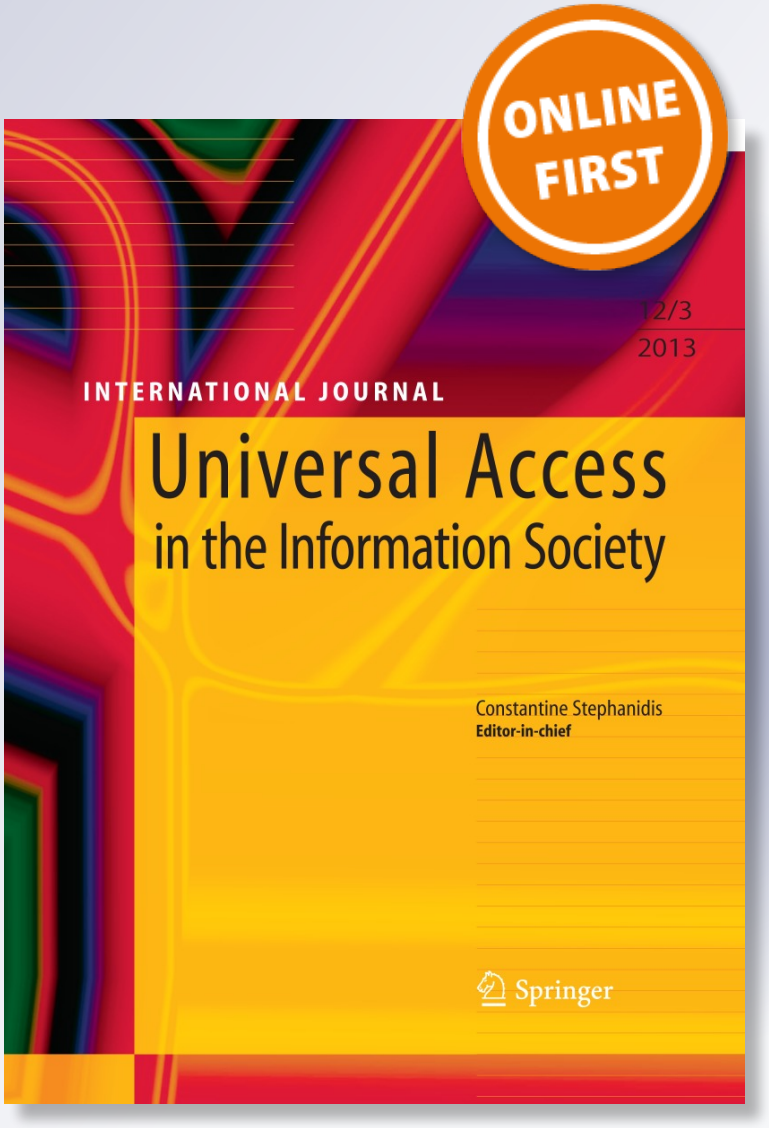
Improving accessibility of Web interfaces: refactoring to the rescue

Alejandra Garrido, Gustavo Rossi, Nuria Medina Medina, Julián Grigera & Sergio Firmenich

Universal Access in the Information Society
International Journal

ISSN 1615-5289

Univ Access Inf Soc
DOI 10.1007/s10209-013-0323-2



Springer

Univ Access Inf Soc
DOI 10.1007/s10209-013-0323-2

LONG PAPER

Improving accessibility of Web interfaces: refactoring to the rescue

Alejandra Garrido · Gustavo Rossi · Nuria Medina Medina · Julián Grigera · Sergio Firmenich

Abstract Universal access should be a target for all public Web sites. However, it is very hard to achieve, and even Web applications that comply with accessibility standards may still lack usability for disabled users. This paper proposes refactoring as an essencial technique to incrementally improve the accessibility and usability of a Web interface. Some accessibility refactorings are described and classified by the problems that each refactoring addresses. The way mainstream Web sites struggle with accessibility is illustrated, and two evaluations of email clients are presented as empirical evidence of the significance of accessibility refactorings at a low implementation cost.

Keywords Accessibility · Web applications · Refactoring · Reengineering

1 Introduction

When building accessible Web applications, the design target is to reach conformance with the W3C Web Content Accessibility Guidelines (WCAG) [1]. These guidelines can be met at three levels of conformance: A, AA, and AAA (highest). WCAG also specify testable success criteria for determining whether the content satisfies them [1]. Nevertheless, satisfying all WCAGs' success criteria does not guarantee usability [2], since people with disabilities may still find it difficult to navigate and interact with the content in a comfortable, easy, and effective way. For example, let us consider Amazon's *Visually Impaired Store* (see Fig. 1). Although the homepage does not conform to WCAG at level A because of a carrousel of unnamed images, even if it is changed to be compliant, it will still have usability problems that complicate its access using a screen reader, such as the presentation of mixed content in different areas of the same page, leading to a saturated page. Such problems are called "bad smells" in usability or accessibility, following the refactoring's jargon for design problems [3].

Even if graphical interface designers have been trying for years to resemble users' real world with interface metaphors, there is little evidence of the real usability or efficiency to support them. For example, small checkboxes (often used to apply an operation to a list of items) have never worked as easy as checking items with a pen on paper in the real world. In addition, checkboxes always appear before their label since (as indicated by WCAG [1]) this their most predictable position, which favors obeying the metaphor over the need of screen reader's users to hear the label before deciding the need to check it. In working applications, developers fear the risk of changing to a new, nonstandard interface metaphor, even when they care about usable accessibility, since the upgrade is too costly particularly if it has to be rolled back. This context is quite similar to the fear of developers to change their working code before the advent of refactoring and refactoring tools, which allow them to incrementally improve design quality

A. Garrido (✉) · G. Rossi · J. Grigera · S. Firmenich
LIFIA, Fac. de Informática, Univ. Nacional de La Plata, 50 y 120, CP 1900 La Plata, Pcia. de Buenos Aires, Argentina
e-mail: garrido@lifia.info.unlp.edu.ar

A. Garrido · G. Rossi
CONICET, Buenos Aires, Argentina

N. M. Medina
Escuela Técnica Sup. de Ingenierías Informática y de Telecomunicación, Univ. Granada, C. Periodista Daniel Saucedo Aranda s/n, 18071 Granada, Spain

