Designing software architectures to facilitate accessible Web applications

D. Hoffman
E. Grivel
L. Battle

The Web application is increasingly a platform of choice for complex business software and online services. However, it remains a challenge to ensure that the Web application is easy, efficient, and effective for people with disabilities. Accessibility requires that users with disabilities, including those who are blind, have low vision, or have mobility impairments, are able to use the applications effectively and with a reasonable amount of effort. Although there has been important progress in recent years in describing the relationship between architecture and usability, the topic of architectural support for accessibility has not been adequately addressed. Based on our experience in designing Web applications for the United States Social Security Administration, we have begun to identify guidelines for architectures that support accessibility. This paper describes common accessibility problems encountered in Web applications and explains how architecture can help address these problems through reusable accessible objects, supplementing information in links, buttons, and labels, assisting in access to Web page visual information, handling errors, and providing time-out notification and recovery. It also discusses the critical role of architecture in supporting the best way of meeting the needs of diverse user groups: multiple dynamic views of the user interface.

INTRODUCTION

As consultants to the United States Social Security Administration, a government agency that is committed to accessibility, we have maintained a focus on improving the user experience for all users of our Web applications. These applications include sophisticated data-entry applications (similar to the Web version of Turbo Tax**), claims-processing systems (similar to those used inside insurance companies), workload-management systems, correspondence-routing systems, and call-center systems. Such applications require complex navigation, extensive data entry, conditional relationships among different data elements, and multiple interrelated user tasks. The baseline software architecture for the front end of these Web applications is DHTML (dynamic HTML), which is dynamically produced using Java** technologies. DHTML may include HTML (Hypertext Markup Language), CSS (Cascading Style Sheets), and JavaScript**.

There are several common challenges encountered when working with project teams to improve accessibility. One is a lack of comprehensive guidelines that apply to interactive Web applications. While there are many published sources of information on Web site accessibility[1,2] and the needs of users with disabilities,[3] existing Web accessibility guidelines typically focus on the design of static informational Web sites or basic Web forms[4–9] and do not address design issues that typically arise in complex software applications. They generally discuss each individual accessibility issue in a vacuum, without addressing external design constraints or the interrelation of issues. Moreover, the introduction of accessibility into an application that is already fully developed can involve significant redesign and recoding, which may be considered outside the project's scope and budget. Obtaining input from accessibility specialists before coding starts (during user-interface design and specification) reduces the risk of rework but does not eliminate the need for significant manual testing and recoding. For this reason, it is important to look across suites of related applications and identify ways of supporting accessibility through the use of reusable components that consistently implement common business rules, design requirements, and other site-wide functionalities, including accessibility.

Two of the authors recently completed an analysis of documented accessibility violations and recommendations; in the analysis more than 1,300 accessibility issues, which were identified during evaluations of 80 software applications over a three-year period, were compiled and categorized. This analysis led to the creation of a list of "Top 20" accessibility issues,[10] which we have used in developing user-interface standards and in reviewing the architectural implications of accessibility.

Although we considered a wide spectrum of disability types, we discovered a stronger emphasis on vision-related disabilities than on other disabilities. There are probably several reasons for this emphasis, including the fact that vision-related challenges to access are the most numerous and significant, the access solutions are relatively feasible, and the fact that people with vision-related disabilities are some of the most active participants in the job market and some of the strongest advocates for their causes. This paper does not discuss the full list of recurring accessibility issues; instead, it focuses on those issues that can be addressed within an intermediate architectural layer of reusable software components. We argue that addressing accessibility issues within such an architecture can significantly enhance accessibility, and failing to address them within such an architecture can significantly limit accessibility.

Accessibility and usability

Software is *accessible* when the user interface is designed to meet the special needs of people with disabilities, allowing them to use software in a manner that is similar to the way that people without disabilities use it. Disabilities may include a limited ability to see, hear, or move (including using a keyboard or mouse), or to process certain types of information easily or at all. Software accessibility is often accomplished by ensuring that necessary information about user-interface elements is available to various assistive technologies, such as screen readers for users who are blind, magnification software for users who have low vision, or speech input software for users with mobility impairments. Although Section 508,[11] the World Wide Web Consortium Web Accessibility Initiative[12] (W3C** WAI), and even the ADA[13] (Americans with Disabilities Act) seek to recommend or mandate various accessibility standards, our primary focus is on the shared goal of all such standards: ensuring that users with disabilities can use software effectively and with a reasonable amount of effort.

Accessibility is closely related to *usability*, the art and science of ensuring that software is efficient and effective to use and that its use is satisfying for users. Usability practitioners sometimes consider accessibility to be a subcategory of usability, despite the fact that, in practice, accessibility is usually handled separately from usability. Although the goals of accessibility and usability are similar in many ways, the specific design enhancements needed to support usability for a general audience and accessibility for users with disabilities may differ significantly and may pose conflicting goals. In addition, different groups of users with disabilities have different needs. Our experience in balancing the needs of these different user groups has led us to conclude that in some situations, one solution for all users is not desirable.[14] The idea of multiple views tailored to the needs of different user groups is explored further in this paper.

Software architecture

There are numerous definitions of "software architecture" in the technical literature.[15] Essentially, software architecture describes the organizational structure of a software system including components, interactions, and constraints. Architectural interactions are abstractions for how components interact in a system. An architecture includes the constraints on component selection and the rationale for choosing a specific component in a given situation.[16] For our purposes, software architecture describes the function of components of a system, including their interaction with each other.

There are many different aspects to the architecture of a system, including the computer hardware, software, and network. Even though this paper discusses software that is developed using a DHTML front-end architecture, our focus is on an intermediate application layer. This layer is located between the back-end processes or databases (or both) and the presentation-layer technologies (such as HTML, CSS, server-side scripting [e.g., JavaServer Pages**] and client scripting [e.g., JavaScript**]). In addition to the business-logic code, this intermediate application layer includes typically proprietary common reusable software components. Just as most software is not designed to run directly on top of an operating system, complex systems are commonly built in development environments that include such an additional layer. This layer of software components includes tools, functions, and restrictions that form the core design and basic architecture of the site or the applications on a site. It consistently implements common business rules, design requirements, and other site-wide functionalities. When such an intermediate architectural layer is used, it can serve either to limit accessibility if it is not designed with accessibility in mind or to enhance accessibility if it is designed with accessibility in mind.

In recent years, several authors have begun to describe the relationship between usability and architecture. Bosch described a direct relationship between architectural decisions and the ability to meet quality requirements.[17] Juristo, Moreno, and Sanchez researched the architectural implications of usability issues and pointed out the danger of assuming that usability only affects the presentation component of software systems.[18] John and Bass[19] have done extensive work on this subject and have illustrated how, despite the best efforts of architects to create modularized software that facilitates changes, it becomes difficult to meet user-experience requirements after architectures are already defined. Reference 19 describes a scene in which usability issues are presented, and one of the developers exclaims, "Oh, no, we can't change THAT!" The problem is that the requested modification reaches too far into the architecture of the system to allow economically viable and timely changes to be made. Even when the functionality is correct and the user interface is separated from that functionality, some architecture decisions may unknowingly limit the ability to implement usability requirements. Bass, John, and Kates have published a collection of architecture patterns intended to help architects anticipate and accommodate usability.[20]

Although these works do not directly address accessibility issues, the relationship between usability and architecture is similar to the relationship between accessibility and architecture. In our experience, a common reason given by development teams for declining to implement accessibility features or enhancements in complex Web applications is a lack of architectural support and the cost in time and money involved in implementing enhancements that require architectural modification. Conversely, after an accessible solution is built into the architecture, it is much easier to consistently extend that solution across multiple applications. In light of this, the goal of this paper is to extend the existing work on the relationship between usability and architecture by providing a set of architectural recommendations to improve the user experience for people with disabilities.

