PERSONALIZED WEB ACCESSIBILITY USING CLIENT-SIDE REFACTORING

Alejandra Garrido, 1Sergio Firmenich, 1Gustavo Rossi, 1Julián Grigera, 2Nuria Medina Medina, and 3Ivana Harari

1,3 Fac. de Informática, Univ. Nacional de La Plata, Argentina
1also CONICET, Argentina
2Escuela Técnica Sup. de Ingenierías Informática y de Telecomunicación, Univ. Granada, Spain

Abstract

According to the W3C (World Wide Web Consortium) accessibility standards, most Web applications are neither accessible nor usable for people with disabilities. Developers often solve this problem by building a separate, accessible application, which is seldom usable and tends to offer less functionality than the original one. To maintain a single application, developers may create an accessible view by applying on-the-fly transformations to each requested page; yet this solution rarely fits all audiences. In this paper we propose to let users improve Web accessibility in their client browsers through interface refactorings, allowing for many customized, accessible views of one single application.

Keywords: D.2.7.e Evolving Internet applications, D.2.14.a User Interfaces. D.2.7.m Restructuring, reverse engineering, and reengineering. D.2.5.t Usability Testing.

1. Introduction

Refactoring was originally conceived as a technique to improve internal qualities of software, like understandability and maintainability, while preserving semantics [1]. We adapted the refactoring approach to improve external attributes of a Web application like usability [2]. These Web refactorings consist of small navigation or interface transformations that enhance perceivable aspects of Web applications, like user interaction and presentation of content, while preserving the application’s functionality. They may also solve accessibility/usability problems for disabled users [3]. Nevertheless, it is usually impractical to address interface improvements for all audiences, since disabilities can be of very different nature (visual, cognitive, motor, etc.), severity (blindness, color blindness, strabismus, etc.) and extent (total or partial). In such different contexts, “one for all” is barely feasible.

When refactoring is applied to improve internal qualities, developers decide which transformations to apply and where, since it’s them who benefit from the improvement, and as Foote and Yoder say “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 may prefer alternative ways of solving the same “bad smell” (i.e., a design problem that motivates refactoring [1]). Following the same philosophy, we claim that final users should tailor the website’s interface for their own
benefit. We therefore propose to empower them (or close representatives) with the ability to select in their client browser the interface refactorings of their choice on each site they access. For example, Figure 1.a) shows the inbox in Gmail (Google’s email reader). Checkboxes on the left allow selecting several emails, which is handy for applying an action to all of them. However, when using a screen reader, having to go through the list of items before applying an operation, and having to go back to the checkbox at the beginning of the line after reading the line, is uncomfortable for most visually impaired users; 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. It eases the local application of an operation since it only requires a single click just after reading the element. Figure 1.b 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).

In contrast, other users that report the same bad smell are more comfortable using contextual menus, so the set of actions is not read with every email. The refactoring “Contextualize Global Menu” is more appropriate for them. Furthermore, experienced users prefer to keep the global menu to operate on several emails at once. They can use the refactoring “Postpone Selection”, which moves checkboxes to the last column.

We call this approach Client-Side Web Refactoring (CSWR). It allows to automatically create different, personalized views of the same application, solving the bad smells that each user recognizes. Note that developers should still focus on addressing general usability problems on the server-side, and reach the minimum level of accessibility (‘A’).

This article describes CSWR for accessibility; thus, the refactorings we propose are compliant with W3C guidelines. Our objective here was to try them with final users, especially with visually impaired, although a similar solution could be applied for other disabilities.
2. Client-Side Web Refactoring

A Web refactoring changes the navigation structure or presentation look and feel of a Web application, preserving its content and operations while removing bad usability or accessibility smells. In previous works we have used Web refactorings to enhance navigation and presentation during the development life cycle [2, 5]. We can generate a complete new version of the same application with some specific aim (e.g. 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 the use of the Web application, altering the interface in the browser itself. This CSWR approach brings 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.

- Architecture independence: developing a CSWR requires little knowledge (if any) of the underlying architecture of the application to transform.

