Closing the Competitive Productivity Gap for People with Disabilities

Software Engineering Guidance and Challenges

March 18, 2009

"Much is known about designing usable accessibility into software, but there remain substantial and long-standing fundamental gaps in our knowledge – including how to provide usable access to static text."

Matt King
IBM I/T Chief Accessibility Strategist
IBM, Office of the CIO
Voice: 719-520-3006
E-mail: mattking@us.ibm.com

Brief abstract

One of the ultimate goals of the many electronic and information technology (E&IT) accessibility standards is to enable full employment of people with disabilities. Yet, these standards are not and can not reasonably be expected to ensure the level of usability for people with disabilities (usable access) that will enable them to be competitively productive. That level of usability can be attained by incorporating people with disabilities into user centered design practices, however it is unreasonable to expect that will ever be done for the majority of E&IT solutions. This paper posits there is a middle ground that will solve 90% of the problem with 10% of the effort by mimicking another common software engineering practice that could be called design by convention or heuristic UI design. This is the practice of piecing together commonly recognized, and sometimes proven, design patterns or conventions. This paper summarizes some of the known conventions that are particularly beneficial to usable access and addresses a major area of concern where there is no generally applicable engineered solution – static and other non-editable text. Lack of usable access to static text for screen reader users is a major gap that spans all platforms, accessibility application programming interfaces (APIs), and standards.

Introduction – Where standards leave off

Standards bodies, government agencies, advocacy groups, assistive technology vendors, and the IT industry are all investing heavily in several electronic and information technology (E&IT) accessibility standards – US section 508, web content accessibility guidelines (WCAG) 2.0, ISO 9241, and others. One of the ultimate goals of such standards is to close the productivity gap between users of IT solutions who have a disability and users who do not have a disability. For example, US section 508 requires, for covered individuals and IT solutions, that people with disabilities, "have access to and use of information and data including communication that are as timely, accurate, complete, and efficient as compared to that provided to the public who are not individuals with disabilities, unless an undue burden would be imposed on the agency."[1]

In other words, the clear intent of section 508 is that IT solutions used by US government agencies provide competitive productivity for people with disabilities. This means people with disabilities who work in federal agencies should be able to accurately use all aspects of IT solutions with a level of efficiency that enables them to compete fairly with their peers who do not have a disability. IBM® has a similar standard that requires all IBM information technology to provide the same benefits and priveleges to people with disabilities that are afforded those who do not have a disability.

Yet, in US section 508, "The determination of timely, accurate, complete, and efficient will not be a quantifiable measure."[1] Not surprisingly, this is the case in IBM standards as well. Standardizing measurements of these attributes, especially efficiency of task completion, is highly problematic. How would tasks be defined such that a collaboration product from vendor A could be compared to a product from vendor B? Would all products have to be measured with the same end users so that the outcomes would be comparable? The list of questions and potential problems is practically endless.

While these unanswered questions do not necessarily render the problem intractable, the practical approach taken by E&IT accessibility standards is to require specific user interface (UI) capabilities and characteristics, e.g., alternatives to non-text content and the ability to access every function from the keyboard. Unlike efficiency, these required capabilities and characteristics are verifiable in real-world practice. While they are necessary prerequisites for realizing the vision of usable access that enables competitive productivity, such requirements can only lay the foundation for that vision – not guarantee its fulfillment. This is aptly demonstrated both when we see people with disabilities flock to and proclaim the accessibility of non-compliant products and when we hear people with disabilities complain about lack of accessibility in compliant solutions.

This limitation of the standards is understandable. The realm of E&IT standards with requirements like full keyboard functionality is primarily, but not completely, functional design whereas the realm of outcomes sought by the standards is purely usability (e.g., efficiency of keyboard navigation). This is not to say technical accessibility and usability

are separate concepts or that there are not any usability concepts in the accessibility standards. In fact, more than half of most accessibility standards require human judgment to evaluate, and a significant portion of that evaluation involves some aspect of usability. However, the aspects of usability covered by the standards are, as previously described, limited to what is easily verifiable by virtue of having functional characteristics. There remains a significant gap between these dominantly functional accessibility requirements and what is required to provide usability for people with disabilities.