Published online: 27 September 2013

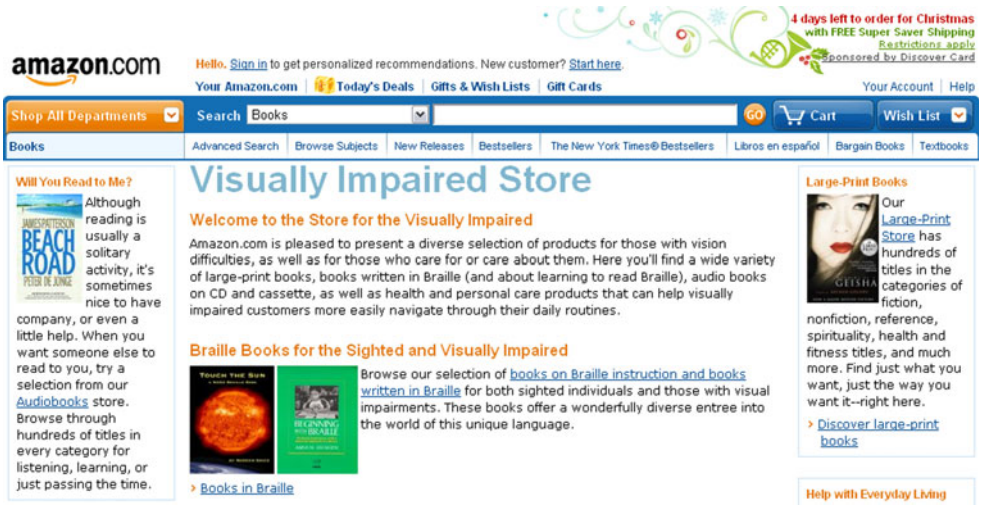


Fig. 1 Amazon's visually impaired store (http://www.amazon.com/Visually-Impaired-Books/b?ie=UTF8&node=14264821)

without breaking functionality. The authors argue that in the case of changing the presentation of Web applications to improve their external quality, refactoring may once again come to the rescue; it provides the ideal context to incite developers to discover how to improve interface metaphors for expert computer users as well as for novices or handicapped users, from an early design phase.

Refactoring, originally defined as *a disciplined process of restructuring source code with a cleaning purpose* [3], has been generalized in two directions: (1) to broaden its scope to other software artifacts, like databases [4] and wikis [5], and (2) to change its intent to other concerns, like look and feel embellishment [6]. The authors have recently proposed refactoring for the design models of a Web application with the goal of improving usability [7]. This article goes further and proposes the application of refactoring to improve the accessibility of Web applications for users of screen readers, without compromising usability for different audiences, leading toward the "*accessible usability*" of Web applications. With these refactorings, developers may change the presentation aspects of a working Web application, for which usage feedback proves inadequate or unusable. These are called Web interface refactorings (WIRs). They may be applied either at a design level (more appropriate when using a model-driven methodology) or at the implementation level. This article describes some WIRs and explains the transformation steps at both levels.

2 Refactorings for accessibility and usability in the development process: a taxonomy

In the interest of helping organizations that understand their social responsibility toward people with disabilities and the elderly, and that are willing to invest in incorporating accessibility to their websites, if the organization uses an agile development process, it is already applying code refactoring since it is an essential technique to improve and maintain internal quality through the whole agile process [8]. In this case, the organization should incorporate WIRs to improve the external quality of their software also through the whole process, looking for refactoring opportunities incrementally, by observing usage feedback after each cycle or sprint [9]. When refactoring is incorporated earlier in the process, even at a design level, developers have better chances to address client needs [8]. Similarly, when an organization shows social responsibility toward elderly and disabled users early, they are more susceptible to capture a larger user base, as described by the W3C[1]: "Organizations with accessible Web sites benefit from search engine optimization, reduced legal risk, demonstration of corporate social responsibility, and increased customer loyalty." If the organization uses a model-driven development, they have better chances of incorporating refactoring at an early design phase, since WIRs are also applicable on the presentation model of the application. The following section shows examples of WIRs that have been implemented on a tool for designing Web applications using the model-driven methodology UWE [10].

Besides the development phase in which refactoring is applied (by programmers, in the first case above or designers, in the second), there are two main approaches to incorporate Web accessibility that an organization should weight in terms of advantages and cost:

1. Incorporating accessibility in a separate version of the application: Building a separate interface makes it possible to reach the highest level of conformance with WCAG, and also to tailor the application for a specific disability, all without depriving other users of rich

[1] http://www.w3.org/WAI/bcase/.

Internet features. However, the cost for the organization is high: It requires an initial effort of creating a separate application and a subsequent and continuous effort of maintaining two applications' operative and consistent. As a consequence of this high cost, the accessible version is usually provided with less and outdated functionality, and consequently, this approach is generally not welcomed from end users. Many organizations do maintain a separate mobile version of the application mostly fully functional, which they provide as the accessible version. An example is the mobile version of Amazon,[2] which is WCAG-level A conformant; however, it does not provide certain functionality (search ordering, related searches, specifying a different shipping address for each product in the cart, etc.) and content (customer rating in product searches, product details, etc.).

2. Incorporating accessibility within the main application: The advantage of this approach is that it is not necessary to build and maintain a separate application to address disabled users. Nevertheless, the hard problem is to reach a balance in addressing usability for the disabled and not disabled audiences. Furthermore, it requires accommodating extra resources for accessibility (e.g., buttons for auditory descriptions of a video) and having them easily available, while providing nondisabled users with the main elements (navigation and content) also "upfront". An example is the Web site www.dbcde.gov.au of the Australian government, which conforms to WCAG at level AA.

From the viewpoint of maintaining an existing Web application, a refactoring that is applied to increase its external quality may be targeted at two different but interrelated aspects: (I) providing access to information that was previously unreachable (transformation on accessibility) or (II) enhancing the interaction with content that was previously available but with some difficulties (transformation on usability).

At the same time, an improvement (either accessibility or usability) may benefit: (A) a group of users with very specific characteristics that limit their interaction with the application or (B) a much wider and generic audience. Clearly, the latter is the most desirable one, but also the most difficult to achieve. Another problem, as mentioned above, is the case of changes aimed at improving the interaction of a particular group of users, although that might adversely affect users who are outside the group and vice versa.

The approaches 1–2, I–II, and A–B present complex interrelationships, which are important for an organization to understand in order to make an informed decision on how to approach accessibility and usability for their Web sites. Nevertheless, whatever path the organization takes, the refactoring technique may help reducing the cost of the initial investment and further maintenance. To better describe these interrelationships and define precisely the intent (accessibility and/or usability) and subject (specific audience and/or general audience) of WIRs, a taxonomy, sketched in Fig. 2, has been developed. This taxonomy also provides a framework to catalog not only the proposed WIRs but also future contributions that the present work may motivate.

