An Incremental Approach for Building Accessible and Usable Web Applications

Nuria Medina Medina[1], Juan Burella[2,4], Gustavo Rossi[3,4],
Julián Grigera[3], and Esteban Robles Luna[3]

[1] Departamento de Lenguajes y Sistemas Informáticos, Universidad de Granada, España
nmedina@ugr.es
[2] Departamento de Computación, Universidad de Buenos Aires, Argentina
jburella@dc.uba.ar
[3] LIFIA, Facultad de Informática, UNLP, La Plata, Argentina
{gustavo,julian.grigera,esteban.robles}@lifia.info.unlp.edu.ar
[4] Also at CONICET

Abstract. Building accessible Web applications is difficult, moreover considering the fact that they are constantly evolving. To make matters more critical, an application which conforms to the well-known W3C accessibility standards is not necessarily usable for handicapped persons. In fact, the user experience, when accessing a complex Web application, using for example screen readers, tends to be far from friendly. In this paper we present an approach to safely transform Web applications into usable and accessible ones. The approach is based on an adaptation of the well-known software refactoring technique. We show how to apply accessibility refactorings to improve usability in accessible applications, and how to make the process of obtaining this "new" application cost-effective, by adapting an agile development process.

Keywords: Accessibility Visually Impaired, Web engineering, TDD, Web requirements.

1 Introduction

Building usable Web applications is difficult, particularly if they are meant for users with physical, visual, auditory, or cognitive disabilities. For these disadvantaged users, usability often seems an overly ambitious quality attribute, and efforts in the scientific community have been generally limited to ensure accessibility. We think accessibility is a good first step, but not the end of the road. Usability and accessibility should go hand in hand, so disabled users can access information in a usable way, since it is not fair to pursue usability for regular users and settle with accessibility for disabled users. Thus we consider that the term Web accessibility falls short and should be replaced by the term "usable web accessibility" or "universal usability", whose definition can be obtained from the combination of the two quality attributes. As an example, let us suppose a blind person accessing a (simplified) Web application

L. Chen, P. Triantafillou, and T. Suel (Eds.): WISE 2010, LNCS 6488, pp. 564–577, 2010.
© Springer-Verlag Berlin Heidelberg 2010

like the one shown in Figure 1, using a screen reader [1]. To enforce our statement, we assume that this application fulfils the maximum level of accessibility, AAA, according to the Web Content Accessibility Guidelines (WCAG) [2]. This means that the HTML source of the application satisfies all the verification points, which check that all the information is accessible despite any user's disabilities. However, using the screen reader, it will be difficult for the blind user to go directly to the central area of the page where the books' information is placed. On the contrary, he will be forced to listen (or jump) one by one all the links before that information can be listened (even when he does not want to use them).

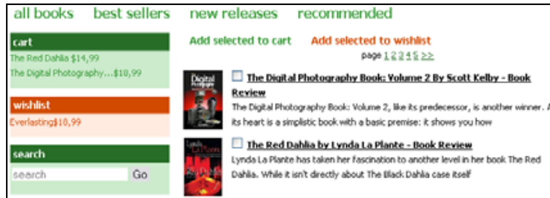

Fig. 1. Accessible but not usable page

The main problem, that we will elaborate later, is that this page has been designed to be usable by sighted users and "only" accessible by blind users. In this paper we present an approach to systematically and safely transform an accessible web application into a usable and accessible (UA) one. The approach consists in applying a set of atomic transformations, which we call accessibility refactorings, to the navigational and interface structure of the accessible application. With these transformations we obtain a new application that can be accessed in a more friendly way when using, for example, a screen reader. We show, in the context of an agile approach, how this strategy can be made cost feasible, particularly when the application evolves, e.g. when there are new requirements. Our ideas are presented in the context of the WebTDD development approach [3], but they can be applied either in model-driven or coding-based approaches without much changes. Also, while the accessibility refactorings we describe are focused on people with sight problems, the approach can be used to improve usability for any kind of disability.

The main contributions of the paper are the following: a) we introduce the concept of usable accessibility b) we show a way to obtain a UA web application by applying small, behaviour-preserving, transformations to its navigation and interface structures; c) we demonstrate the feasibility of the approach by showing not only how to generate the UA application but also how to reduce efforts when the application evolves. The rest of the paper is structured as follows: Section 2 discusses some related work in building accessible applications; section 3 presents the concept of accessibility refactoring and briefly outlines some refactorings from our catalogue. In section 4 we present the core of the approach using the example of Figure 1, and finally section 5 concludes the paper and discusses some further work we are pursuing.