The engine behind CSWRs uses a framework for client-side adaptation, which aims to adapt existing applications by changing their DOM structure [6]. CSWRs are implemented by specializing 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 contents of original page by an index to the new pages (see the bottom level of Fig. 2.a).

To apply the refactoring Split Page to a specific page, it is first necessary to create an instance of SplitPage (like the two instances that appear in middle level of Fig. 2.a), passing as parameters:

(i) a URL identifying the target page; alternatively, it may receive a URI pattern for a set of pages of the same site (e.g. all Gmail pages in the code of Fig. 2.a), and

(ii) instances of class Section which contain DOM elements specified through xPath expressions; since DOM elements may not always be identified through xPath queries based on attributes, absolute xPath expressions from the DOM root may be required.

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

- Javascript (JS) Programmer: creates new refactorings by writing a parameterized script, reusing several components offered by the refactoring engine in our framework.
- **Intermediary**: instantiates refactorings for a specific website (though it may be the same JS Programmer). This is done either by writing code (like in the middle level of Fig. 2.a) or using our refactoring tool (like in Fig. 2.b). This graphical tool allows point-and-click on the target page to select the components that act as values for each refactoring parameter (e.g., Fig. 2.b shows an instantiation of *Split Page*, where the group of e-mail labels is dragged to the Folders’ section that will go into a new page).

- **Final user**: installs instantiated refactorings by choosing them from an accessible menu in their Web browser, and from then on accesses the refactored website, configured to her needs. Compared with the traditional refactoring process, this is a new step that we incorporated separately from the instantiation, so that a handicapped user unable to code a script or use a graphical refactoring tool may still be part of the process by choosing her own refactorings.

---

**Figure 2.** Abstract representation (a) and concrete instantiation (b) of a CSWR
Note that the power of our CSWRs comes not only from the possibility of choosing a specific refactoring, but also from the ability to compose them in different ways. CSWR composition is done at instantiation time, and requires special handling, as refactorings may interfere with each other. Similarly to code refactorings, an applied refactoring may invalidate the preconditions of the next one. In this case, the preconditions of CSWRs are the existence of the DOM elements specified as parameters (XPath expressions). Thus, an applied refactoring may 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 elements of the original DOM, and Split Page is applied first, the second one will not find its target elements in their original XPath location.

We help intermediary users in the correct composition of CSWRs with our refactoring tool, 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, by placing first structural refactorings (e.g., Split Page, section 3), second refactorings that adapt specific DOM elements (e.g., Distribute Global Menu), and last DOM-independent refactorings (e.g., Replace images for their alt text, section 3). Then, CSWRs are instantiated in order, and the user may specify certain non-interference CSWRs to be independent. With this information, the tool creates the menu of optional CSWRs, so that when final users install a composed set of CSWRs, they may still choose to activate/deactivate those independent CSWRs individually.

3. Case Study: Accessible Gmail

We carried out a case study on the HTML version of Gmail with visually impaired users in order to validate that refactorings offer a better experience when they are customized for/by final users, depending on different aspects like expertise, screen-reader of choice, or personal preferences. We played the roles of JS Programmers and Intermediaries.

3.1 Preliminary study

We first conducted a study with 7 potential users of Gmail, to test our preliminary hypothesis: our refactored version of Gmail is more accessible and usable than the original. Out of the 7 users, 6 of them were blind and 1 had a severe sight deficiency, and they had different skill levels using computers and Internet (most used email clients and Web browsers, but just one of them had previously used webmail tools).

We gave users the following tasks:
1. Read and reply 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 how this
first refactoring attempt improved/spoiled the user experience. The selected refactorings were: *Split Page* (to partition email list, tags list and general 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 gathered different kinds of feedback, by observation and questionnaires:

- Different levels of skill in the use of the screen reader gave way to different complications, e.g. refactorings aimed at simplifying the structure were mostly useful for novice users, but burdensome for the experienced ones.
- New refactorings were suggested like the use of context menus as an alternative to distributed menus.
- Users new to webmail clients could not complete the tasks in the original Gmail, but could complete them with external aid in the refactored version (which proves our hypothesis with fresh users).
- Users reported positive feedback on the contents organization and functionality (5 good qualifications and 1 regular), but not on ease of use (4 regulars and 2 good), which helped us gather other bad smells and improve the tools for intermediaries and final users.