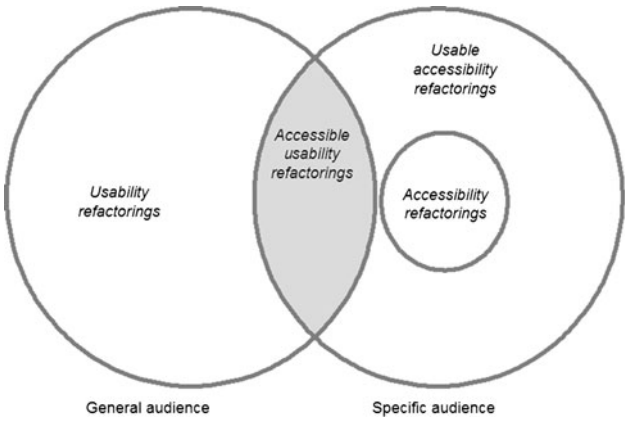


Fig. 2 Taxonomy of accessibility and usability refactorings

The set labeled "Specific audience" contains WIRs aimed at improving the accessibility offered to an audience with specific needs due to the access technology they use, their disabilities, or their personal and organizational limitations. These refactorings are focused on improving the accessibility of the application. Some of them are limited to allow access (*accessibility refactorings*), while others also include allowing such access in a pleasant, easy, and efficient way (*usable accessibility refactorings*).

The set labeled "General audience" contains WIRs aimed at improving the user experience during the interaction with the application, i.e., emphasizing usability. Within this set, there are refactorings that do not take into account, or are even harmful, to users with special needs (*usability refactorings*) from those that are also beneficial for disabled users (*accessible usability refactorings*). The latter appears as the intersection between *usability refactorings* and *usable accessibility refactorings*. They make it possible to have a single version of the Web site for all users (balancing usability and accessibility to handle the needs of a larger audience).

Table 1 summarizes the purpose of each type of WIR in the proposed taxonomy, also indicating the following: Whether it is applicable on the main or a separate version of a Web site, its classification, and an example.

[2] http://www.amazon.com/gp/aw/h.html.

Table 1 Types of refactorings

	Purpose	Version	Classification	Example
Accessibility refactorings (AR)	Make a Web site accessible, i.e., WCAG compliant	Main version	I A	Convert images to text
Usable accessibility refactorings (UAR)	Make an accessible Web site easier to use by a disabled person	Separate version	I II A	Postpone selection and/or operation
Usability refactorings (UR)	Make a Web site more usable (comfortable and organized) for a person without especial needs	Separate version	II B	Introduce information on demand
Accessible usability refactorings (AUR)	Improve the usability of a Web site for both disabled and non-disabled users	Main version	II A B	Split page

3 Web interface refactorings for usable accessibility and accessible usability

A WIR, by definition of refactoring, does not change the behavior of the application; rather, its purpose is to improve the way users access the application's content or trigger its behavior [7]. For instance, a WIR may move, copy, or replace the type of an interface component that allows executing an operation (e.g., changing a combination of checkbox + button for a single button), but it must not eliminate the possibility of executing that operation. In operations composed by several steps, such as checkouts, some of the steps that span multiple pages may be refactored into a single page (e.g., shipping and handling + credit card information) or divided into separate ones; however, none should be removed. Finally, content may be deleted from a page only when it is obsolete, duplicated, or unnecessary. Some of the refactorings that are more complex may usually be created as the composition of multiple simpler refactorings to create a more substantial change [7].

The targets of WIRs, i.e., the elements that a WIR may change, are as follows:

- layout structure and distribution of semantic sections;
- distribution of data and operations among available pages;
- content format, like color scheme, fonts, and background images;
- type of interface component used to display data or execute an operation (e.g., replacing a drop-down menu by a fixed menu);
- dynamic effects of components, like blink, sidle, or format change;
- interaction styles, e.g., mouse hover, double click, etc.;
- navigation aids, i.e., elements offered to browse links, like menus, lists, or tab rows.

3.1 Bad accessibility smells

Economic (and even methodological) reasons often push for building the "standard" application first, which is then improved with accessibility and/or usability. Fortunately, the use of this initial application helps detecting signs about the failure or poor compliance of any accessibility or usability principles. The case is similar for any kind of application, in which internal quality features are only recognized to be poor or defective following usage feedback and gradually improved throughout the development cycle. The refactoring literature calls "bad smells" to the signs of problems in the code that degrade its internal quality [3]. In the case of accessibility and usability, such problems are so-called "bad accessibility smells" and "bad usability smells" [7], respectively.

Some "bad accessibility smells" are as follows:

- Long navigation paths to achieve a goal, which shows problems of synthesis and efficiency;
- Saturated pages, i.e., large pages with mixed content, reflecting problems of synthesis and comprehension;
- Heavy use of visual annotations or floating menus;
- Unpredictable amount of content, or excessive detail upfront;
- Operations appearing before the data on which they apply, which require reading the list of operations twice, before and after reading, and selecting the data for the operation.

"Bad accessibility smells" arise from many sources in the literature (e.g., [1, 2, 5]), besides the current practice and feedback of the users. Another important source for "bad accessibility smells" emerges from poor or non-compliance with WCAG accessibility guidelines and WAI-ARIA technology[3] (the Accessible Rich Internet Applications Suite). WAI-ARIA focuses on dynamic Web content widgets and complex interaction through the mouse, or those that show difficulties for being accessed using a screen reader, for example drag and drop elements, tree widgets that present hierarchical contents, or the excessive use of DIV tags to divide sections instead of more

[3] http://www.w3.org/TR/wai-aria.

semantically appropriate ones (especially in RIA applications with the heavy use of AJAX).

Identifying "bad accessibility smells" in the design or interface of a particular application is an important step in the development process, as much as it is identifying bugs, since they provide the motivation for refactoring. This step should be included after acceptance testing or usage feedback. The following section goes one step further in helping developers and associates "bad smells" with the WIRs that fix them, thus pointing to the specific solution for each problem.

3.2 Initial catalog of Web interface refactorings

Table 2 lists some *usable accessibility refactorings* (UARs) and *accessible usability refactorings* (AURs). Each refactoring is associated with the "bad smells" it aims to solve, thus helping developers select the best refactoring for each problem. The third column in the table describes the interface elements that each refactoring changes. They are grouped in two coarse sections according to their main purpose, i.e., to improve access to content or navigation.

Six WIRs are described below: three usable accessibility refactorings (UARs) and three accessible usability refactorings (AURs). Regarding accessibility, they are specially targeted for visually impaired users. The refactorings are described in terms of the following:

- *bad smells* as already explained, each refactoring is triggered by one or more specific accessibility or usability problems; the citation in a "bad smell" indicates the source that identifies that fact as a problem;
- *motivation* whereas the "bad smells" are short sentences, the motivation has a description of the problem with examples;
- *transformation* describes the steps required in order to apply the refactoring. Since most WIRs may be applied at the level of design models or in the code, the transformation steps at both levels for some of them are described.

The refactorings are illustrated in the description of the case study presented in Sect. 5.

3.2.1 Replace non-accessible menu by list of links (UAR)

Bad smells Disguised operations, options, or navigation structure [2]; Unpredictable number of operations [1]; Screen reader users needing to read the menu again.