ACCESSIBILITY ISSUES AND SOFTWARE ARCHITECTURE SOLUTIONS

This section provides specific information about addressing accessibility within an architecture of reusable software components.

Using libraries of reusable objects

A common challenge in developing accessible applications is the significant knowledge gap that exists between software developers and accessibility specialists. Most developers do not have experience or training in coding for accessibility, and most accessibility specialists have limited programming training or experience. These specialists may not be able to provide sample code that developers can use to achieve the desired results. This leads to inconsistent results; for example, different developers have varied levels of accessibility awareness, and even when they do implement accessibility features, they may use different approaches, or code the same approach differently. In addition, even when accessibility features are implemented properly, they must still be manually applied to each individual page element throughout the entire application.

Complex Web applications or online services increasingly use software-generated HTML (utilizing technologies such as Java-programming-language custom tags), rather than simple static HTML. In fact, the use of software-generated HTML is partially responsible for the very architectural complexities that distinguish complex Web applications from static Web sites. The use of software-generated HTML enables the use of common reusable components.

The value of designing with reusable components is apparent in projects remediating the accessibility of Web applications. Projects not using reusable components required manual testing, recoding, and retesting of dozens of controls across hundreds of pages. Projects using reusable components required testing and recoding only for each type of reusable component. The changes then automatically propagated across the application.

Common accessible solutions can be incorporated into reusable software components and data repositories across applications or even suites of applications. Reusable components define the structure and attributes of a particular type of page element. These components can use data repositories (i.e., flat files, such as Java property files) that contain attribute values, such as field labels and titles. Reusable components and data repositories exist completely in the background and remain invisible to the user interface, but can be essential to the consistent and efficient implementation of accessibility features.

The following are some examples of reusable objects that we have designed to promote accessibility:

1. *Fields with associated field labels*. Using a reusable object ensures that field labels are read with the appropriate fields in a consistent way throughout the application.
2. *Set of radio buttons with captions.* This object is a set of radio buttons with captions. Using a reusable object ensures that the radio button caption is read with each set of radio buttons in a consistent way throughout the application.
3. "*Continue*" *and* "*previous*" *page buttons with associated hotkeys*. These buttons and "hotkeys" (keyboard shortcuts, also known as accelerator keys) allow keyboard users to quickly navigate to the next or previous page without having to tab through all the controls on the current page.

The code in reusable objects can enforce certain coding standards, such as a requirement to provide alternate or supplemental text for an image, field label, link, or push button. For example, Java-based custom tags can be used to encapsulate the logic to generate consistent, accessible HTML. Although they cannot ensure that the alternate or supplemental text is accurate or appropriate, they can ensure that the text is not omitted or forgotten completely. This strategy ensures that once each type of control has been developed, tested, and refined, it will always be the same. Reliance on the knowledge of individual developers or accessibility specialists in order to code for accessibility is thus reduced.

Reusable objects also provide the ability to implement accessibility features by changing a relatively small number of reusable components, rather than individually changing many individual controls across multiple pages throughout an application or families of applications. The use of reusable components provides a level of consistency that is difficult to achieve even after significant testing and tweaking of individually coded components. The difference between remediating accessibility issues in an application that uses reusable components and one that does not is very significant.

The software architecture should contain reusable components that encapsulate logic which generates consistent, accessible HTML and should reference these components from each place where they are needed in an application. For example, rather than coding each set of radio buttons independently, radio buttons can be standardized into a single construct (or a limited number of different logical constructs, if appropriate). This construct should include code to associate both the radio buttons with their individual labels and the caption with the entire set of radio buttons.

Field-specific data attributes are commonly specified when a reusable control is called. In some situations, such as when the same type of control appears in multiple places, there is a benefit from using *data repositories* to store the data attributes related to each control. These data repositories can then be used to create the controls by populating the attributes of the reusable components. The data repositories can include or even require the storage of various types of supplemental text, as discussed next.

It is important to note that as consistency improves through the inclusion of more functionality in the architecture, flexibility is likely to decline. This is not necessarily bad, especially if the application is accurately designed in logical components that match business needs. However, because merely using reusable components does not ensure that anything is implemented correctly, it can also cause accessibility deficiencies to be implemented consistently. Likewise, it can be more difficult to correct certain accessibility deficiencies that are incorrectly implemented after they have become part of the architecture of reusable software components for a suite of Web applications.

Supplementing information

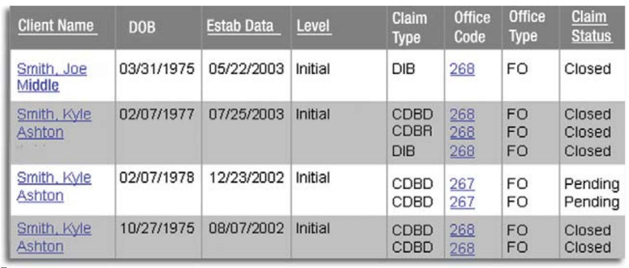
One of the most common accessibility issues involves information that is either not available or not as readily available to users with disabilities as it is to other users. For example, the purpose of some links, push buttons, and field labels on a Web page may be unclear to screen-reader users without the surrounding context. In addition, users may not be aware of the existence of error messages, help, or tips related to a field, or the fact that a field is required, if that information is not included in the field label. Visual users can associate contextual information by simply scanning with their eyes without ever removing their primary focus from the form fields, but screen-reader users must choose between a textual view that enables the reading of all content in order and a field manipulation view that enables the manipulation of all data-entry fields. Therefore, supplemental information can be essential for providing comparable access (i.e., access for people with disabilities which is comparable to that of nondisabled users) to textual information when screen-reader users are performing data entry.

Screen readers do a very good job of handling semantically coded informational pages. They also effectively handle properly coded form fields. Nonetheless, screen-reader users face a unique challenge when navigating through pages of mixed content (form fields and other controls interspersed with informational screen text), because screen readers must distinguish between manipulation of HTML form controls and the reading of HTML textual information. Even though all of the contextual information is likely to be available somewhere on the page, users with disabilities may face the extra burden of trying to locate and associate the contextual information while engaged in data-entry activities (without knowing whether the information is even available).

Although this use of mixed content does not affect every link, push button, or field label, there are often situations when some of the information needed to understand the purpose of data entry fields or selection mechanisms is conveyed to the user through context, that is, a combination of the surrounding text, page layout, and proximity. For example, when links are presented within a paragraph of text or within a table, the surrounding information often plays a role in communicating their purpose. Much of this contextual information is not easily available to screen-reader users, who may encounter the links or push buttons in isolation. For example, a user who is tabbing from control to control, as is typical when completing a form, may skip over essential screen text without even realizing that the information is available. Similarly, a user who accesses links in a special list of links may not have easy access to surrounding text information. Low-vision users may also have difficulties if the contextual clues are too far away from the link, push button, or field label so that they cannot seen at the same time when the screen is magnified.