2 Related Work

Web usability for visually impaired users is a problem that is far from being solved. The starting point of the proposed solutions leans on two basic supports: the WCAG Guidelines [2] and the screen readers [1] used together with traditional or "talking" browsers [4]. Then, the methods and proposed tools to achieve accessibility and usability in the Web diverge in two directions [5]: assessment or transformation. In the first group, the automated evaluation tools, such as Bobby (Watchfire) [6], analyze the HTML code to ensure that it conforms to accessibility or usability guidelines. In the second group, automated transformation tools help end users, rather than Web application developers. These tools dynamically modify web pages to better meet accessibility guidelines or the specific needs of the users.

Automated transformation tools are usually supported by some middleware, and they act as an intermediary between the Web page stored in the server, and the Web page shown in the client. Thus, in the middleware, diverse transcodings are performed. An example is the middleware presented in [7], which is able to adapt the Web content on-the-fly, applying a transcoding to expand the context of a link (the context is inferred from the text surrounding the link), and other transcoding to expand the preview of the link (processing the destination of the link). Other example is the proposal in [8], in which semantic information is automatically determined from the HTML structure. Using these semantics, the tool is able to identify blocks and reorganize the page (grouping similar blocks, i.e. all the menus, all the content areas, etc). This will create sections within the page that allow users to know the structure of the page and move easily between sections (ignoring non essential information for him). However, none of these automatic transcodings are enough to properly reduce the overhead of textual and graphic elements, as well as links, which clutter most pages (making their reading through a screen reader very noisy). This is because discerning meaningful from accessory content is a task that must be manually performed. A basic example of "manual" transcoding is the accessible method proposed in [9], which uses stylesheets to hide text (marked with a special label) from the page prepared for sighted users. Another interesting example is Dante [10], a semi-automated tool capable of analyzing Web pages to extract objects which are meaningful for the handicapped person during navigation, discover their roles, annotate them and transform pages based on the annotations. In [11] meanwhile, Dante annotations are automatically generated in the design process. In this case, the intervention of the designer is performed in the phase of modeling, but still needed.

We believe that the problem of usability for impaired people must be attacked from the early stages of applications design. Furthermore, all stakeholders (customers, designers and users) must be involved in the process. Hence, instead of proposing an automatic transcoding tool, we provide a catalogue of refactorings that the designer can apply during the development process, and later during the evolution of the Web application. The catalogue is independent of the underlying methodology and development environment, so refactorings can be integrated into traditional life cycle models or agile methodologies. However, to emphasize our point we show how a wise combination of agile and model-driven approaches can improve the process and allow the generation of two different applications, one for "normal" users and another, which provides usable accessibility for impaired users.

3 Making Accessible Web Applications More Usable

Achieving universal usability is a gradual and interdisciplinary process in which we should involve all application's stakeholders. In addition, we think that it is a user-centred process that must be considered in early phases of the design of Web applications. For the sake of conciseness, however, we will stress out the techniques we use, more than the process issues, which will be briefly commented in Section 4.

The key concept in our approach is refactoring for accessibility. Refactoring [12] was originally conceived as a technique to improve the design of object-oriented programs and models by applying small, behaviour-preserving, transformations to the code base, to obtain a more modular program. In [3] we extended the idea for Web applications with some slight differences with respect to the original approach: the transformations are applied to the navigational or presentation structures, and with the aim of improving usability rather than modularity. In this context we defined an initial catalogue of refactorings, which must be applied when a bad usability smell [13] is detected. More recently, in [14] we extended the catalogue incorporating a new intent: usable accessibility. As said before, we will concentrate on those refactorings targeted to sight disabled persons. Subsequently, section 3.1 briefly describes the specific catalogue of refactorings to achieve UA for sight impaired users.

3.1 The Refactoring Catalogue

Each refactoring in our catalogue to improve UA, specifies a concrete and practical solution to improve the usability of a Web application, that will be accessed by a visually impaired user. Each UA refactoring is uniformly specified with a standard template, so it can be an effective means of communication between designer and developers. The basic points included in the template are three: purpose, bad smells and mechanics. The purpose, defined in terms of objectives and goals, establishes the property of usability to be achieved with the application of the refactoring. The bad smells are sample scenarios in which it is appropriate to apply the refactoring, that is, elements or features of the Web site which generate a usability problem. Finally, each mechanics explains, step by step, the transformation process needed to apply the refactoring and thus solve the existing usability problem.

The refactorings included in the catalogue are divided in two groups: Navigation Refactorings and Presentation Refactorings. Navigation refactorings try to solve usability problems related to the navigational structure of the Web application. Therefore, the changes proposed by this first type of refactorings modify the nodes and links of the application. Presentation refactorings meanwhile propose solutions to usability problems whose origin are the pages' interfaces. Therefore this second type of refactorings implies changes in the appearance of the Web pages.

Concretely, the navigation refactorings included in the UA catalogue allow: to split a complex node, to join two small nodes whose contents are deeply related, to make easier the access between nodes creating new links between them, to remove an unnecessary link, information or functionality with the aim to simplify the node without losing significant content or, conversely, to repeat a link, functionality or information contained on a node in another node where it is also necessary and its inclusion does not overload the resulting node, etc.