Table 2 Catalog of Web interface refactorings

Refactoring	Bad smells	Change applied
Simplify content		
Split page (AUR)	Saturated page	Layout and navigation structure; distribution of contents/operations
Focus-dispersed content and/or operations (AUR)	Confusing organization; Long navigation paths	Distribution of contents and/or operations
Add content summaries (AUR)	Saturated page; Abuse of scrolling; Useless link activations	Layout and navigation structure
Add size indicators (UAR)	Unpredictable size; Users quitting before completing a goal	List and tables
Remove unnecessary content (UAR)	Obsolete content; Duplicated content; Large page	Content
Simplify navigation		
Merge pages (AUR)	Long navigation paths	Layout and navigation structure; distribution of contents/operations
Replace non-accessible menus by list of links (UAR)	Disguised navigation structure	Type of interface component
Show all structural links at once (AUR)	Disguised/spread navigation structure	Layout structure; interaction style
Fix menu (AUR)	Excessive backward scrolling toward menu	Section layout
Postpone selection and/or operation (UAR)	Reading twice through an element to select it	Selection checkbox placement
Distribute menu for an elements container (UAR)	Difficult interaction to apply operations on elements	Distribution of operations

Motivation Non-accessible menus are those that appear in a drop-down fashion, or in a pop-up, and those that display their options one-by-one as the user hovers over an image. They are also unusable since they make their available options difficult to spot even for a sighted user [2] (for this reason, this refactoring may be also classified as AUR). Menus that appear in a table row may be accessible, but not usable for a visually impaired person, since the number of options is unknown, and so is the end of the menu. Lists are accessible and usable containers since all their elements are shown at once, and screen readers say the number of elements at the beginning of a list.

Transformation Gather all menu elements and arrange them in a single list of links in decreasing order of usage frequency. For instance, several Web pages have menus, which are built by using DIVs. Typically, the menu sub-items are hidden until the mouse hovers over a particular element. The transformation in this case implies replacing DIVs around menu items by a bulleted list (e.g., using UL tag), which will make the menu sub-items visible and will cause any screen reader to say the number of elements upfront.

3.2.2 Postpone selection and/or operation (UAR)

Bad smells Difficult interaction to apply operations on elements of a list or table; Reading twice through an element to select it.

Motivation In order to apply an operation on a list or table of elements, a common approach is used to provide a drop-down menu of operations at the top of the table and checkboxes on each line to select the target elements. Since checkboxes appear before the description of each element, a screen reader user needs to go through the line to listen to the contents first and then go back to the checkbox at the beginning of the line for selecting it. Once the user selects some elements, she/he needs to go back to the top and browse through the operations again. These interactions become too complex for a visually impaired person that does not know the keyboard shortcuts to move fast through the page. Even when WCAG recommend placing menus and checkboxes in their most predictable position, the blind users in the experiments conducted in this study clearly showed a preference in the ability of selecting an item or operation on it, just after reading it.

Transformation Move the column containing the checkboxes to the last column in a table of elements, so a user can select an element after reading or hearing it; move or copy the menu of available operations after the table or container.

3.2.3 Remove unnecessary content (UAR)

Bad smells Obsolete content; duplicated content [1]; large page [2]; unused operations [2].

Motivation Unnecessary content refers to obsolete data that were left in a page as dead code that developers forget to remove. Sighted users may easily skip it; however, it is not that easy for screen reader users. Unnecessary content also refers to duplicated data, links, or operations which are made available in several places to be easier to spot, though again, cumbersome to read through it many times. Other times, a site provides operations that users never select and should be removed because it all contributes to a large page.

Transformation This refactoring may be applied at a higher level of abstraction, in the presentation model of a Web application. For example, in the UWE design notation [10], each page is modeled as a stereotyped class with a set of attributes, each attribute corresponding to a data item, input field, link, or operation, inside a box that represents its abstract location (Fig. 3 shows an example). Duplicated data, links, or operations are easier to spot at this level, and the corresponding attribute may be easily selected and deleted.

3.2.4 Split page (AUR)

Bad smells Overloaded page [2]; complex and heterogeneous page [2]; confusing organization.

Motivation A large page that requires too much scrolling is annoying for any user. Large pages should only be destination pages [2], with detailed information that some users are willing to scroll to read, but should be avoided for intermediary, or home pages where most users want to quickly figure out whether the site can handle their needs. Furthermore, a page that mixes too many concepts or several operations that should have been presented in stages is confusing, loses the focus of the reader, and makes it hard to find specific information. For people with visual impairments using a screen reader, a large page with mixed content is frustrating and multiplies the time to find a specific content.

Transformation This refactoring is also applicable at design time, in the presentation model of the application. In the case of using the UWE notation, the page model that is found to be complex and heterogeneous must be divided into a structure of interrelated pages, reorganizing semantic, layout, and navigation structure so that each new page is simpler and more cohesive. For this purpose, each set of cohesive content is moved into a new page and replaced in the original page by a link to the new page. Figure 3 shows

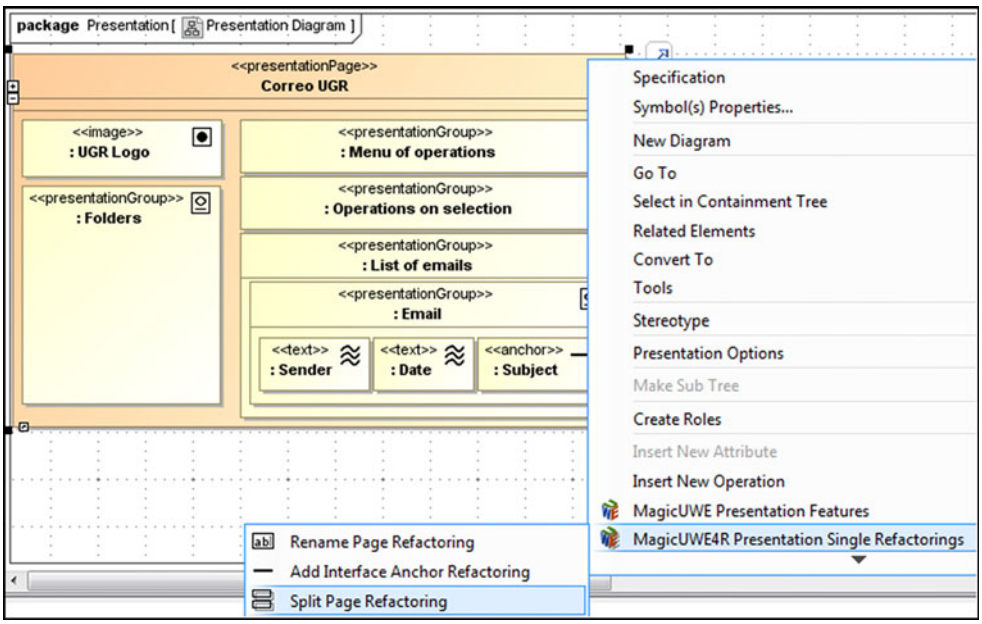


Fig. 3 Applying split page over the presentation model through a menu option

a simplified version of the presentation model of the case study's home page, made with the tool MagicUWE [11], which has been extended with model refactorings such as *Split Page*.