Links

When links are used in tables, their meaning is conveyed in part by their row and column location



Client Name	DOB	Estab Data	Level	Claim Type	Office Code	Office Type	Claim Status
Smith, Joe Middle	03/31/1975	05/22/2003	Initial	DIB	268	FO	Closed
Smith, Kyle Ashton	02/07/1977	07/25/2003	Initial	CDBD CDBR DIB	268 268 268	FO FO FO	Closed Closed Closed
Smith, Kyle Ashton	02/07/1978	12/23/2002	Initial	CDBD CDBD	267 267	FO FO	Pending Pending
Smith, Kyle Ashton	10/27/1975	08/07/2002	Initial	CDBD CDBD	268 268	FO FO	Closed Closed

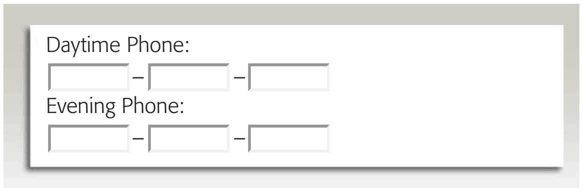
Figure 1
Links used in tables

(see, for example, the client name and office code links in *Figure 1*). Links are used in many different ways. When clicking on a link serves to activate a folder tab or menu system, the link's purpose may be conveyed through its visual appearance; when a link is used to provide pop-up definitions or examples, its purpose may be conveyed through proximity to the item being defined; when a link is used to re-sort a list of items, its purpose may be conveyed by means of a graphic or simply the appearance of a link as a table column heading. All of these situations can potentially create accessibility challenges because the meaning of the link is not clear out of context. Unfortunately, in HTML, all links are simply links, despite the fact that links can play extremely varied roles.

Field labels

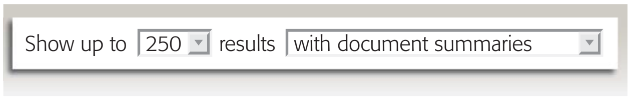
When related fields are grouped, they are often labeled in a way which assumes that the user has some knowledge of the grouping. This makes sense to most users and makes labels more concise, but again, it can pose a challenge for users who are blind and may not have access to the context. There are several common variations on this theme:

1. Fields may share part of their label because they are logically related to one another. For example, a "name" label may be subdivided into separate fields for title, first, middle, last, and suffix; a telephone number may be subdivided into three separate fields, as seen in *Figure 2*.



Daytime Phone:
[ ]–[ ]–[ ]
Evening Phone:
[ ]–[ ]–[ ]

Figure 2
Fields that share part of a label



Show up to [250 ▾] results [with document summaries ▾]

Figure 3
Fields in sentence format

2. Fields may be arranged in a sentence format that makes sense when the whole sentence is seen together, but not when a part of it is seen in isolation. For example, the first drop-down list in *Figure 3* might be read by a screen reader as "Show up to select menu with x items" for the first drop-down list, which technically does provide a label, but does not provide enough information for the user to understand and make a selection (the obvious question, "Show what?" cannot be answered without the label for the second drop-down list: results). For the second drop-down list, a screen reader might say "results select menu with y items," which technically does provide a label, but one that is not easy to comprehend. Neither field has a distinct meaningful label without the context of the entire sentence. A better approach is to use a hidden label (for example, a title attribute) identifying the first field as "number of results to show" and the second field as "display preference."

3. Fields may lack individual labels. For example, month, day, and year fields, or city, state, and zip-code fields often do not have separate labels because their meaning is clear when they are seen together. Another increasingly common situation where fields may lack individual labels is when they appear in tables as seen in *Figure 4*.

In all these situations field labels may need to be coded differently to ensure that screen readers read the correct information.

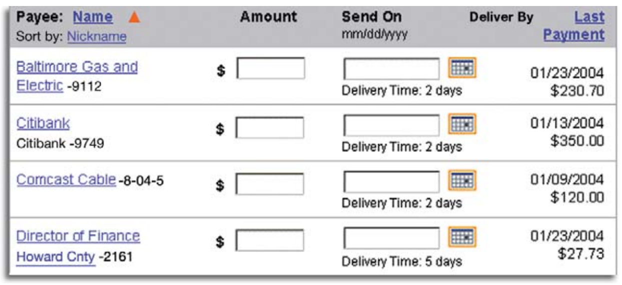
Field-level instructions and messages

Additional information is sometimes provided at the field level, as seen in *Figure 5*. This information may include instructions or tips for answering the question and an indication of whether the field is "required" or "optional." Error messages are also provided when a required field is left blank or when an inappropriate value is entered. See the section "Error handling" for a discussion of alternative approaches. Ensuring that all users have comparable access to such directions and cues again presents a challenge. Users who are blind may not be aware of the existence of an error message or an instruction associated with a particular field if it is not coded as part of the field label. Users with low vision may not be aware of the additional information if it is placed too far from the field to be seen at the same time with a magnified view, as is the case in Figure 5.

In order to use supplemental information as an accessibility solution, labels must be explicitly associated with the appropriate fields so that they can be read reliably with screen readers. Although proper and consistent label placement can usually enable screen readers to find labels, only explicit association ensures accurate results. If it is not possible or desirable to include sufficient information in the element (i.e., a visible link, push button or field label) so that it can be understood out of context, it is necessary to supplement the label. The supplementary information must be read by the screen reader when the reader focuses on the control.

Our recommended approach to implementing such a solution is by using a "title" attribute in the link, push button, or field. The title value should include both the visual text and the supplementary information because, when screen readers are set to read title attributes, they typically read them instead of, rather than in addition to, the visual text. There are times when it is necessary to supplement the text presented on the screen in more than one way, such as by including supplementary text before and after the screen text.

All relevant information should be included in field title attributes, including error messages, help, tips, and an indication if the field is required, although it is important to keep this as brief as possible. If the help or tip information is too long to be read with the field, it is helpful to include a brief indicator, such as "help follows," inside the title attribute. Placing a "tab stop" on the remaining help or tip allows the focus to land there when the user is tabbing through the form (which can be achieved for the Internet Explorer** browser through the use of a "tabindex" attribute) and ensures that the tab stop immediately follows the field in the tab order.



Payee: Name ▲ Sort by: Nickname	Amount	Send On mm/dd/yyyy	Deliver By	Last Payment
Baltimore Gas and Electric -9112	$	Delivery Time: 2 days		01/23/2004 $230.70
Citibank Citibank -9749	$	Delivery Time: 2 days		01/13/2004 $350.00
Comcast Cable -8-04-5	$	Delivery Time: 2 days		01/09/2004 $120.00
Director of Finance Howard Cnty -2161	$	Delivery Time: 5 days		01/23/2004 $27.73

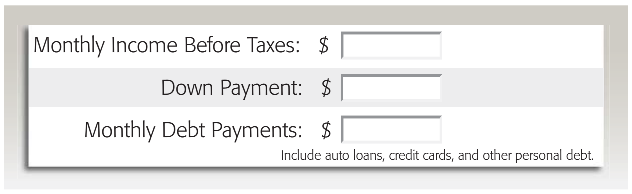
Figure 4
Fields without individual labels

An alternative approach to supplementing screen field-label text is to include the supplementary information within the label tags, hiding it from view but enabling screen readers to recognize and speak the information. The following CSS code visually hides information from the display and, at the same time, allows screen readers to access the information:

```
{position:absolute; left:0px top:-100px;
width:1px; height:1px; overflow:hidden;}
```

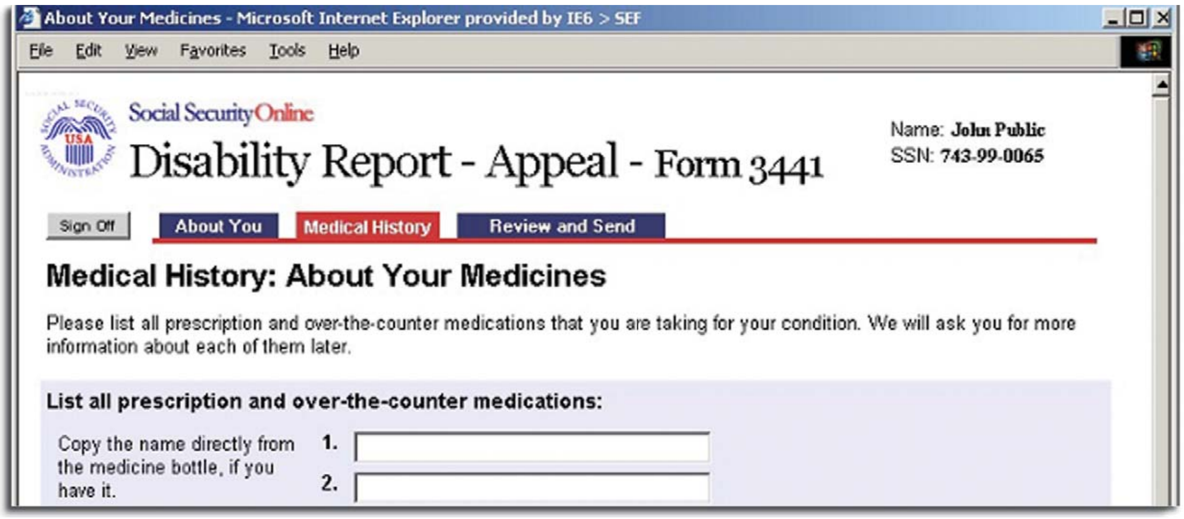
Unfortunately, neither of these solutions really addresses the issue for users with low vision. Generally, it is only possible to mitigate the issue by using a "multiple view" solution, as discussed later.