### 3.2 Actual experiment

After the previous study, we conducted an experiment on 10 blind users, different from the subjects of the previous study: 3 of them where experienced operating Gmail and the other 7 were experienced in web browsing, but not on Gmail webmail. 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 ground than the 3 we used in the previous study, namely:

1. Deleting all emails from a specific sender.
2. Finding a specific deleted email in the Trash and putting it back in the Inbox.
3. Answering the email recovered in task 2.

Before the actual experiment, we requested users to complete task 3 (with a specific email) on the original Gmail. This had a twofold purpose: getting a first appreciation of their expertise with the tools (browser and screen reader), and reducing the bias that harms the first run of tasks for being the first encounter (some) users have with Gmail.

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 the contents of each page to ease the contents access.
- *Distribute Menu* to simplify the tasks applied to each item on a list (e.g. emails).
- *Contextualize Menu* where actions over an item are presented as a contextual menu.
- *Postpone Selection* to allow the users to read the emails subjects before checking them for grouped actions.

Before we asked the subjects to complete the tasks, we explained them each refactoring. Once they were finished with the tasks, we had them arrange their own set of refactorings by using the menu options in their browser. Then, they performed the affected tasks (i.e. those reached by the selected refactorings) again for further comparison.

### 3.3 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, 5 preferred *Contextualize Menu* to *Distribute Menu*, 2 of them discarded *Split Page*, and the rest preferred the refactored site as it was.

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

<table>
<thead>
<tr>
<th>User</th>
<th>Selected refactorings</th>
<th>Completely refactored (secs)</th>
<th>Personalized (secs)</th>
<th>Drop rate (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>SplitPage, ContextualizeMenu, Postpone Selection</td>
<td>180</td>
<td>160</td>
<td>11,11%</td>
</tr>
<tr>
<td>2</td>
<td>SplitPage, ContextualizeMenu, Postpone Selection</td>
<td>300</td>
<td>200</td>
<td>33,33%</td>
</tr>
<tr>
<td>3</td>
<td>SplitPage, ContextualizeMenu, Postpone Selection</td>
<td>143</td>
<td>43</td>
<td>69,93%</td>
</tr>
<tr>
<td>4</td>
<td>SplitPage, ContextualizeMenu, Postpone Selection</td>
<td>91</td>
<td>52</td>
<td>42,80%</td>
</tr>
<tr>
<td>5</td>
<td>SplitPage, ContextualizeMenu, Postpone Selection</td>
<td>68</td>
<td>66</td>
<td>2,94%</td>
</tr>
<tr>
<td></td>
<td>Partial</td>
<td></td>
<td></td>
<td>32,03%</td>
</tr>
<tr>
<td>6</td>
<td>DistributeMenu, Postpone Selection</td>
<td>90</td>
<td>65</td>
<td>27,78%</td>
</tr>
<tr>
<td>7</td>
<td>DistributeMenu, Postpone Selection</td>
<td>180</td>
<td>97</td>
<td>46,11%</td>
</tr>
<tr>
<td></td>
<td>Partial</td>
<td></td>
<td></td>
<td>36,94%</td>
</tr>
<tr>
<td></td>
<td><strong>Overall</strong></td>
<td></td>
<td></td>
<td><strong>33,44%</strong></td>
</tr>
</tbody>
</table>

The new feedback we gathered again from observation and questionnaires was:

- Novice users found it easier to navigate using *Split Page*, while experienced ones didn’t, as the split structure requires additional navigation steps for some tasks (for example, folders had been moved to a folder index in a separate page).
- Some novice users suggested a different way of splitting the pages to have some of the features always present, while others asked for the possibility of directly hiding 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, since they were previously familiarized and used to deal with the global menu.
These results clearly show the importance of personalization: since experienced users can move quickly through a page with keyboard combinations, they prefer loaded pages and shorter navigation paths while the same solution is frustrating for non-experienced users, as it demands going through lots of content every time the page reloads.