3.2.5 *Merge pages (AUR)*

Bad smells Long navigation paths [2].

Motivation Although the "three click rule" may be too strict, unnecessary long navigation paths to reach a goal are uncomfortable for any audience [2]. Users are not really patient if they feel like wasting their time in searching for something without getting any more information in return. Unsighted users always complain about this problem since it makes them feel frustrated. Moreover, splitting a page into several pages may result in some of the new pages being too small; if some of them are related, they may be merged.

Transformation Also applicable at the model level, the content attributes of a small page may be moved into another existing page with related contents. Consider renaming the existing page and the anchors for incoming links to refer to the added content, so that users will still find the contents of the merged page.

3.2.6 *Add content summaries (AUR)*

Bad smells Large page; excessive scrolling; excessive detail upfront [2].

Motivation The "bad smells" of this refactoring are similar to those of *Split Page*, implying that this is a different solution for a related problem. Upon a large page requiring scrolling or causing frustration because of mixed content, this refactoring offers an intermediate step that shows summaries of all content. This solution keeps a complete overview of content while avoiding excessive scrolling or unwanted link activations. Visually impaired users always benefit from shorter pages that prevent unnecessary navigation.

Transformation At the model or code level, add an intermediate page showing a synthesis of the content, and provide a link to follow for more information. In this way, users can have a quick understanding of core content and decide whether to navigate to any item.

4 Case studies and evaluation

The above refactorings have been applied to the email reader of the University of Granada (UGR).[4] This Web application is not accessible, so refactorings have been applied to make it accessible and also usable, especially for visually impaired users. The authors do not claim to have reached the best Web interface design, but one that allows all users (disabled and nondisabled) to access and complete more tasks than before. Additionally, WAI-ARIA technology still does not offer enough implementation to consider it in the redesign; however, future WIRs would benefit from the use of ARIA technology, enhancing the keyboard interaction mode with the application.

[4] https://webmail.ugr.es.

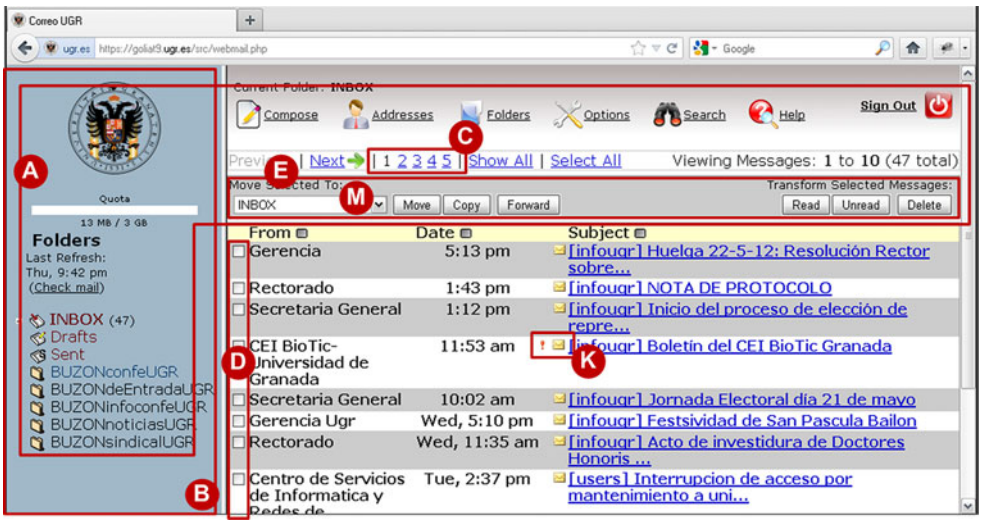


Fig. 4 Current email reader before refactoring showing some bad smells

4.1 Webmail UGR: users' perspectives

4.1.1 Discovering bad smells

The first step with the target application was to analyze it in search of "bad smells". For this purpose, different sources of "bad smells" in the literature (such as [1, 2] as discussed in Sect. 3.1) were used, besides accessibility assessments such as [12], which thoroughly describes the problems that blind people find in leading Webmail applications. Second, a preliminary test with a blind person was conducted, as well as short surveys with other blind persons (our focus group), enquiring about usual problems with Web applications and email clients. The following list contains the "bad accessibility smells" resulting from the previous analysis and survey on UGR Webmail. Figure 4 shows the application before refactoring, and some of these "bad smells":

(A) *Overloaded pages*: All pages contain the list of folders and all operations at the top, even when they do not apply to the current data. Having a screen reader go through all this in every page gets very tedious.

(B) *Use of frames*: Frames are known to cause usability and navigation issues. When using a screen reader, it is hard to move between frames with keystrokes, which made it quite hard to access the folders.

(C) *Unnecessary links*: The links to each page number are rarely used, but take a long time to read if there are many pages.

(D) *Difficulty to select emails*: In the email list, the selection checkbox for each email appears before the subject, forcing the reader to go through it before knowing the referred email, eventually having to go back to the checkbox.

(E) *Operations before data*: Similarly to checkboxes, operations appear before the data on which they apply, which complicates their application.

(F) *Confusing organization*: There is a "Folders" heading in the left frame with a list of folders, and a "Folders" button at the top menu, which does not navigate to a list of folders as one would expect, but allows creating and deleting a given folder. This is confusing even for sighted users.

(G) *Unpredictable amount of operations*: Users of screen readers have to hear through an excessively large set of operations listed at the top of each page, without knowing how many there are.

(H) *Users quitting before completing a goal*: Testing with screen reader users showed that it was very difficult to change folders since they had to change to the left frame, and keyboard combinations did not work.

(I) *Excessive detail up front*: All the email fields are listed when displaying an email, and the majority of these details are generally unnecessary.

(J) *Absence of returning links*: Some pages do not provide a link to go back to the previous page or canceling the operation, like in the email composition page.

(K) *Missing Alt text for images*: like attachments and email priority in email list.

(L) *Intermediate blanks*: Blanks are used to separate each menu option on top, which makes the screen reader say "blank" too often.

(M) *Drop-down folder's menu goes unheard:* The screen reader says "combo box" for the drop-down menu with the list of folders, so users fail to select a folder.

4.1.2 Fixing bad smells through refactoring

The applied WIRs are listed below, and the final result appears in Fig. 5.

1. *Unframe application*: to solve the "bad smell" "B. Use of frames".