The following architectural recommendations support the supplemental-information accessibility solutions we have described. The architecture should require that every field have a label. Field labels should be explicitly associated with the appropriate fields by using label tags. This ensures that they will



Monthly Income Before Taxes: $
Down Payment: $
Monthly Debt Payments: $
Include auto loans, credit cards, and other personal debt.

Figure 5
Field-level instruction


About Your Medicines - Microsoft Internet Explorer provided by IE6 > SEF

File Edit View Favorites Tools Help

Social Security Online

Disability Report - Appeal - Form 3441

Name: John Public
SSN: 743-99-0065

Sign Off | About You | Medical History | Review and Send

Medical History: About Your Medicines

Please list all prescription and over-the-counter medications that you are taking for your condition. We will ask you for more information about each of them later.

List all prescription and over-the-counter medications:

Copy the name directly from the medicine bottle, if you have it.
1.
2.

Figure 6
Signposting conveyed through visual elements

be read reliably by screen readers. If a visible label is not provided, an alternative label must be provided instead. The text for supplementary title attributes, hidden text, or field-label tags should be specified in data repositories that define all such objects in the application. This ensures that there is a placeholder in the architecture to accommodate supplementary text as needed for all links, push buttons, field labels, and images.

Reusable components should contain placeholders for storing any information that can be associated with a control (error messages, help, or tips, etc.) within the HTML. The text for title attributes, hidden text, or field-label tags should be automatically generated by reusable components, using the information in the data repositories. In this way, the architecture enforces the existence of all types of text that can be associated with a control.

At the architecture level, each element is marked with a value of "required," "optional," or "conditionally required." This value can be used to generate the information in the label that indicates the required fields. This can be represented as an asterisk for the visible label, and it can be represented as the word "Required" in the hidden text.

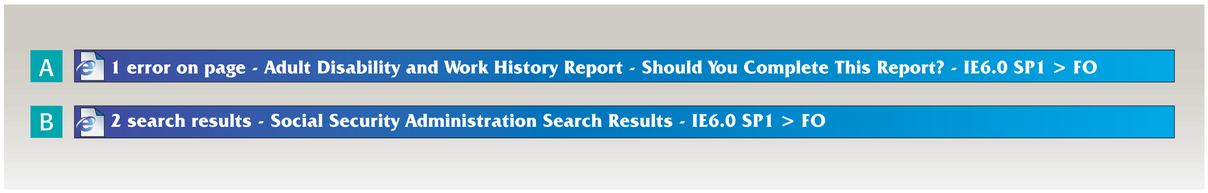
Different types of links should be recognized as specific entities by the architecture. For each type of link, certain attributes should be required (such as link 'type') or optional to provide the necessary context information for the link.

For instance, a link may have the type of "dynamic element." When this link type is defined, attributes may be provided to augment the information available when the link is rendered. The link can be rendered automatically depending on the chosen view and using the available HTML syntax. The advantage of this type of generic architectural solution over manually coding the title attributes, hidden text, or field labels is that it encourages consistency, ensures inclusion, and requires less manual effort.

Providing access to on-screen signposting

Signposting refers to Web page visual information that communicates the title of the application and page, any essential system messages, and any feedback or status messages. Signposting is essential for helping users know where they are in an application. However, users with disabilities may not have comparable access to the signposting cues that show current location and status information.

Visual users have the ability to quickly scan the display text and graphics for on-screen signposting. They can then go directly to the task at hand. In contrast, screen-reader access is linear—the users cannot perform a visual scan. They may need to


A

1 error on page - Adult Disability and Work History Report - Should You Complete This Report? - IE6.0 SP1 > FO

B

2 search results - Social Security Administration Search Results - IE6.0 SP1 > FO

Figure 7
Examples of effective title-bar signposting: (A) title bar showing error information; (B) title bar showing search results

actually listen to a significant amount of potentially irrelevant text merely to determine their location within the business task or to determine the system status.[19]

Figure 6 illustrates how much information about the current location—including the name of the application and Web site, the page, and the section—is available at a glance. Status information, which includes the existence and number of errors, the existence (or the lack) of search results, and specific record or user identifiers, is also typically visible at a glance.

While modern screen readers provide features that attempt to help, these features are typically limited to page structure information, such as the focus location in terms of percentage of the page or the control number on the page.

The browser title bar of each page should briefly provide a summary of the user's location and any special status. The title bar should include the name of the application, followed by the application section, followed by the page, followed by any specific user or record identifiers, as appropriate. Special status information such as the existence of errors, the existence of search results, or the lack of search results should be concisely inserted at the beginning of the title bar when applicable. Other status information, such as specific record or user identifiers, should be inserted at the end.

This solution does not replace the display of the same information on other parts of the page. The page title, the currently selected tab, the record identifiers, and any feedback or error messages are still displayed on the page as appropriate for the visual layout. The advantage of including them in the title bar is that they are read by the screen reader as soon as the page loads, making it easy for the user to obtain the title bar information on demand by using an assistive technology keystroke. This can provide a significant benefit for users switching back and forth between multiple application windows.

Figure 7 shows examples of effective title bar signposting. In Figure 7A the application name, page name, and error notification are displayed in the title bar. In Figure 7B, the page name and notification of search results are displayed.

To support this accessibility solution, the HTML page title tags can be used to define the application name and section and the individual page, as appropriate. Standard functions can be used to create the content of the page title tag by using not only the actual page title but also system-status information, such as the application name, the current section, the existence and number of errors, or the existence and number of search results.

When frames are used and the main content resides in a child frame (Frame 3 in *Figure 8*), the frameset (i.e., a set of frames in a Web page) title bar does not display the title of the child frame. In this case,

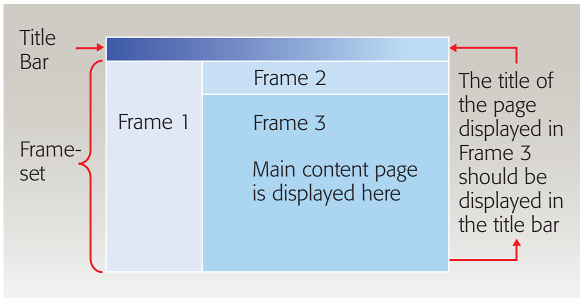

Title Bar
Frame 2
Frame 1
Frame 3
Main content page is displayed here
Frame-set
The title of the page displayed in Frame 3 should be displayed in the title bar

Figure 8
Display of current page name in the title bar

JavaScript should be used to display the title of the current child frame in the title bar of the frameset.