Presentation refactorings included in the UA catalogue determine when and how: divide a complex and heterogeneous page in a structure of simpler pages, combine two atomic pages in a cohesive page, add needed anchors, remove superfluous anchors, add contextual information such as size indicators in dynamic list and tables, distribute or duplicate the options of a general menu for each one of the items valid for the menu, replace pictures and graphics for an equivalent specific text or remove the figure if it is purely aesthetic, reorder the information and functionality on the page in a coherent order to read and use, reorganize panels and sections to be read from top to bottom and from left to right, fix the floating elements, transform nested menus into linear tables more easier to read, etc.

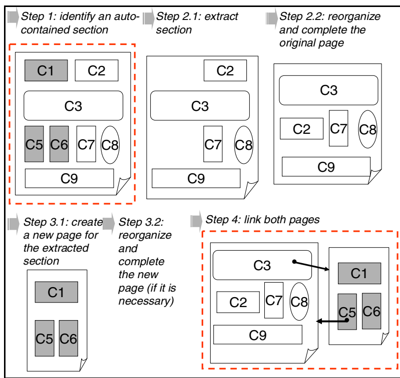

Fig. 2a. “Split Page”

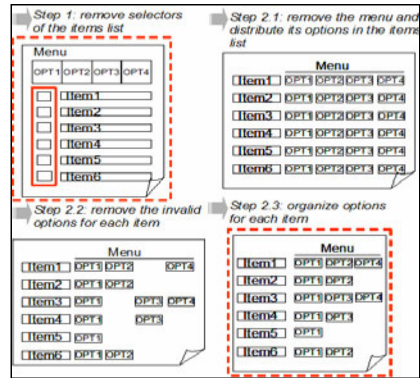

Fig. 2b. “Distribute General Menu”

Figure 2a shows the steps needed to put into operation the “Split Page” presentation refactoring. As shown in the figure, the application of this refactoring involves simplifying an existing page, identifying and extracting self-contained blocks of information and functionality (steps 1 and 2.1), and then, creating one o more pages with the information / functionality extracted from the original page (step 3.1). Both, the original page (step 2.2) as the news pages (step 3.2) must be structured (that is, to organize the information for their appropriate reading and viewing), and can be supplemented or not with other additional information. Finally, the original page and the new pages must be linked together (step 4). Most refactorings allow several alternatives for certain steps in their mechanics. For the sake of conciseness Figure 2a shows the “normal” course of the “Split Page” refactoring. In Section 4 we illustrate the use of this refactoring in a concrete example.

Figure 2b shows the mechanism of the refactoring “Distribute General Menu”, which proposes to remove the general menu affecting a list of elements by adding the menu actions to each element. The selectors of elements (e.g. checkboxes) are also removed in the container as the operations are now locally applied to each element.

In most cases, when solving a usability problem we need to update both navigation and presentation levels. Thus, many navigation refactorings have associated an automatic mechanism for changes propagation, which implies the execution of one or more presentation refactorings. More details can be read at [14]. We next show how we use the ideas behind accessibility refactorings in an agile development process.

4 Our Approach in a Nutshell

Along this section, we will show how we use the catalogue of accessibility refactorings to make the development of UA Web Applications easier. In a coarse grained description of our approach, we can say that it has roughly the same steps that any refactoring-based development process has (e.g. see [12]), namely: (a) capture application requirements, (b) develop the application according to the WACG accessibility guidelines, (c) detect bad smells (in this case UA bad smells), and (d) refactor the application to obtain an application that does not smell that way, i.e. which is more usable, besides being accessible.

Notice that step *b* (application development) may be performed in a model-based way, i.e. creating models and deriving the application, or in a code-based fashion, therefore developing the application by "just" programming. Step *c* (detecting bad smells) may be done "manually", either by inspecting the application, by performing usability tests with users, or by using automated tools. Finally, step *d*, when refactorings are applied, may be manually performed following the corresponding mechanics (See Figure 2), or automatically performed by means of transformations upon the models or the programming modules. A relevant difference with regard to the general process proposed in [12] is that in our approach step *d* is only applied when the application is in a stable step (e.g. a new release is going to be published) and not each time we add a new requirement. Anyway, for each accessibility refactoring we perform a short cycle, to improve the application incrementally.

One important concern that might arise regarding this process is that it might be costly, particularly during evolution. Therefore, we have developed an agile and flexible development process, and a set of associated tools which guarantee that we can handle evolution in a cost-effective way [15]. For the sake of conciseness, we will focus only on the features related to accessibility rather than evolution issues which are outside the scope of the paper. We discuss them as part of our further work.