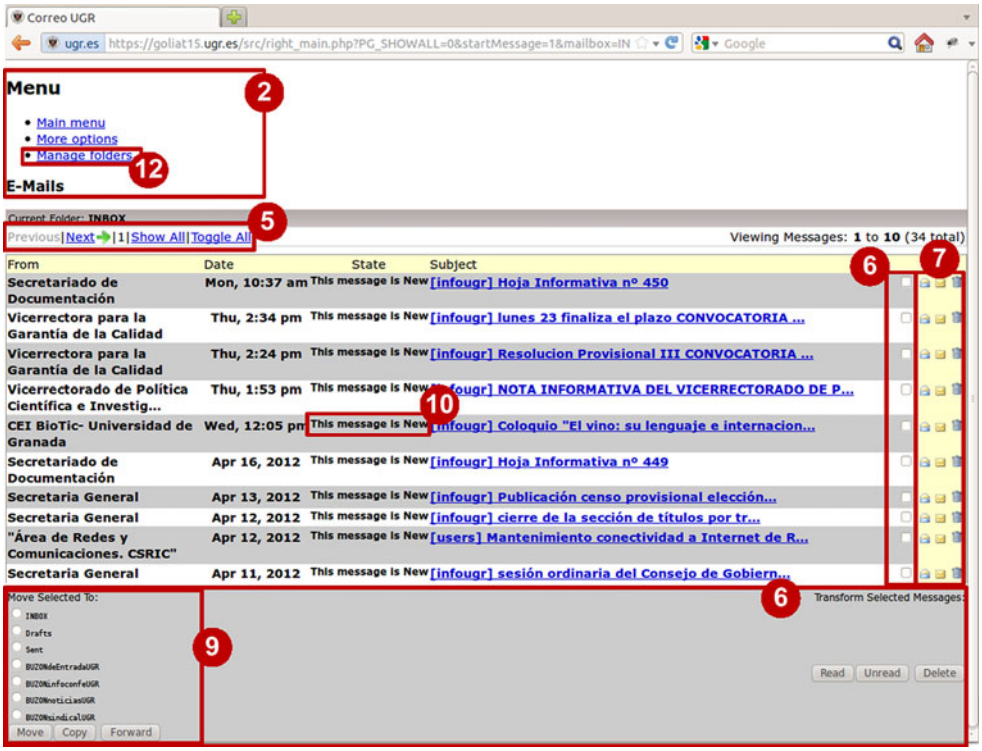


Fig. 5 Refactored UGR webmail

2. *Split page*: applied to solve "A. Overloaded pages," "F. Confusing organization," and "H. Users quitting before completing a goal." It was applied twice: (1) to split the left frame and (2) to split the top menu, which are now accessed through the menu options marked with circle "2" in Fig. 5.
3. *Merge pages*: The result of applying *Split Page* twice resulted in a new bad smell: "Long navigation paths." Thus, the email list and the list of operations were merged in the top menu in a single page.
4. *Replace non-accessible menu by list of links*: This solves "G. Unpredictable amount of operations" on top of the page.
5. *Remove unnecessary content* used to remove "C. Unnecessary links," in this case the links to each page number that lists emails.
6. *Postpone selection and operations*: Checkboxes were moved to the last column, so that a user can select an email after it was read, solving "D. Difficulty to select emails." Additionally, operations were moved after the table to solve "E. Operations before data".
7. *Distribute menu for an elements container*: to facilitate the local application of an operation right after the email is read, also solving "E. Operations before data," this refactoring was applied to attach the list of operations to each email at the end of the row.
8. *Add content summary*: to solve "I. Excessive detail upfront," when navigating to an email, it shows only sender and subject, proving a link to the rest of the email fields.
9. *Transform drop-down into radio buttons' group:* The drop-down list of folders to choose where to move selected emails is transformed into a group of radio buttons, thus solving "M. Drop-down folder's menu goes unheard".
10. *Replace image by text:* All images (<img>) were replaced by their *title* and/or *alt* text, solving "K. Missing Alt text for images".
11. *Delete blank spaces:* Blanks were removed from all pages solving "L. Intermediate blanks".
12. *Improve description*: The caption for the "Folders" button was replaced for "Manage Folders," clarifying its purpose and solving "F. Confusing organization".

4.1.3 Evaluation of UGR Webmail

An evaluation of UGR Webmail was conducted with 24 sighted and nonsighted users. Most of them (92 %) already knew the application, and half of them use it with high frequency. The participants were divided in 4 groups, namely UGR_{JWS}: 6 unsighted participants used the normal application with Jaws; $ReUGR_{JWS}$: 6 unsighted participants used the refactored version with Jaws; *UGR*: 6 sighted participants used the normal application; *ReUGR*: 6 sighted participants used the refactored application.

The experiment was composed of three tasks:

Table 3 Number of users that completed each task

Task	UGR_{JWS}	$ReUGR_{JWS}$	UGR	ReUGR
I.1	5	6	6	6
I.2	1	6	6	6
I.3	2	6	6	6
II.1	3	6	6	6
II.2	3	6	6	6
III.1	0	5	3	6
III.2	2	6	6	6

I. (1) Find an email from a given sender in the inbox, (2) read it, and (3) reply answering a question that appears in the email;
II. (1) Search the reply message in the sent folder and (2) delete it;
III. (1) Return to the inbox, find the first message with high priority and (2) log out.

Table 3 shows the number of users in each group that were able to complete the tasks. Note that, for group UGR_{JWS}, none of the tasks were completed by the whole group of 6 users. The main problems reported (besides the listed "bad smells") were as follows:

- the presence of the frame and its incompatibility with Jaws made it really hard to change folders: Some of the users tried Jaws' keystroke combination to change frames without success;
- the absence of Alt text for the image showing an email's priority made it impossible to accomplish task III.1 (find a message with high priority);
- Jaws' accent and speed were hard to understand, mostly because the setting was different than what they had at home.

In contrast to group UGR_{JWS}, group $ReUGR_{JWS}$ completed all tasks except for just one person in one task. In the case of sighted participants, half of them could not complete task III.1 (they did not know the image for high priority) in the normal application, while all sighted participants completed all tasks with the refactored version. Figure 6 shows the box plots with the time to accomplish each task in seconds. In order to include in the box plots all the results, even those for unaccomplished tasks, a sufficiently large number (1,000 s) were used, to represent a task that a user could not complete (the largest time to accomplish a task was 900 s.).

4.1.4 Threats to validity and discussion

Looking at Fig. 6, one notices that in general, the refactored versions received better results than the normal ones. The differences are especially significant for the normal and refactored versions in unsighted participants (UGR_{JWS} vs. $ReUGR_{JWS}$). For example, in Task I.1, 75 % of unsighted users of $ReUGR_{JWS}$ lie below the median of unsighted users of UGR_{JWS}. In Task I.3, the maximum time for sighted users of ReUGR lies below the median of sighted users of UGR. There are four cases where the participants took longer time with the refactored version; however, this coincides with frequent users of UGR Webmail, implying that they are highly accustomed to its normal interface.