4. Related Work and Discussion

Accessibility should be ideally contemplated early, during Web applications design. Web pages should follow existing standards or guidelines such as WCAG (Web Content Accessibility Guidelines). For example [7, 8] propose to incorporate these guidelines in the Web engineering life cycle. Systematically assessing compliance to guidelines [9] and automatically detecting accessibility problems in Web pages. [10, 11] are also key approaches to ensure/enforce accessibility. Despite these research efforts, most Web applications are yet not fully accessible and very often, the problem has to be tackled with more dynamic approaches.

A well known technique to transform existing Web pages to be accessible is transcoding [12]. Transcoding applies transformations on the fly, based on semantic annotations manually added by developers or automatically derived from Web design models. Transcodings may be applied on the server, the client or a proxy [12]. We share the philosophy behind transcoding, but we claim that most of the existing transcodings systems for accessibility have a lack of extensibility and personalization:

- All transcodings methods (Text Magnification, Content Reorder, etc) [12] are predefined by their developers, and it is not possible to add new transcoding methods. Most of these systems are only extensible (in terms of which Web pages will be transcodded) by allowing volunteers to annotate a websites, thus applying the transcodings for future visitors of the website. Our CSWR approach allows for a new type of volunteer (JS programmer) who may add new refactorings in response to new bad smells, or to tackle the same bad-smell in a different way.
- Transcoding-aware annotations have the same impact for all users regardless of their special capacities. As transcodings are thought to be transparent from the users’ point of view, it is not possible to fine tune them for a specific Web page according to each user. With CSWR, each user may select a different set of refactorings for each website.
- Transcodings do not necessarily preserve behavior, as they may remove some operations for example when they aim to simplify content; in contrast, refactorings were conceived as behavior-preserving transformations [1], which in the case of Web applications means preserving content and functionality [3];
- Transcodings do not necessarily compose and may even interfere with each other [12]. We propose composition of CSWR as an additional way of customizing a website, allowing for a sequence of refactorings to be applied incrementally.
There is a growing interest in client-side scripting [13] to customize existing pages, proven by large communities using GreaseMonkey², a popular tool for client-side scripting that allows any kind of change over a webpage’s DOM. There are also specific tools like WebAdapt2Me³ and AccessMonkey [14], focused on accessibility. Both only allow users to make basic changes of style like font size or color and basic content reorder.

However, current client-side tools are too primitive regarding accessibility improvements. On one hand, generic purpose tools like GreaseMonkey hardly provide mechanisms for scripts compatibility; when different scripts are applied over the same pages, the execution of one script can spoil the changes made by the previous one, or invalidate the execution of the following. Besides, while GreaseMonkey allows adapting a specific page, it doesn’t allow for generalization, i.e., applying the same change on different pages, when changes are dependent on the DOM’s structure. Although GreaseMonkey is excellent as a weaver, it does not offer facilities for accessibility. Our tool is not only about weaving but it provides mechanisms for refactoring definition, composition and installation. On the other hand, specific tools for accessibility based on client-side scripting like AccessMonkey have several limitations, mainly because they are focused on basic changes of style, usually insufficient to solve problems like users’ disorientation or long navigation chains.

In a previous work [6] we have developed tools for allowing users to create conceptual models and then define adaptations in terms of these models, based on the idea of ModdingInterface [13]. Now, we are planning to adapt this conceptual layer specifically to be used with CSWR. This new abstraction level would allow developers to 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. In this way, if two Web applications manage the same concepts, and thus 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, the Web Mail family of applications sharing the same abstract model (e.g. Inbox, Folder, Email, etc.), may use the same set of CSWRs defined in terms of these concepts. This approach not only allows more reusability of CSWRs but it may improve the resilience of scripts. Resilience is one of the most important drawbacks for client-side scripting, and our approach is not exempt of this: when Web pages DOMs change, scripts may stop working. If the development uses an agile process, server-side refactorings may update the DOM often. Another common constraint in this kind of technologies is that these are not applicable to all Web sites, for example, there are problems with those developed with technologies such as Flash.