Time-out functionality

Time-out functionality addresses the interval of time for which an Internet connection may remain open without any transmission of data. When a time-out function is used within an Internet application, the user must receive sufficient notification (at the beginning of the application and when the time-out is about to occur), and the user must have the ability to indicate that more time is required, as mandated by Section 508.[21] It is also important for the user to be given the ability to recover after a time-out occurs.

For security reasons, many Internet applications employ a time-out function. This is intended to limit the risks that may occur if an Internet application is left running unattended on a computer where someone other than the appropriate user may have access to it.

Time-out functionality creates a need for three related usability features: user notification, additional-time request, and user recovery. However, time-out functionality is also an accessibility issue because users with disabilities may work more slowly with an application and thus may be more likely to be affected by an application timing out.[22]

It is important to notify users in advance that an application may time out. This notification should appear at the start of each application and can be included as part of the instructions for completing that application. The notice must include the amount of time that is allowed on a screen before a session will time out, instructions on how to request an extension for more time, the number of extensions that will be granted, the consequences that result from a session time-out, and a notification that client-side scripting is required for this functionality. (If scripting is turned off, the user will not receive any notification until after the session has expired.)

Users should be notified when a session time-out is about to occur. Providing the user with sufficient warnings and the opportunity to request more time can help the user to avoid losing data. User notifications should be written in such a way that they are clear and not intimidating.

When there has been no transmission (for example, continuing to the next page in an application) on an open connection for the established interval, the user should receive two or more alerts. The initial pop-up messages notify users that the application will time out within a designated time frame, provide them the opportunity to extend their time, and specify the length of the available time extension. The final alert also appears in a pop-up window and notifies users that their time has expired and the session has ended. It also informs them of what data may have been lost as a result and how to begin a new session. (If client-side scripting is turned off, the user will not receive any of the pop-up alert messages.)

It is important to provide users with the opportunity to recover gracefully from an expired session. This means clearly notifying them that the session has ended, indicating what data may have been lost, and providing the ability to immediately log back into the application.

To support this accessibility solution, an application or family of applications should establish a consistent and reusable architectural component for providing the time-out functionality. The time-out interval should begin when the user enters a page. Any interaction with the server (such as progressing to the next page or submitting information) should reset the timer.

A pop-up window (a *first alert* message) should notify the user that the application will time out within a certain time interval and provide the user with the option to extend the time interval. If more time is not requested, or no more extensions are possible, a pop-up window containing the *final alert* should notify the user that the time interval has expired and the session has ended.

An HTML page should appear when the time interval has expired. This page also should appear if the user attempts to perform an action on a page after the session has already expired. It should provide the ability to immediately log back in to complete the application or to return to a reasonable point of reentry.

Implementing time-out functionality involves a unique challenge. The client must contact the server within the inactivity interval to inform it that the user has requested additional time. The server must then respond by sending back an HTML page, potentially interfering with the partially completed form. One technique for handling this response from the server without forcing the user to begin the form again is by sending the form and data to the server (without performing any server-side validation), temporarily saving the data in a separate location, sending the form and data back to the browser as the response from the server, and using a combination of XHTML and JavaScript to reset the field values to their previous values and reset the focus to its previous location. This solution, however, is very complicated and should be weighted against using an alternative, the simpler frameset solution.

The frameset technique involves creating a parent frame and two child frames. One of the child frames is used as the form that the user fills out, and the other is a hidden frame. When the time-out is about to occur and contact needs to be made with the server to keep the connection alive, the hidden frame is used as the target for making the call to the server. The response that is returned from the server can then be directed to that hidden frame unbeknownst to the user, and the hidden page can be used to restart the timer. In this way, the connection to the server is extended, and the user can continue filling out the page without interruption. This solution is currently used in public Web applications, but is somewhat complicated. In addition, hidden frames can create significant confusion if they are read by screen readers. (With some screen readers, this problem can be avoided by setting the height and width of the hidden frame to zero.) Developers have continued to search for better solutions.

As this paper was being prepared for publication, it was discovered that HTTP (Hypertext Transfer Protocol) actually includes a straightforward means of preventing the response from a server from interfering with the partially completed form that submitted the request to the server. This can be accomplished as follows. The client sends a request for additional time to the server, specifying a desired response of status code 204. The server then sends a response to the client consisting of only an HTTP status line with status code 204 (signifying that no content is coming) and a blank line, which does not interfere with the current page. Early testing indicates that this method may, in fact, provide a simple, elegant solution.[23]

Error handling for accessibility

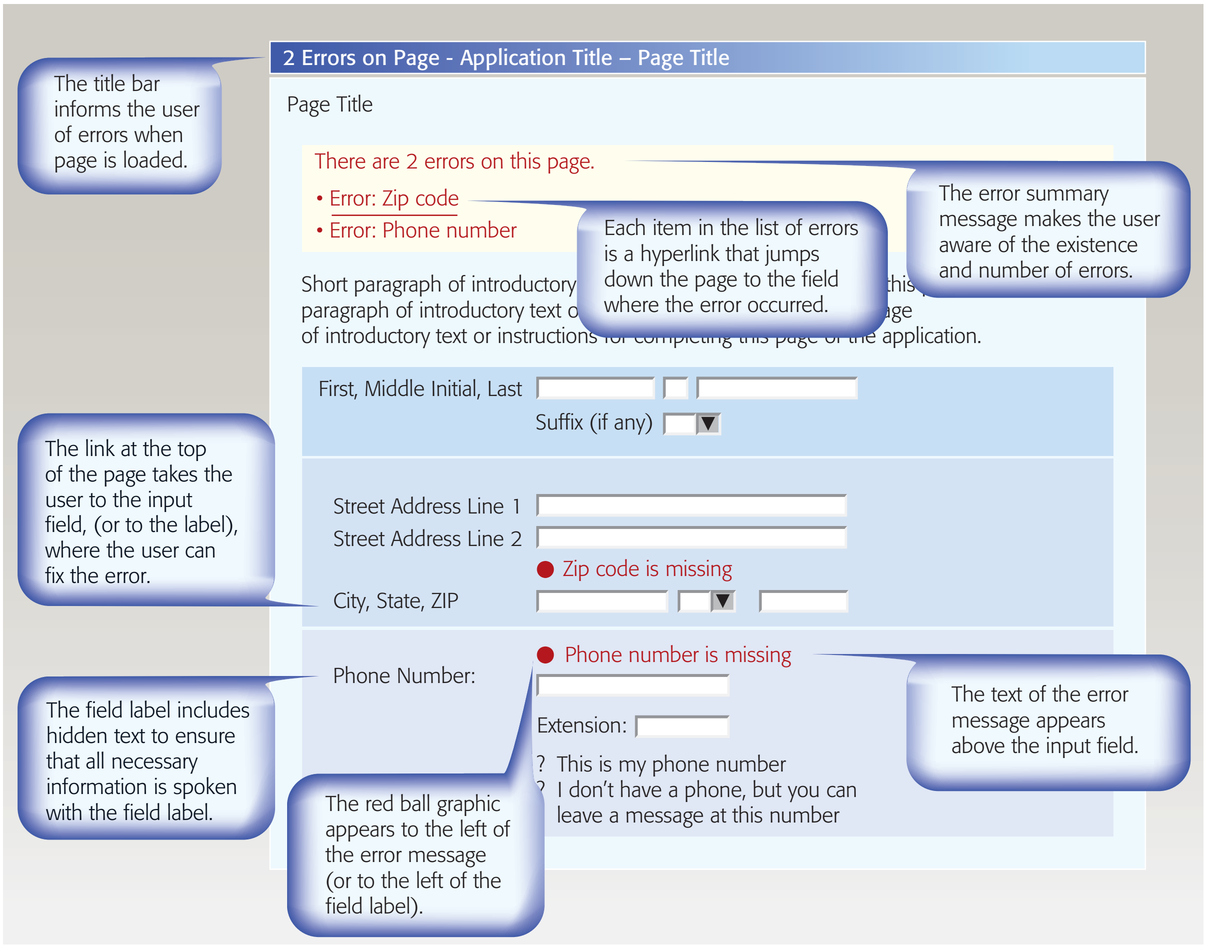
Several important architectural aspects of error handling have already been covered in earlier sections: providing comparable access to on-screen signposting (ensuring that all users are aware of the existence and number of errors) and supplementing information (ensuring that all users have comparable access to error messages and cues associated with a particular field).