Concretely, we use WebTDD [3], an agile method that puts much emphasis in the continuous involvement of customers, and comprises short development cycles in which stakeholders agree on the current application state. WebTDD uses specific artefacts to represent navigation and interaction requirements, which we consider to be essential for accessible applications. Similarly to Test-Driven Development (TDD) [16], WebTDD uses tests created before the application is developed to "drive" the development process. These tests are used later to verify that requirements have been corrected fulfilled. Different from "conventional" TDD, we complement unit tests with interaction and navigation tests using tools like Selenium (http://seleniumhq.org/). Figure 3 shows a simplified sketch of the development process. In the first step (1) we "pick" a requirement (e.g. represented with use cases or user stories) and in (2) we agree on the look and feel of the application using mockups. We capture navigation and interaction requirements, and represent them using WebSpec [17], a domain-specific language (DSL) which allows automatic test generation and tracking of requirement changes. At this point we can exercise mockups and simulate the application, either using a browser or a screen reader; therefore we can check accessibility guidelines and have early information on the need to refactor to improve usable accessibility in the step 7. Next (3), we derive the interaction tests from the WebSpec diagrams, and run them (4); it's likely that these tests will fail,

indicating the starting point to begin the development to make tests pass. As said before, step 5 might imply dealing with models (generating code automatically), coding or a combination of both. In step 6, we run the tests again and iterate the process until all tests pass. Once we have the current version of the application ready we repeat the cycle with a new requirement (steps 1 to 6).

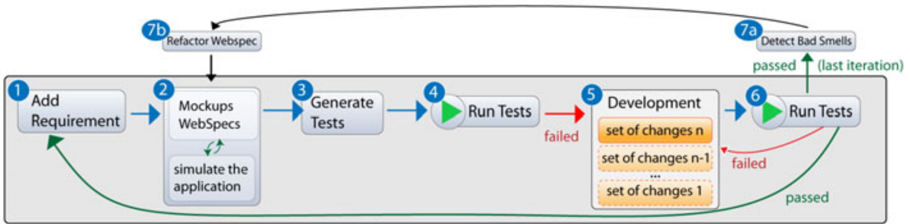

Fig. 3. WebTDD process for UA applications

After we reach a stable state of the application and we want to publish its current version, we look for bad accessibility smells and identify the need for UA refactorings (7.a). Next, we perform short cycles by applying each refactoring to the WebSpec diagrams containing the detected bad smells (7.b). The altered WebSpecs will generate new tests that check the new accessibility/usability features, and propagate the changes to the code (or model) to obtain the new UA application (steps 2 to 6).

In the next subsections we explain some of these aspects in a more detailed way focusing on UA development. To illustrate our approach, we will use the development of a simplified online book store (as the one shown in Figure 1), and when possible, we ignore the activities related with tests since they are outside the scope of the paper.

4.1 Gathering Navigation and Interface Requirements

Navigation and interface requirements are captured early in the development cycle through mockups and WebSpecs (step 2 in Figure 3). User interface Mockups help to establish the look and feel of the applications, along with other broad interaction aspects. They can be elaborated using plain HTML or commercial tools such as Balsamiq (http://www.balsamiq.com). Mockups can be easily adjusted to comply with accessibility guidelines. Figure 1 showed a mockup for our example's homepage.

WebSpecs are simple state machines that represent interactions as states and navigations as transitions, adding the formal power of preconditions and invariants to assert properties in states. An "interaction" represents a point where the user consumes information (expressed with interface widgets), and interacts with the application by using some of its widgets. Some actions (clicking a button, adding some text in a text field, etc) might produce "navigation" from one "interaction" to another and, as a consequence, the user moves through the application's navigation space.

Figure 4 shows a simplified WebSpec diagram that specifies the navigations paths from the BookList interaction and is related with the mockup of Figure 1. In the *BookList* "interaction" the user can authenticate, add books to the cart or to the wish

list and search books. This diagram is the starting point for developing our simplified book store application, as it has key information to specify (at least partial) navigational models (as shown in [3]). Additionally, WebSpecs allow the automatic generation of navigation tests for the piece of functionality it represents, and with the aid of a tool suite, it records requirements changes, to trace and simplify implementation changes. In this sense, every changes made in a WebSpec (even the "initial" WebSpec as a whole) are recorded as "first class" change objects (as shown in Figure 4); these objects are later related with the corresponding model (or implementation) artefacts to improve traceability and automatic change management, using effect managers as explained in [15]. Specifically, each feature of the WebSpec in Figure 4 is traced to the corresponding modelling elements; in this way, when some of these features change, there is a way to automatically (or with minor designer intervention) change the corresponding models or programs. Therefore, step 5 in Figure 3 is viewed as the incremental application of these changes to the current implementation.