A partial bridge – user centered design

The most successful bridge across the gap between functional requirements and usability is user centered design. Significant work has been done, both inside and outside IBM, e.g., “Conducting user evaluations with people with disabilities”[2], to develop best practices for incorporating people with disabilities into user interface design processes. This is what is known as user centered design for people with disabilities, and it ensures usable access. It is essential that some critical mass of solutions employ user centered design for people with disabilities in order to develop a sound understanding of usable access.

However, because user centered design for people without disabilities is employed in only a small subset of all solutions, it is unreasonable to think it will be the norm for people with disabilities. The most obvious barrier is marginal cost. But, even if money were no object, large scale adoption of user centered design for people with disabilities is unthinkable when we consider the number of software projects in flight at any one time and the number of people with disabilities alive, let alone the subset who would actually be available to participate in the design process. There simply are not enough people with disabilities available to help design even a fraction of the software they may need to use on the job. Given this, how will we ever put even a small dent in the massive productivity gap created by the lack of usable access in main stream software?

Much of the main stream software designed without fully employing user centered design practices still manages to reasonably meet many end user needs. This is possible because the designs tend to follow well-established design patterns and conventions established by market leaders. Many of these patterns and conventions are well baked into UI development tool kits, further enhancing the likelihood of successful design by convention, or heuristic UI design.

Although heuristic UI design has significant limitations and pitfalls, it is particularly valuable in practical efforts to deliver usable accessibility. It may bridge as much as 90% of the gap between unusable access and completely usable access at a fraction of the cost of user centered design for people with disabilities. The suggestion here is not to scrap user centered design for people with disabilities but rather to leverage it by developing heuristic design principles and patterns that can be applied when user centered design is not practical.

Principles of usable access UI design heuristics

To develop an understanding of the principles of usable access UI design heuristics, it is helpful to relate them to current accessibility standards. This is most easily done with W3C's web content accessibility guidelines (WCAG) 2.0[5] as they have been intentionally built upon a set of guiding principles. These same principles are generally applicable to any of the industry accessibility standards. The four principles of WCAG 2.0 are to ensure that web content is perceivable, operable, understandable, and robust.[6] Understandability primarily falls into the realm of usability while the other three principles are generally more functional in nature.

We can extend the WCAG 2.0 principles into the arena of usable access by applying seven common principles of usable design: discoverable, complete, intuitive, consistent, efficient, logical, and customizable. For example, not only should it be possible for the user to perceive information and features, but the information and features should be easily and completely discoverable. Operable features that are usable will be consistently, efficiently, and intuitively operable. Understandable information that is usable will appear logical to the user. Robust features that are usable will adapt to the users needs and in some cases be customizable. Naturally, these principles are among those the industry has long been using to evaluate the recommended implementation techniques included in E&IT accessibility standards. The challenge for the UI designer is learning how to knit together a complete user experience that incorporates these principles.

Overview of usable access UI design heuristics

A good set of usable access UI design heuristics will enable THE designer to readily understand the difference between a UI that implements accessibility standards and is very usable by people with disabilities and another UI that also follows the standards but is less usable. For example, a designer could assemble a collection of web widgets that individually implement recommended WAI-ARIA best practices[2], creating an accessible UI but not necessarily one with a keyboard interface that provides usable access. The designer must also specify a navigation order that matches the flow of task completion. At the same time, the keyboard navigation sequence must be efficient when compared to the mouse click sequence, which may require the designer to consider how many tasks are supported within a particular navigation sequence.

We can develop usable access UI design heuristics by:

1. Studying how the principles of usable access UI design heuristics are manifest in user interfaces that have features commonly considered usable by people with disabilities;

2. Applying principles of usable access UI design heuristics to the analysis of user interfaces with features that are technically compliant with standards but not considered usable by people with disabilities.

At IBM, we have begun to develop best practices to aid our designers in developing solutions that are more usable by people with disabilities. The usable access UI design heuristics that have been developed so far are summarized in Table 1.