Sighted users reported that the inbox was cleaner and straightforward, although accessing folders through a different page took longer. They suggested improving the design of the refactored pages, which they did not find appealing, although this was not the objective at this time. Unsighted users required more time to get used to the new interface and to read through each page to find the new place for each operation. After granting them that time, they indicated that their experience was more pleasant using the refactored application. Some positive points that they emphasized were as follows:

- the new structure provides less crowded pages, and blanks are not read, which allows reaching targets faster in all three tasks;
- the elimination of vertical frames and the split of folders in a separate page allows moving from the inbox to the folder list, while this was too difficult or impossible to do with the standard application, and prevented them from using folders. This change was imperative to complete the second task;
- textual information provided instead of unnamed graphics made it possible to learn from the email list if one had an attachment or priority and thus crucial to complete the third task.

Both sighted and unsighted participants used Jaws. It was observed that it is very important to find a comfortable configuration of Jaws for each user since otherwise, it may be too hard to understand, and this was reported as a real drawback. It was also observed that in general, visually impaired persons do not know/use many Jaws' keyword commands besides the arrows to move (combining them with Ctrl) and spacebar to follow links.

4.2 WebMail Horde: developers' perspective

A second case study was conducted to measure the effort needed for a developer to implement some of the refactorings. Instead of UGR, Horde was used as it is a larger and highly configurable, open source Webmail application. Since the authors were not previously familiar with Horde, the numbers obtained represent a maximum value, as they also include the cost of learning the code first.

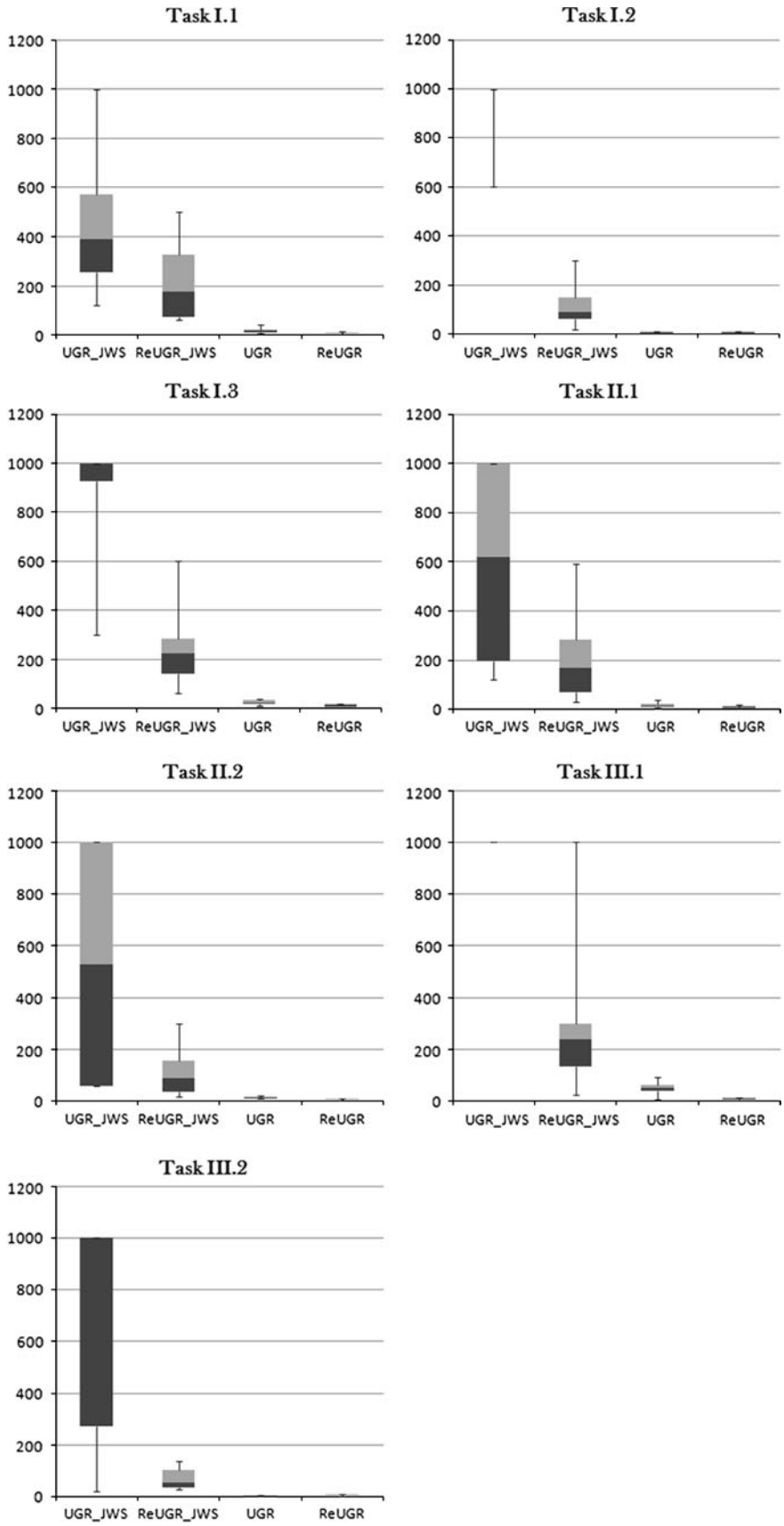


Fig. 6 Box plots with 25, 50, and 75 % for the first four tasks

Table 4 Cost of implementing refactorings in Horde

Refactoring	Time (min)	Modified LOC	Added LOC	Changed files
Distribute menu	125	7 CSS lines	4 CSS lines 15 HTML lines	3
Split page V1	45	5 HTML lines	25 HTML lines 50 JavaScript lines	2
Postpone selection	20	4 HTML lines 8 CSS lines	–	2

Three of the most representative refactorings were selected: *Split Page*, *Distribute Menu,* and *Postpone Selection*. The measurements taken for each selected refactoring were as follows: the time required to apply it, the number of updated lines of code (LOC) and its types: CSS, HTML, etc., the number of lines of code added, and finally, the number of modified files. Table 4 summarizes the results.

The refactoring *Distribute Menu* was the most time-consuming because of the time to understand the Webmail source code. Note that, while *Distribute Menu* needed 125 min solely for updating 7 CSS lines and adding both 4 CSS and 15 HTML lines, *Split Page* needed 45 min with much more effort in the modification itself (5 HTML lines modified, 25 HTML lines, and 50 JavaScript lines added). The *Split Page* refactoring was developed using JavaScript, since this Webmail application uses this kind of technology for building the UI, so emphasis was given in keeping the same style. *Postpone Selection* was easily applied by changing 4 HTML lines and other 8 CSS lines that took 20 min.

In this way, it can be seen that Web application accessibility may be improved by applying few changes on source code. The hardest refactoring to implement was *Split Page*; however, it would be easier to apply it without using JavaScript, as described in Sect. 3.2.4.

5 Related work

Since the interface is a critical aspect of a Web site, many approaches to improve its appearance and interaction styles have emerged in recent decades. Two of them are interaction patterns and transcodings.

