering and testing, user-centered design, free and open source software (FOSS) tools for disabled people, and adaptive and accessible mobile interfaces. Harari has an education specialist degree in university teaching from the University of La Plata. Contact her at iharari@ada.info.unlp.edu.ar.



Selected CS articles and columns are also available for free at <http://ComputingNow.computer.org>.