Another important architectural aspect of error handling is ensuring that users can find the errors on a page. Typically, visual users can scan the page from top to bottom, looking for a distinctive visual cue that identifies the error. Users with disabilities require a comparable, efficient means of finding errors.

Even when input fields are designed with error prevention in mind, errors sometimes still occur. The application generates error messages for the fields that are in error, which typically must be corrected before the user can proceed to the next page. However, this can present challenges for several groups of users.

Because users who are color-blind may not be able to distinguish red from black, it is not acceptable to rely on color alone to enable users to find error locations. Users who have low vision may have difficulty finding where the errors occurred on the page, especially if the page is long, because with a magnified screen they may not be able to see the error text and they may need to search for it by scrolling up, down, and sideways until they find the errors. Users who are blind may need to tab through each field on the page to listen for error messages. This can be extremely time-consuming and frustrating. Users who have mobility impairments may find it difficult and frustrating to tab through a long page to get to an error when they can plainly see the location of the field.

A user should be able to quickly identify the locations of errors and navigate to them easily. There are several possible ways to do this. One approach is to display a page with only the error



2 Errors on Page - Application Title – Page Title

Page Title

There are 2 errors on this page.

- Error: Zip code
- Error: Phone number

Short paragraph of introductory … this … paragraph of introductory text o… …age of introductory text or instructions for completing this page of the application.

First, Middle Initial, Last

Suffix (if any)

Street Address Line 1

Street Address Line 2

● Zip code is missing

City, State, ZIP

Phone Number:

● Phone number is missing

Extension:

? This is my phone number

? I don't have a phone, but you can leave a message at this number

The title bar informs the user of errors when page is loaded.

The error summary message makes the user aware of the existence and number of errors.

Each item in the list of errors is a hyperlink that jumps down the page to the field where the error occurred.

The link at the top of the page takes the user to the input field, (or to the label), where the user can fix the error.

The field label includes hidden text to ensure that all necessary information is spoken with the field label.

The red ball graphic appears to the left of the error message (or to the left of the field label).

The text of the error message appears above the input field.

Figure 9
Error handling

fields on it. This makes it easy to see and navigate to the errors. However, that approach is rarely used because it requires the coding of a separate page and can confuse users. It also may prevent users from revising related errors that were not caught by the system logic. Another possible approach is to use individual pop-up alert messages for each error. The focus can even be moved directly to the error when the user acknowledges the pop-up alert. This approach is not commonly used because users may become frustrated by repeated pop-up error messages when they make multiple errors on a page. If this approach is used, it is especially important to also use error prevention techniques in order to minimize the appearance of numerous error pop-up alerts. Additional considerations include client-side browser issues and complications associated with providing client-side error handling for errors that require server-side error checking.

The common approach for presenting errors is to display the same page that was seen the first time but with errors clearly marked. The best way to accelerate this process and ensure that all user groups are able to find errors is to provide a bulleted list of links to the error fields at the top of the page, as shown in *Figure 9*. Each link should contain the word “error,” followed by the field label of the field that contains the error or a shortened version of that field label. Selecting the link should move the focus to the erroneous field.

An application or family of applications should establish a consistent and reusable architectural component for providing a bulleted list of links to errors. The business logic component of the application should determine whether errors occurred.

The text for the abbreviated field label (used in the list of error links at the top of the page) should be specified in an attribute in the HTML page or from data repositories, if they are being used to define field attributes. This ensures that there is a placeholder in the architecture to accommodate the text for the same-page navigation links to the error fields.

A reusable component should read the abbreviated field-label names (or shortened versions of those names) and link target information from an attribute in the HTML page or from the data repository. Whenever an error is indicated by the business logic component, the reusable component should generate the appropriate error link by inserting the data into placeholders. The same reusable component should automatically supplement the field-label information to reflect the error status by adding a nonvisual component to the beginning of the field label stating the word "error" and adding the actual error message to the end of the field label. This ensures that the error status information is available to non-visual users, even when they tab through the page instead of using the error links.

Multiple user-interface views

In many situations, the benefits of accessibility are broad, and the general user population is likely to benefit from a solution that addresses the special needs of users with disabilities. For example, if sidewalks have curb cuts, they are accessible to people in wheelchairs and to people pushing baby carriages and shopping carts, as well as to those on bicycles and roller blades.[24] Likewise, if software can be used without vision, it is accessible to people who are blind and to people who are in darkness or whose eyes are otherwise occupied.[25]

There is a tendency, however, to oversimplify the issue, with claims that accessibility universally benefits all users.[26,27] Often, accessibility enhancements do improve general usability; at other times they have no impact on general usability; and at still other times they can have an adverse effect on general usability. Likewise, usability enhancements often have a positive impact on users with disabilities, but there are times when usability enhancements have a negative impact on accessibility. In summary, the best solution for one group sometimes compromises the needs of another group.[14,28,29] There are times when trying to design one solution that meets everyone's needs results in a solution that is inadequate for everyone.

User interface elements that commonly cause trade-offs in software include images that are used to convey information (which benefit sighted users but pose potential challenges for users who are blind or have low vision), graphical icons or clickable images (which benefit sighted users but pose potential challenges for users who are blind, have low vision, or have mobility impairments), and title attributes (which benefit users who are blind but pose potential challenges to users with mobility impairments). Similar trade-offs apply to tab stops on text, multicolumn page layouts, left-justified field label layouts, and differences in control type preferences. These issues and more are explained in the following.

A first step toward resolving contradictory requirements can be the use of multiple views of an application. In Reference 14, two of the authors advocated providing alternate or multiple views to address trade-offs between different types of user groups and to optimize the user experience of those user groups. In the following, we take a comprehensive look at specific features of such multiple views and their architectural implications. It should be noted that using multiple views introduces an added dimension of complexity. Nevertheless, developers increasingly prefer to develop multiple-view solutions rather than deal with the complexities of conflicting requirements in a single view.

We do not suggest separate applications or pages for groups with special needs because we recognize that a legitimate concern exists that such dual source applications or separate interfaces would not be designed or maintained equally. Instead, emerging technologies that enable dynamic and customized views of the same application page may present an opportunity to improve overall user experience.

Simply to meet existing requirements such as Section 508, an architecture should include support for customizing all content presentation using the style sheet. There should not be any direct specification (hard coding) of appearances (colors, fonts, sizes, styles) in the HTML files. Users must be able to customize text fonts, color, and size, and even be able to use the page with the style sheet turned off, if they so require or prefer, or to replace the style sheet with their own style sheet.[30] However, in practice, it is not realistic to assume that most users will be capable of sophisticated customization, especially if it requires creating their own style sheets. Therefore, rather than providing a single style sheet for all users, we propose providing users with carefully designed specialized style sheets, each specifically targeted toward a different user group, and providing an easy-to-use mechanism for choosing among them. While the primary focus of providing multiple views is on the use of multiple style sheets, some features would require customization at the DHTML level. When necessary, the equivalence of DHTML "alternate" accessible views can be ensured by defining information sources on an abstract level and deriving the specific page implementation based on the view preference.