An interaction pattern presents a solution to a recurring problem in the design of Web interfaces, from abstract solutions like "Instant gratification" to concrete ones such as "One window drilldown" [13]. In contrast, an interface refactoring specifies a step-by-step transformation to achieve an enhancement on a Web interface. Thus, refactorings could be used to apply interaction patterns on an existing design or a running application [14], similarly to how traditional refactorings may introduce design patterns [15]. For example, the refactoring "Introduce Information on Demand" [7] may be used on a cluttered interface to apply the pattern "Information on Demand" [16], which uses the same screen area to show different contents as the user hovers over menu titles.

Transcoding is a technique that applies on-the-fly transformations on a working application, normally based on semantic annotations previously made by developers or derived from Web design models [17]. For example, transcodings may replace a floating menu for a static one, a link menu for a linear table, or add size indicators in dynamic lists. Refactoring of graphical interfaces comes close to transcoding, though several differences between both approaches have been identified:

- transcodings do not necessarily preserve behavior, as they may remove some operations, for example when they aim to simplify content [18]; in contrast, refactorings were conceived as behavior-preserving transformations [3], which in the case of Web applications means preserving content and functionality [7];
- transcodings apply on HTML code [17], while refactorings can work either at the implementation or design levels [7];
- refactorings are by nature composable and can be applied incrementally to create larger changes [3]. This allows for a sequence of refactorings to be applied incrementally, and developers may choose to inject gradual transformations thus avoiding user disorientation with radical changes. Instead, transcodings do not necessarily compose and may even interfere with each other [17];
- refactorings allow a wider range of transformations because they do not need to be applied on-the-fly; they are part of the developer armory. For a complex refactoring like merging related node classes at the navigation model [14], the annotations in the "equivalent" transcoding may not be manageable. The difficulty augments, if one considers the increasingly dynamic nature of Web applications, which makes it very hard to update the annotations (in spite of the progress that transcoding is doing to reduce this problem, such as differential analysis [19]).

There are also usability and accessibility studies, including for example the work of Wentz and Lazar [12], which presents an evaluation performed by blind users on the leading desktop and Webmail applications. This evaluation was used to identify usual "bad smells" in email clients. Some examples of the problems they report are as follows: loses focus on screen reader cursor (all-tested applications); basic HTML version easier to use but with

less features (Gmail, Yahoo); too many links (Hotmail); difficult to navigate (Gmail, Hotmail, Yahoo); and saturated page with advertising (Yahoo) [12].

6 Conclusions and future work

The solutions that the presented WIRs offer allow developers to apply well-known Web usability patterns and provide accessible and useful interfaces that allow visually impaired users interact with Web pages in a more comfortable way. Current work includes measuring the improvement gain in usability and accessibility for a better understanding of the consequences of each refactoring and automating the refactorings in some development environment to offer developers a tool that can apply the refactorings, log them, replay them, and roll them back. Furthermore, work is carried out on how to customize refactorings for each user through client-side scripting [20].

While Web applications continue to hit the mainstream and social networks offer unimaginable opportunities, technology should not be allowed to hinder usability for all audiences. Developers should strive to provide universal access, and researchers must find efficient techniques and tools to help developers in this challenging goal. If they can attain this goal consuming fewer resources, managers will be more willing to invest in it and change accessibility policies. The use of refactoring toward this goal is thus advocated as an incremental and systematic process of finding opportunities for quality improvement (identifying "bad smells") and producing safe transformations toward universal access.

References

1. W3C: Web Content Accessibility Guidelines (WCAG) 2.0, http://www.w3.org/TR/WCAG20 (2008). Accessed 10 July 2012
2. Nielsen, J.: Designing Web Usability: The Practice of Simplicity. New Riders Publishing, Indianapolis (2000)
3. Fowler, M.: Refactoring: Improving the Design of Existing Code. Addison-Wesley, Boston (1999)
4. Ambler, S.W., Sadalage, P.J.: Refactoring Databases: Evolutionary Database Design. Addison-Wesley, Boston (2006)
5. Puente, G., Diaz, O.: Wiki Refactoring as Mind Maps Reshaping. In: 24th International Conference on Advanced Information Systems Engineering (CAiSE'12) Poland, pp. 646–661
6. Harold, E.R.: Refactoring HTML: Improving the Design of Existing Web Applications. Addison-Wesley, Boston (2008)
7. Garrido, A., Rossi, G., Distante, D.: Refactoring for usability in web applications. IEEE Softw. **28**(3), 60–67 (2011)
8. Beck, K., Andres, C.: Extreme Programming Explained: Embrace Change, 2nd edn. Addison-Wesley, Boston (2004)
9. Schwaber, K., Beedle, M.: Agile Software Development with Scrum. Prentice Hall, Upper Saddle River (2001)
10. Koch, N., Knapp, A., Zhang, G., Baumeister, H.: UML-based web engineering: an approach based on standards. In: Rossi et al. (eds.) Web Engineering: Modelling and Implementing Web Applications. Human–Computer Interaction Series, pp. 157–191. Springer, New York (2008)
11. Busch, M., Koch, N.: MagicUWE—a CASE tool plugin for modeling web applications. In: Proceedings of 9th International Conference Web Engineering (ICWE'09), Springer, Berlin, LNCS 5648: 505–508
12. Wentz, B., Lazar, J.: Usability evaluation of email applications by blind users. J. Usability Stud. **6**(2), 75–89 (2011)
13. Tidwel, J.: (2011) Designing Interfaces: Patterns for Effective Interaction Design. O'Reilly Media Inc, California
14. Garrido, A., Rossi, G., Distante, D.: Systematic improvement of web application design. J. Web Eng. **8**(4), 371–404 (2009)
15. Kerievsky, J.: Refactoring to Patterns. Addison Wesley, Boston (2004)
16. Rossi, G., Schwabe, D., Garrido, A.: Design Reuse in Hypermedia Applications Development. In: Proceedings of the eighth ACM Conference on Hypertext (Hypertext'97). Southampton, United Kingdom, pp. 57–66 (1997)
17. Asakawa, C., Takagi, H.: Transcoding. In: Harper, S., Yesilada, Y. (eds.) Web Accessibility: A Foundation for Research, pp. 231–261. Springer, New York (2008)
18. Huang, A.W., Sundaresan, N.: Aurora: a conceptual model for Web-content adaptation to support the universal usability of Web-based services. In: Proceedings of the 2000 Conference on Universal Usability, Virginia, United States, pp. 124–131 (2000)
19. Asakawa, C., Takagi, H.: Annotation-based transcoding for nonvisual web access. In: Proceedings of the 14th International ACM Conference on Assistive Technologies, Virginia, USA, pp. 172–179 (2000)
20. Garrido, A., Firmenich, S., Rossi, G., Grigera, J., Medina Medina N., Harari, I.: Personalized web accessibility using client-side refactoring. IEEE Internet Comput. *To appear* (2013)