Although we propose CSWR to improve accessibility for unsighted users, the same approach may be easily applied to create different 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 (WCAG: Web Content Accessibility Guidelines) and mobile (MWBP: Mobile Web Best Practices) have several similarities, which can be implemented as CSWR when these are not contemplated originally by Web applications.

² http://www.greasespot.net
³ http://www-03.ibm.com/able/accessibility_services/WebAdapt2Me.html
5. Conclusions and future work

Refactoring is a powerful and essential tool that allows improving a running application after learning from feedback. The feedback may 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 learned from users’ feedback, it usually takes a long time, especially for bad accessibility smells, which are generally not considered as a priority. Therefore, we place refactoring in the hands of users, who know better what they need. This way, specific interaction improvements are not only custom-made for each user, but they are also removed 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, given their ability to compose to each other and create different versions of a Web application without any knowledge of the application’s internal design. This is a huge benefit since it also allows a crowdsourcing approach to make CSWRs and their compositions available. Our future work includes building a crowdsourcing tool for volunteers to upload new generic refactorings, or instantiations of existing refactorings for a particular website, which may also come as a package of composed refactorings to create a complete new version of a site. We propose hosting crowdsourced CSWRs to spread their adoption with the least possible burden for final users. Moreover, to overcome the existence of different versions of refactorings in response to Web pages’ DOM evolution, the architecture of our crowdsourcing tool should automatically select the latest versions of a given CSWR or set of them.

References


Authors’ bios

Alejandra Garrido is an assistant professor at the College of Informatics, University of La Plata, Argentina, and a researcher at CONICET (Argentina's National Scientific and Technical Research Council). Her current research interests include refactoring and Web engineering. She has particularly worked on design patterns, frameworks, refactoring for the C language, and refactoring for usability. Garrido has MS and PhD degrees in Computer Science from the University of Illinois at Urbana-Champaign. She is a member of the Hillside Group. Contact her at garrido@lifia.info.unlp.edu.ar.

Sergio Firmenich is a PhD student at the College of Informatics, University of La Plata, Argentina. He has a PhD scholarship at CONICET, and he is a teaching assistant at the same college. The main Firmenich’s research interest is adaptability of Web applications, specifically, engineering the adaptation of existing Web applications. Contact him at firmenich@lifia.info.unlp.edu.ar.

Gustavo Rossi is a full professor at the College of Informatics, University of La Plata, Argentina and a researcher at CONICET. His current interests include Web applications design and Agile approaches. He is one of the developers of the OOHDM Web applications design approach. He holds a PhD from PUC-Rio, Brazil. He is a member of IEEE and ACM. Contact him at gustavo@lifia.info.unlp.edu.ar.

Julián Grigera is a PhD student at the College of Informatics, University of La Plata, Argentina. His main research interest is web development and agile methodologies. He has previously worked on context-aware systems architecture and sensing mechanisms, and also on usability and accessibility for web applications and mobile devices. Contact him at juliang@lifia.info.unlp.edu.ar.

Nuria Medina-Medina received the Ph.D. in Computer Science from the University of Granada in 2004. Since 2001, she works as researcher in the Department of Computer Languages and Systems of this Spanish University. Currently, she is associate professor in this department. She is member of the GEDES group—research group in specification, development and evolution of software. Her main research interests include hypermedia systems, user modeling, user adaptation and software evolution. In addition, since 2009 she also works on other topics such as Web browsing, refactoring for visually impaired and bioinformatics. Contact her at nmedina@ugr.es.

Ivana Harari is an assistant professor and the director of Web Accessibility at the College of Informatics, University of La Plata, Argentina. Her current research interests include Human computer interaction, Mobile user interface design and Web accessibility. She has worked over the last fifteen years on Usability engineering & testing, User-centered design, FOSS software tools for disabled people, and adaptive & accessible mobile interfaces. She has an Education Specialist degree in university teaching from the University of La Plata. Contact her at iharari@ada.info.unlp.edu.ar.