The following are some of the features that could be accommodated through the use of a multiple-view architecture:

- *Images*. When information is conveyed through graphics, it is not accessible to all users. This issue is challenging because sometimes information is conveyed in graphics precisely because it makes more sense in a graphical form than it does in text. For example, many user manuals contain screen captures or illustrations. Online applications may contain bar charts and other graphical representations of data. Multiple views should allow visual users to see the images by default and should replace the images with text for users who are blind or have low vision. For users of voice input devices who may have trouble knowing how to activate graphical icons or clickable images, the icons or images can either be supplemented with text or simply replaced with text.

- *Page layout*. Common problems for users with low vision include objects that are too far apart and the need for both horizontal and vertical scrolling around the page, which can be disorienting. Using a layout where field labels are right-justified for users with low vision and left-justified in a standard view can alleviate this issue. In addition, using an entirely different narrow view could reduce horizontal scrolling.

- *Changing controls*. Some types of controls work better than others for a particular user group. Multiple views could enable the replacement of one control with another. For example, the "A through Z" links in an alphabetical index can be replaced with a drop-down list so that users who are blind or have mobility impairments do not need to navigate through the whole alphabet to make a selection. Another example is a set of radio buttons with a large number of options, or any set of radio buttons in a frequently used application. Although most users do best with radio buttons, screen-reader users benefit from a drop-down list because it enables them to quickly and directly navigate to the desired option with a keystroke. Both control options can appear in the same HTML code. Each style sheet can display one of the controls and hide the other one.

- *Internal page navigation*. When a page includes more than two unique sections, it can be difficult for users who are blind to navigate to different parts of the page. Existing accessibility requirements specify the use of "skip" links to provide a means of skipping to the main content of the page. Providing a bulleted list of same-page links in a special view can enhance that feature to include skip links to various sections of the page, rather than just a single "main content" section. The standard view would not include the bulleted list of page navigation links.

- *Field-level help tab behavior*. By default, text on HTML pages does not receive focus when a user is tabbing through a page. As a result, non-visual users who tab through a Web form will jump past any directions and cues, without even knowing that they appear on the page. There are no formal standards that specifically require text on a Web page to receive a tab focus. However, when a keyboard is the interface to a form or application, users typically tab through the controls. Enabling the screen text to receive the focus allows non-visual users to access and hear all page elements when tabbing through the form. Section 508 requires that Web forms allow people using assistive technology to access all information, field elements, and functionality required for completion and submission of a form, including all directions and cues. It also requires access to and use of information that is comparable to that provided to nondisabled individuals. Just as the only way to provide comparable access to a screen text field label is by associating it with the field so that it is read with the field, so too, comparable access to associated directions, help, or examples must be provided. Providing a tab focus is a means of providing comparable access. Of the most popular browsers, only Internet Explorer supports the use of the "tabindex" attribute (typically tabindex=0) to provide a tab focus to screen text. There is a significant need for a W3C-compliant means of associating text field-level help information with a particular field, so that the user can be made aware that the help information exists and access it with an assistive technology keystroke. Similarly, there is a need to enable screen readers to have tab stops on specific strings of text without affecting functionality when a screen reader is not being used. When forms contain field-level directions, help, or examples, a key technique for providing comparable access to those features is changing the tab behavior by placing "tabindexes" on the screen text. This feature should be activated in a screen reader or low-vision view. It should not be used in a mobility-impaired view because extra tab stops can make navigation frustrating and lead to user fatigue.

- *Field-level help location*. Field-level help should be placed near the field to which it applies for the benefit of most users, although the appropriate placement can vary depending on the visual layout of the page or application and the length of the help text. Nevertheless, the best placement of field-level help for the benefit of users who are blind or who have low vision is within the label tag. Alternate views could allow field-level help to be turned on and off by the user in a frequently used application, so that novices could see the field-level help and experts could choose to turn it off. In addition, alternate views could allow an application to display help and tips to the right of the data-entry fields to complement the visual layout in the standard view, but move the help and tips next to the field label, immediately below the label text, for users who are blind or who have low vision.

- *Title attributes*. Usually, title attributes provide benefit to some users, without hindering other users. The exception can be users who use voice recognition software. Title attributes that start with text that is different from the screen text can interfere with easy access to the control by voice. An alternate view for persons with mobility impairments can eliminate the title attributes.

- *Repetitions and abbreviations*. Words that are hard to pronounce or are not unique on the page can also create problems for voice access. Sometimes such situations cannot be avoided without hindering other users. Alternate views can address the situations where trade-offs are inevitable.

- *Font size and color*. The size and color of fonts can give clear and concise cues to visual users. Those same cues must be provided through extra text or other means for users with visual impairments.

- *Page organization*. In a typical Web application, information and data entry are organized in sections, pages, and other groupings. Determining exactly how to distribute that information throughout the pages often involves compromises between different usability and accessibility considerations. Multiple views can provide multiple organization options.

It is useful to separate presentation from content and from functionality. This is normally considered a best practice, but it is a prerequisite to the creation of multiple views. Developers should be directed to use styles rather than directly specifying any fonts, colors, sizes, or appearances in the HTML code. From an architectural perspective, it is important to ensure that style sheets are comprehensive enough to accommodate all presentation needs, so that no additional specific coding is required.

One should avoid using absolute text units like points and pixels. Instead, relative sizes should be used so that the user can adjust browser preferences to display text in a larger font. If possible, using style sheets rather than HTML tables for page layout is helpful. Tables should ideally only be used to format tabular information.

Styles should be named for their semantic meaning rather than for their display attributes. For example, a style should be named "error message" rather than "bold red text." Using semantically based names makes it easier to apply the correct style when multiple styles may appear similar to the developer but may have very different effects on accessibility.

A single source should be used for the page content, both to ensure in practice and to reassure in perception that the content is consistent throughout all views. When possible, style sheets should be used for providing alternate view functionality. When aspects of alternate view functionality cannot be provided by using style sheets, then XHTML or other technology layers can be used to handle that functionality. Different style sheets can be created that are optimized for each group of users (for example, users who are blind, who have low vision, or who have impaired mobility).

A mechanism can be created to enable a user to switch between multiple display preferences, including switching from one style sheet to another "on the fly." The desired view preference should be maintained both during the current session and between sessions, when appropriate. Properties of reusable objects that are not optimized for all users could indicate the specific group or groups for which they are optimized. Generating the entire application using libraries of reusable objects can facilitate the implementation of a multiple view solution.

Reviewing and adapting the architecture

Although we can work to ensure that software architectures address known accessibility issues, even the most extensive architecture planning is unlikely to anticipate every future situation and desired functionality. Even with the best of architectures, we are likely to hear yet again the exclamation, "Oh, no, we can't change THAT!" when a new requested modification is not addressed by the architecture and reaches too far into the architecture of the system to allow economically viable and timely changes to be made.

The only solution to such iterative issues is to recursively refine the architecture. While it may not be feasible to implement the desired feature in the current application, it may be feasible to conduct a post-implementation architecture review with accessibility input. Iterating the architecture is necessary in order to keep the accessibility features up to date with emerging user interface standards.

CONCLUSIONS

This paper has provided an introduction to the relationship between accessibility and architecture, described common architecture-related accessibility issues, and provided architectural guidelines for addressing those issues. These guidelines are certainly incomplete, especially regarding the use of multiple views to address the needs of different user groups, but it is hoped that they will provide a starting point for systems architects and developers who want to know how architectures can be improved, as well as for accessibility specialists who need to specify requirements for accessibility and engage in cooperative work with development teams. An increased understanding of this important relationship may lead to the development of tools and assessment techniques that assist software architects in designing to support accessibility. In our work, we will continue to evaluate accessibility issues and design solutions for their architectural implications in our efforts to improve the user experience for people with disabilities.