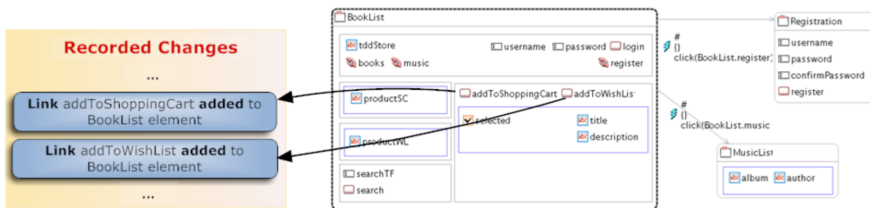

Fig. 4. *BookList* interaction in the *Books Store* WebSpec

4.2 Deriving an Accessible Application

In our approach we do not prescribe any particular development style, though we have experienced with model-driven (specifically, WebML [18]) and code-based (with Seaside- www.seaside.st/) approaches. Once we run the tests for a requirement and noticed that they fail, we build the corresponding models to satisfy such tests and derive the application (steps 3 to 6, in Figure 3). The construction of these models is an incremental task, managing the effects of each recorded change (step 5 in Figure 3). In addition to the changes log, we record the relationship between the WebSpec elements and its counterpart in the model, necessary for the automation of future changes on these elements. This recording is done at this stage, when change effects are managed and the model is built. For example, when the 'add to cart' addition link is managed, a counterpart element is added to the model and the relationship between both elements is recorded. Then, if we need to manage a change to configure any property of this element (e.g. its value), this can be automated since the change management tool knows its representation in the model.

In this stage, accessibility can be addressed using any of the approaches cited in Section 2, for example by incorporating Dante annotations in the corresponding model-driven approach (See [10, 11]). Alternatively, we can "manually" work on the resulting application by improving the HTML pages to make them fully accessible. In both cases, given the nature of the WebTDD approach, the improvement is incremental; in each cycle we produce an accessible version of the application.

In traditional Web application development, accessibility is tackled as a monolithic requirement that must be satisfied by the application which is checked by running accessibility tests such as TAW (http://www.tawdis.net/). What is different in our approach is that we do not try to make the application accessible in one step; instead, we can decide which tests must be run on each development iteration and specific page. Therefore, in the first iteration we may want to make the "BookList" page accessible and satisfying the accessibility test "Page Titled" (Web pages must have titles that describe topic or purpose) and in the second iteration we may want to do the same with page "Best sellers". Our approach follows the very nature of agile development trying to incrementally improve the accessibility of an existing application. Stakeholders' involvement obviously helps in this process. To achieve this goal, we can specify which tests must be run on a specific interaction for each WebSpec diagram. For instance, in the diagram of Figure 4, we can initially run the "Page Titled" test during the first iteration, and the "Headings and Labels**"** (headings and labels describe topic or purpose) test during the second iteration. This approach helps to improve times during development and allows focusing on a specific accessibility requirement, though we can still execute all accessibility tests for every "interaction" like in traditional Web application development if necessary. From an implementation point of view, this "selective" testing is performed using a Javascript version of the WGAC accessibility tests and executing them depending on the tests selected on the WebSpec diagrams.

4.3 Detecting Bad Accessibility Smells

By following the WebTDD cycle (steps 1 to 6 in Figure 3), we will obtain an accessible application, but not necessarily a UA application. For example, if we analyze the page shown in Figure 1 (accessible according the WCAG), we can see that it presents several bad smells contemplated in the UA catalogue that have been outlined in Section 3.1.

First, the page mixes concepts and functions that are not closely related, such as: shopping cart, wish list, information on books, access to other products and user registration. A sighted user quickly disregards the information in which he is not interested (e.g. the registration if he just wants to take a look) and goes quickly to the area that contains what he wants (e.g. the central area where the available books are listed). However, a blind user does not have the ability to look through; when accessing the page using a screen reader which sequentially reads the page content, he will be forced to listen to a lot of information and functionality in what he may be not interested before reaching the desired content. In order to eliminate this bad smell, the refactoring "Split Page" can be applied. Besides, the actions provided to operate with the products listed in the central area of the page (books in this moment) refer to the selected books in the list; this implies that before applying an action in this menu (for example, add a book to the cart), the book or books must be selected by using checkboxes. This task is trivial for a sighted user, but it is considerably more complicated for a user who is accessing through a screen reader, as the reader reads the actions first and then the book list. Even though it is possible to scroll through the links on the page with the use of navigation buttons (provided by most readers), moving back (e.g. to look for the option once you have marked the products), can cause confusion and

be tedious if the list is long. In order to eliminate this bad smell, the refactoring "Distribute General Menu" can be applied.

Therefore, we conclude that we need to apply some refactorings to obtain a better application. This could be done manually on the final application but it might be difficult to check that we didn't break any application behaviour. Next, we show how to make this process safer and compatible with the underlying WebTDD process and at the same time settle the basis to simplify evolution.