Table 1: Usable access UI design heuristics

Heuristic Category	*Descriptive Example*
Consistent Keyboard Navigation and Key Assignments Keyboard navigation and key assignments are consistent within the solution and with the platform conventions.	Ensure all keyboard behaviors expected on a platform are provided, especially in custom widgets.
Logical Navigation The keyboard navigation order is logical and enables efficient task completion.	Minimize the number of key strokes to complete a task. Includes consideration of how screen reader users have to "see" the task by pressing keys.
Discoverable Key Assignments Key assignments are easy to determine by being exposed in the user interface and effectively documented.	Provide a dynamic keys index that reflects customization.
Intuitive Key Assignments Key assignments are easy to learn and remember.	Assign keys that are a natural extension of platform or solution key assignments.
Customizable Key Assignments Users can customize key assignments to meet their individual requirements.	In applications with extensive or rich keyboard interfaces, allow the user to customize the key assignments.
Tolerant Direct Manipulation The direct manipulation interface makes allowances for imprecise and/or slow pointer movements (mouse, etc).	Design drop-down, slide-out and pop-up menus to allow for imprecise mouse movement before item selection.
Consistent Presentation The presentation of information and user interface elements is consistent within the solution and with the platform conventions.	User interface elements that perform the same function must have consistent labels.
Logical Presentation The structure and relationships of information and user interface elements are logically	Use separate lists instead of listing information that contains multiple pieces of data. For example, when the user is

conveyed.	required to select a city and state, break the data into two lists, one listing the states and another listing the cities.
Clear Visual Presentation The visual design allows the user to clearly see information presented in the user interface.	Do not rely on a blurred image to indicate the state of a user interface element (e.g., blurred = unavailable).
Complete Presentation All information is discoverable and effectively consumable.	Screen reader users must be able to navigate static text.
Alternative Methods Easily discoverable and efficient alternative methods for completing tasks are available when the primary interaction methods do not enable a level of productivity for users with disabilities that is competitive with the productivity of users without disabilities.	When visual scanning of data is required, provide search, sort, and filter capabilities.

The above list of usable access heuristic categories and their supporting heuristics and design recommendations is not complete and is a work in progress.

Competing needs, conflicts, and gaps

The categories of design heuristics described above all have some widely accepted practices from which we can readily identify traits that contribute to usable access and thereby form specific design heuristics. However, we have also found a few areas where research and innovation are required in order to develop generally applicable guidance. The problems have arisen in the following situations:

- The needs of one disability group conflict with the needs of another.
- The needs of people with disabilities compete with those of people without a disability.
- There currently are not any industry conventions or techniques to support the heuristic.

One example is a common practice that pits the needs of sighted keyboard-only users against the needs of blind users. It arose during our work at IBM when developing examples of how the number of keystrokes required to complete a task can be minimized. The technique that was considered but not included as a best practice for want of supporting data was:

“Unique keys: Shortcuts (accelerators) and mnemonics should be unique, and not duplicated within the context they are used. For example, do not set the mnemonic F for File and Format within the same menu.”

This technique is very helpful to keyboard users who can visually scan the menu to find a particular item and then identify the underlined letter for the accelerator key. However, using a letter other than the obvious and intuitive first letter of a function name for an accelerator key simply so the assignment can be unique dramatically reduces the intuitiveness of the assignment. This makes the assigned key far more difficult to recall. When a blind user is focused on the task at hand, as opposed to memorizing the user interface, the accelerator key may be forgotten in just minutes. Ultimately this reduces efficiency for the blind user who ends up pressing the arrow key many times to get to the menu item instead of using the accelerator. Generally it would be more efficient for a blind user to press a non-unique accelerator key several times rather than navigate through each menu item to find the targeted function.

One way this could be partially resolved would be to allow the blind user to customize the accelerators. However, there is a good possibility that approach would not yield benefit for anyone other than the rare power user. Another possibility is a single solution-wide configuration option that allows the user to choose the style of accelerator. Besides being complex, this too may not yield the desired efficiency improvement for the impacted users.

Most examples of competing needs and conflicts are relatively low impact. More pressing are the areas where there are no industry practices or conventions that yield a usable result. Two such pervasive and significant gaps are:

1. There are no conventions for providing fully functional, robust, and usable WYSIWYG capabilities to users who require high contrast display schemes. When provisions are made, the features usually lack discoverability, are not intuitive, or have significant functional limitations. The alternative approach, of course, discovered by some, is to rely on assistive technology.
2. There are no efficient, functionally complete, and easily discovered conventions for ensuring screen reader users perceive all static or non-editable text.

This latter gap is as old as screen reading technology itself, and perhaps because of its age and pervasiveness, is rarely recognized as the fundamental accessibility and usability gap that it is – one big enough to drive a truck through. Ironically, it is most often those who are new to accessibility verification that stumble upon it and are perplexed by the dearth of good answers to their questions. The remainder of this paper is devoted to further characterizing this issue.