ACKNOWLEDGMENTS

We would like to thank our colleagues Dena Fishkind, Craig Cecil, and Jeremy Linzer for the insights that they have shared over the past four years and during the writing of this paper. Their expertise in software development and architecture was instrumental to the writing of this paper. We are also grateful for the support of Sean Wheeler, the technical lead for usability at the Social Security Administration, and John Palumbo, the lead advocate for software accessibility within Operations at the Social Security Administration.

CITED REFERENCES AND NOTES

1. J. Thatcher, M. Burks, S. Swierenga, C. Waddell, B. Regan, P. Bohman, S. L. Henry, and M. Urban, *Constructing Accessible Web Sites*, Glasshaus, Birmingham, U.K. (2002).
2. *Developer Guidelines for Web Accessibility*, IBM Web Accessibility Center, http://www-306.ibm.com/able/guidelines/web/accessweb.html.
3. M. Theofanos and G. Redish, "Bridging the Gap: Between Accessibility and Usability," *ACM Interactions* **10**, No. 6, 36–51 (November–December 2003).
4. *Creating Accessible Forms*, WebAIM—Web Accessibility in Mind (2005), http://www.webaim.org/techniques/forms/.
5. S. Faulkner, *Techniques for Making Forms More Accessible*, Accessible Information Solutions, National Informational Library Service (NILS) (2004), http://www.nils.org.au/ais/web/resources/WSG_Oct_04/toc.html.
6. I. Lloyd, *Accessible HTML/XHTML Forms*, The Web Standards Project (May 2004), http://webstandards.org/learn/tutorials/accessible-forms/01-accessible-forms.html.

7. *Permanent Archive: Invisible Form Prompts*, Juicy Studio (September 18, 2004), http://www.juicystudio.com/invisible-form-prompts.asp.

8. J. Thatcher, *Accessible Forms*, JimThatcher.com Accessibility Consulting (2002), http://www.jimthatcher.com/webcourse8.htm.

9. I. Lloyd, *Better Accessible Forms*, Accessify.com (2002), http://www.accessify.com/tutorials/better-accessible-forms.asp.

10. D. Hoffman and L. Battle, "Top 20 Design Recommendations for Accessible (and Usable) Web Applications," *Proceedings of the Usability Professionals' Association (UPA) Conference* (2005, forthcoming).

11. *Section 508 of the Rehabilitation Act of 1973*, http://www.section508.gov/.

12. *The World Wide Web Consortium (W3C) Web Accessibility Initiative (WAI)*, http://www.w3.org/WAI/.

13. U.S. Department of Justice, Americans with Disabilities Act, ADA Home Page, http://www.usdoj.gov/crt/ada/adahom1.htm.

14. L. Battle and D. Hoffman, "Design Patterns and Guidelines for Usable and Accessible Web Applications," *Proceedings of the Usability Professionals' Association (UPA) Conference*, p. 4 (2004).

15. *How Do You Define Software Architecture?*, Carnegie Mellon Software Engineering Institute (2005), http://www.sei.cmu.edu/architecture/definitions.html.

16. P. Clements and P. Kogut, "The Software Architecture Renaissance," *Bridge*, Issue 3, The Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, pp. 11–18 (1994).

17. J. Bosch, *Design and Use of Software Architectures: Adopting and Evolving a Product Line Approach,* Pearson Education, Addison-Wesley and ACM Press (2000).

18. N. Juristo, A. M. Moreno, and M. I. Sánchez, "Clarifying the Relationship between Software Architecture and Usability," *Proceedings of the Sixteenth International Conference on Software Engineering and Knowledge Engineering*, p. 2 (2004).

19. B. E. John, B. L. Bass, N. Juristo, and M.-I. Sanchez-Segura, "Avoiding 'We can't change THAT!': Software Architecture and Usability," Tutorial materials presented at the Conference on Human-Computer Interaction (CHI 2004), Vienna, Austria (April 24–29, 2004), pp. 5–6 & 23–24, http://www-2.cs.cmu.edu/~bej/usa/publications/FinalCHI2004Tutorial.pdf.

20. L. Bass, B. John, and J. Kates, *Achieving Usability Through Software Architecture*, Carnegie Mellon Software Engineering Institute technical report (March 2001).

21. Section 508, Subpart B § 1194.22 (p), http://www.section508.gov/index.cfm?FuseAction=Content&ID=12#Web.

22. *Web-Based Intranet and Internet Information and Applications*, The Access Board (June 2001), http://www.access-board.gov/sec508/guide/1194.22.htm.

23. Private communication with L. McLeroy.

24. E. Bergman and E. Johnson, "Toward Accessible Human-Computer Interaction," in *Advances in Human-Computer Interaction, Volume 5*, J. Nielsen, Editor (1995), http://www.sun.com/access/developers/updt.HCI.advance.html.

25. G. C. Vanderheiden and S. L. Henry, "Designing Flexible, Accessible Interfaces That Are More Usable by Everyone," Tutorial presented at the Conference on Human Factors in Computing Systems (CHI 2003) (2003), http://www.chi2003.org/docs/t10.pdf.

26. S. Pemberton, "Accessibility is for Everyone," *ACM Interactions* **10**, No. 6, 4–5 (November–December 2003).

27. S. Salamone, "Improved Web Access for Disabled Users Benefits All," *TechRepublic* (December 2001), http://techrepublic.com.com/5100-6301-5032988.html.

28. J. Moore and J. Mathews, "The Blind Leading the Blind," http://www.aarp.org/olderwiserwired/oww-events/Articles/a2004-03-03-oww-blind-leading-blind.html.

29. G. Redish and M. Theofanos, "Achieving Experience Equity and Universal Usable Access," http://redish.net/content/talks.html.

30. Section 508 Standards, Subpart B § 1194.22 (d), http://www.section508.gov/index.cfm?FuseAction=Content&ID=12#Web.

Accepted for publication February 24, 2005.
Published online August 4, 2005.

David Hoffman
MILVETS Systems Technology, 4675 Annex Building, 6401 Security Boulevard, Baltimore, Maryland 21235 (dyhoffman@yahoo.com). Mr. Hoffman is a senior accessibility and usability specialist with extensive Web technology experience. In his current position, he provides user interface accessibility and usability design support for the Social Security Administration. He serves as a top agency expert on interactions between Web applications and assistive technologies, as well as on achieving Web application accessibility. In addition to mentoring various project teams in accessibility issues, Mr. Hoffman contributes to the definition of implementable interactive standards that incorporate both accessibility and usability. Mr. Hoffman holds a master's degree in applied information technology from Towson University. He is a member of the Usability Professionals' Association (UPA).

Eric Grivel
Lockheed Martin Information Technology, 3300 Lord Baltimore Drive, Baltimore, Maryland 21244 (egrivel@acm.org). Mr. Grivel is a senior software developer with 18 years of professional experience in architecture and application design and implementation. In his current position at Lockheed Martin, he supports the Social Security Administration's Web applications. Mr. Grivel has a master's degree from the Delft University of Technology. He is a member of the Association for Computing Machinery (ACM).

Lisa Battle
Lockheed Martin Information Technology, 3300 Lord Baltimore Drive, Baltimore, Maryland 21244 (lbattle@acm.org). Ms. Battle is a senior user interface designer with more than 12 years of experience creating usable software, Web-based applications, and Web sites for clients in a variety of industries and the federal government. Her work focuses on making users successful and achieving business goals through a combination of analysis and iterative design techniques. Over the past five years, Ms. Battle has been instrumental in introducing user-centered design into the Social Security Administration, as well as contributing to standards definition and integrating user-centered methods into project life cycles. Ms. Battle holds a master's degree in cognitive psychology and human factors from George Mason University. She is a member of the Usability Professionals' Association (UPA) and the Association for Computing Machinery (ACM-CHI). ■