4.4 Applying Refactorings to the WebSpec Diagrams

As a solution to safely produce UA Web applications from existing ones, we propose to apply accessibility refactorings to the navigation and interaction requirements specifications (step 7 in Figure 3). Since WebSpec is a DSL formally defined in a metamodel [17], these refactorings are essentially model transformations of WebSpec's concepts. Each transformation comprises a sequence of changes on a WebSpec diagram, which are aimed to eliminate a specific bad smell. Moreover, as shown in [15,17] and explained before, these changes are also recorded in change objects that can be used to semi automatically upgrade the application as we will show in Section 4.5.

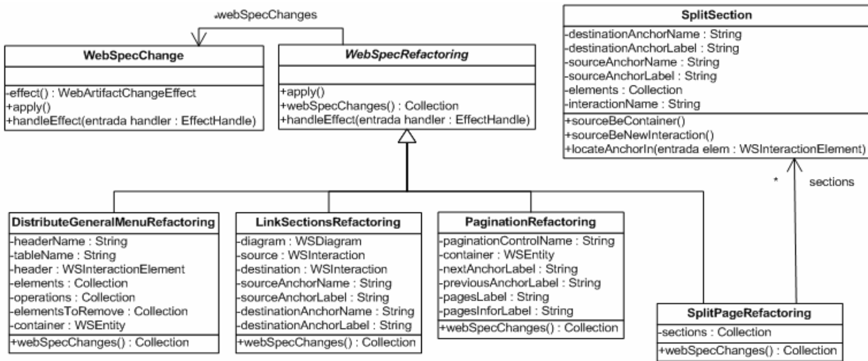

Fig. 5. Refactoring's metamodel

Usability refactorings are also conceptualized in a metamodel, part of which is shown in Figure 5. Refactorings classes provide an extension to the WebSpec metamodel; this allows grouping a set of changes with a coherent meaning. An interesting point to remark is the fact that these refactorings transform WebSpec diagrams instead of models (or code). This has several advantages; for example, with these diagrams and the corresponding new mockups, we can simulate the application. The new mockups could be automatically generated if they are also "imported" from a metamodel like we do in [19]. Also we automatically generate the new navigation tests to assess if the implementation changes were implemented correctly.

As an example, let us consider the application of the "Distribute General Menu" refactoring to the WebSpec of Figure 4. This refactoring takes as input an item container, elements and menu options to be distributed into this container, and elements

to be eliminated for each item. In our example, we configure this refactoring with the “book” element as container, the elements “title” and “description”, the menu options “Add to cart” and “Add to Wish List” and the checkbox to be removed. In Figure 6 we show the result of applying the refactoring, where the “Add to cart” and “Add to Wish List” options are added in each book item (in order to simplify the diagram we only shows the BookList interaction).

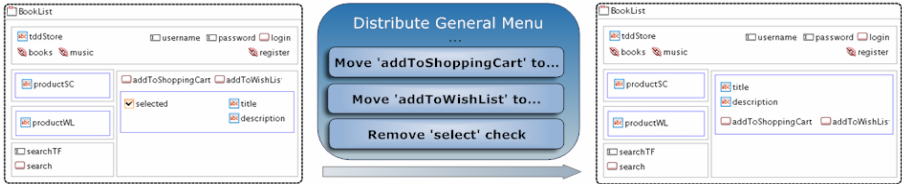

Fig. 6. “Distribute Menu” refactoring example

4.5 Deriving the UA Web Application

Once we applied a refactoring to the needed WebSpec specification, we proceed with the cycle (Figure 3). From now on, we work as we did with “normal” requirements (steps 2 to 6). Once we agreed the new look and feel of the refactored application with the customer (step 2), we generate and run the navigation tests to drive the implementation of these changes (steps 3 and 4). We run these tests; they obviously fail and the process continues with the refactoring effect management (step 5). As we previously explained, refactorings introduce changes in the corresponding WebSpecs, and their explicit representation as first-class objects helps us manage the changes to be applied to the application. As a refactoring is a group of WebSpec changes, we “visit” each of these changes to manage its effects. We start delegating these changes to a Change Management tool, which can automatically (or with some programmer intervention) alter either the models that will in turn generate the new application, or the code in a code-based approach.

In our tool suite we deal with these refactorings in the same way that we manage the changes generated by any new requirement. For example as shown in Figure 7, the “Distribute General Menu” refactoring involves “Move operations” changes; when we manage these changes the “Add to cart” and “Add to Wish List” links are moved into each book item on the application.

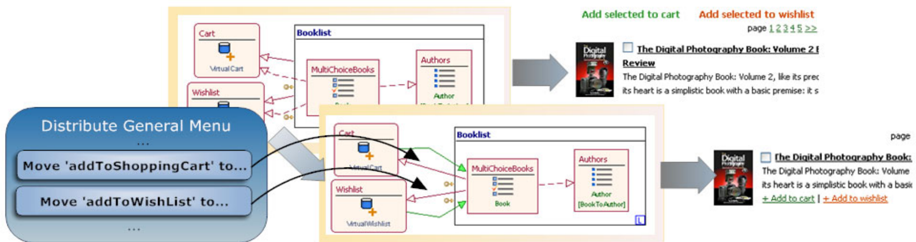

Fig. 7. Handle “Move operations” effects

We use these changes in the WebSpec specification to improve the development stage, with the aim of reducing the cost of their effects on the application, automating these effects in many cases. Additionally, we are able to determine which tests are affected by each change, to trim the set of required tests that must be performed (see the details of this change management process in [15,17]). Finally, a UA requirement is completed when all tests pass (step 6).

In our example, one of the bad smells detected in 4.2 is the way the interaction with the items on the book list is performed: a checklist with general operations to apply on the selected books. In this situation, the refactoring "Distribute General Menu" can be applied in order to improve the usability.

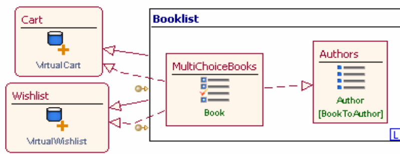

Fig. 8a. General Menu with checklists

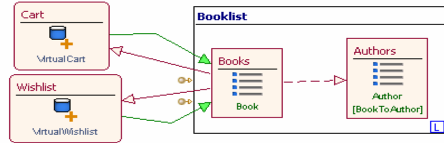

Fig. 8b. Distributed Menu

Figure 8a shows a WebML diagram for the page that lists all books and lets the user add books to a Shopping Cart or a Whish list. Since the book list is presented as a checkbox set (using a specific WebML unit called "Multi Choice Index Unit"), the user has the ability to check different books and select an action to perform on the selected group, as seen in the units "Cart" and "Whishlist". The application of this refactoring generates automatically the diagram in Figure 8b, where the book list becomes a simple list (replacing the "Multi Choice Index Unit" unit with a plain "Index Unit"); the actions "Cart" and "Whishlist" are now directly linked from the list, and therefore every item on the Index Unit called "Books" gets individual links to each action. From this new navigational model and the corresponding interface template (derived from the mockup), we are able to derive a UA version of the home page shown in Figure 1. Figure 9a shows the result of the process.

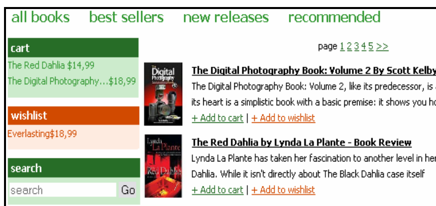

Fig. 9a. Distributed Menu in book list

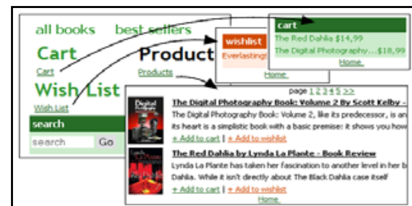

Fig. 9b. A new, usable and accessible home

Another bad smell detected is the mixed up contents on the bookstore's homepage. To overcome this problem, the "Split Page" refactoring is applied. For the sake of conciseness we only show the final result of applying the refactoring in the final application. Figure 9b shows the result of the new iteration, where the initial page has been cleaned, extracting in three new pages the information and functionality needed

to: list products (BookList page), manage the wish list (WishList page) and manage the shopping cart (ShoppingCart page). After finishing this process we end with two Web applications: the "normal" one and the UA application. From now on, evolution can be tackled in two different ways: by treating the two applications separately or by working on the WebSpecs of the original one, following the WebTDD cycle and then re-applying the "old" refactorings to the modified specifications when needed.

5 Concluding Remarks and Further Work

In this paper we faced the problem of improving the usability of accessible Web applications. We consider that an application that has been developed to be usable for regular users is generally not usable (even if accessible) for handicapped users, and vice versa. In order to provide a solution for such important problem, we have presented an approach supported by there pillars: a) a catalogue of refactoring specialized in UA problems for blind and visually impaired users; b) a test-driven development process, which uses mockups and Webspecs to simulate the application and to generate the set of tests to assure that all the requirements are satisfied (included the accessibility requirements) and c) a metamodel capable of internally representing the elements of the application and the changes upon these elements (included changes resulting from refactoring) in the same way; this makes easier the evolution of both, the normal application and the UA application. As further work, we are considering how to define catalogues of refactorings for other types of disabilities, for example: hearing impairments, physical disabilities, speech disabilities and cognitive and neurological disabilities. In turn, we are working in order to specialize each catalogue according to the particular type of disability. In addition, the catalogues may be also specialized according to the type of web application: communication applications (facebook, twitter, etc.), electronic commerce (amazon, e-bay, etc.), e-learning, etc. On the other hand, we are considering how to gather and represent usable and accessibility requirements. In this way, the UA refactorings could be applied at any iteration of the development cycle. For this to be feasible (and not too costly), we need to improve the change effect management, to automate the propagation of most changes from the original application to the UA one. Finally, we are improving out tool support to simplify evolution when new requirements affect those pages which were refactored during the usability improvement process. In this sense we need to have a smart composition strategy to be able to compose the "new" change objects with those which appeared in the refactoring stage.

Acknowledgements. This research is supported by the Spanish MCYT R+D project TIN2008-06596-C02-02 and by the Andalusian Government R+D project P08-TIC-03717. It has been also funded by Argentinian Mincyt Project PICT 2187.

References

1. Barnicle, K.: Usability Testing with Screen Reading Technology in a Windows Environment. In: Conf. on Universal Usability, pp. 102–109. ACM Press, New York (2000)
2. W3C.: Web Content Accessibility Guidelines 2.0 (December 2008), http://www.w3.org/TR/WCAG20/

3. Robles Luna, E., Grigera, J., Rossi, G.: Bridging Test and Model-Driven Approaches in Web Engineering. In: Gaedke, M., Grossniklaus, M. (eds.) Web Engineering. LNCS, vol. 5648, pp. 136–150. Springer, Heidelberg (2009)
4. Zajicek, M., Venetsanopoulos, I., Morrissey, W.: Web Access for Visually Impaired People using Active Accessibility. In: Int. Ergonomics Association 2000/HFES, San Diego, pp. 445–448 (2000)
5. Ivory, M., Mankoff, J., Le, A.: Using Automated Tools to Improve Web Site Usage by Users with Diverse Abilities. Journal IT & Society 1, 195–236 (2003)
6. IBM: Watchfire's Bobby, `http://www.watchfire.com`
7. Harper, S., Goble, C., Steven, R., Yesilada, Y.: Middleware to Expand Context and Preview in Hypertext. In: ASSETS 2004, pp. 63–70. ACM Press, New York (2004)
8. Fernandes, A., Carvalho, A., Almeida, J., Simoes, A.: Transcoding for Web Accessibility for the Blind: Semantics from Structure. In: ELPUB 2006 Conference on Electronic Publishing, Bansko, pp. 123–133 (2006)
9. Bohman, P.R., Anderson, S.: An Accessible Method of Hiding Html Content. In: The International Cross-Disciplinary Workshop on Web Accessibility (W4A), pp. 39–43. ACM Press, New York (2004)
10. Yesilada, Y., Stevens, R., Harper, S., Goble, C.: Evaluating DANTE: Semantic Transcoding for Visually Disabled Users. ACM Transactions on Computer-Human Interaction (TOCHI) 14, 14-es (2007)
11. Plessers, P., Casteleyn, S., Yesilada, Y., De Troyer, O., Stevens, R., Harper, S., Goble, C.: Accessibility: A Web Engineering Approach. In: 14th International World Wide Web Conference (WWW 2005), pp. 353–362. ACM, New York (2005)
12. Fowler, M., Beck, K.: Refactoring: Improving the Design of Existing Code. Addison-Wesley Professional, Reading (1999)
13. Garrido, A., Rossi, G., Distante, D.: Model Refactoring in Web Applications. In: 9th IEEE Int. Workshop on Web Site Evolution, pp. 89–96. IEEE CS Press, Washington (2007)
14. Medina-Medina, N., Rossi, G., Garrido, A., Grigera, J.: Refactoring for Accessibility in Web Applications. In: Proceedings of the XI Congreso Internacional de Interacción Persona-Ordenador (INTERACCIÓN 2010), Valencia, Spain, pp. 427–430 (2010)
15. Burella, J., Rossi, G., Robles Luna, E., Grigera, J.: Dealing with Navigation and Interaction Requirement Changes in a TDD-Based Web Engineering Approach. In: Sillitti, A. (ed.) XP 2010. LNCS, vol. 48, pp. 220–225. Springer, Heidelberg (2010)
16. Beck, K.: Test-driven development: by example. Addison-Wesley, Boston (2003)
17. Robles Luna, E., Garrigós, I., Grigera, J., Winckler, M.: Capture and Evolution of Web requirements using WebSpec. In: Benatallah, B., Casati, F., Kappel, G., Rossi, G. (eds.) ICWE 2010. LNCS, vol. 6189, pp. 173–188. Springer, Heidelberg (2010)
18. Acerbis, R., Bongio, A., Brambilla, M., Butti, S., Ceri, S., Fraternali, P.: Web Applications Design and Development with WebML and WebRatio 5.0. In: Objects, Components, Models and Patterns. LNCS, vol. 11, pp. 392–411. Springer, Heidelberg (2008)
19. Rivero, J., Rossi, G., Grigera, J., Burella, J., Robles Luna, E., Gordillo, S.: From Mockups to User Interface Models: An extensible Model Driven Approach. To be published in Proceedings of the 6th Workshop on MDWE. LNCS. Springer, Heidelberg (2010)