The challenge of complete presentation to screen reader users

For the presentation of a user interface, including all the information it contains, to be both complete and effectively consumable by screen reader users, all the following are required:

- All information must be easily discoverable within normal task flow. Sighted users do not have to learn about or take special actions to ensure they are aware of what is “right in front of their face.” Blind users need the same level of discoverability to compete effectively.
- All information must be efficiently and effectively consumable. Besides being perceivable and understandable, this means that the screen reader user needs to be able to efficiently access the information. That is, perceive it in whatever logical unit the user chooses to suit his need, be that character, word, line, sentence, etc. Again, this access must fit within the normal task flow and not require unusual means, e.g., utilization of a special key combination to virtualize information.
- All aspects of the information must be communicated, e.g., structure, visual semantics.

With the exception of accessible web 1.0 content, it is rare for all three of these conditions to exist for static text. In fact, it is common in desktop software for none of these conditions to exist for most or all of the static text in an application.

How did we get here?

The cause of this predicament is obvious in retrospect. Screen reader users must see by navigating some kind of cursor. In the 1980’s, nearly all application output was static text and the review cursor was born. Screens were simple and text was all unstructured so the situation was satisfactory. Screen readers retrieved text to be read by the review cursor from an off-screen model (OSM). As UIs evolved and became more complex, so did the art of OSM construction so as to retain as much review cursor functionality as possible.

When the first accessibility standards were written, the dependency on OSMs was perpetuated with requirements that text be written via standard system calls. At that time, the modern-day alternative to OSMs -- engineered accessibility APIs -- were few and completely unproven. Now, reading static text from an OSM with a review cursor is recognized as unspeakably inferior to an engineered accessibility API. Moreover, it is an approach that does not meet the criteria for a UI that provides usable access. Nevertheless, the arcane requirements for using standard system calls to write text remain in today’s standards because the amount of legacy code that would have to be re-written to provide fully accessible, structured, static text is enormous. We simply can not afford to retrofit that much software.

What is the static text challenge?

This is an opportune moment in the history of IT for fundamental change that could ensure static and non-editable information will be equally usable by all users in the future. As a huge portion of the world’s software is being re-written to live in the cloud, UIs are being recoded as well. The open question challenging us, though, is, “what is a good general approach to ensuring all software can be engineered to meet this objective?”

After characterizing the problem in the following sections, we shall see that the solution is not simple. For example, if we could resolve this issue for rich internet applications simply by tweaking an underlying engineered accessibility API, e.g., IAccessible2[7], or by making additions to the declarative markup used to build accessible user interfaces -- WAI-ARIA[3], then the issues would already be resolved. But, before such steps can be made effectively, there is need to arrive at industry agreement on the required traits of the end user experience and the best way of engineering those traits into software systems.

There are two parts to the solution of making static text fully accessible for screen reader users: presenting the text and defining the interaction model. Presenting the text should be fairly simple but defining the interaction model and what is required to support it is not so simple. To illuminate the issues, we will start with some examples that help define the scope of the problems to be solved.

Example 1 -- Simple Dialogs

Let's consider a simple dialog with a message, an OK button, and possibly one or two other controls. The dialog is ordinarily automatically read by most screen readers on most platforms when it pops up as long as it meets today's only applicable requirement – that the text be written with standard system function calls. But, if the user was distracted or simply wishes to hear the message again, he must rely on screen reader features that typically require listening to the entire message repeated. While this may be adequate accessibility in some circumstances, in general it is not. The user needs to be able to reliably navigate all messages, especially messages that include detailed information such as a message number, phone number, file name, or URL. On some platforms, mouse emulation can provide users with the ability to read the message by character or word, but there is no guarantee that text outside the dialog will not be mixed with text inside the dialog. Some screen readers include a feature that can pull the dialog text into a custom screen reader dialog with navigation and selection capabilities, but such features are not easily discovered by the user and may not work in all circumstances; they are a bolt-on stopgap remedy.

For example, consider the dialog from IBM Lotus® Sametime® Connect 7.5.1 pictured in Figure 1. It tells a user that the server and local copies of the contact list have an unexpected difference, summarizes the differences, and asks the user what to do. While some users may be able to listen to this dialog read only once and respond to it, many others may want to re-read the details of the differences one at a time to consider how to respond. This dialog is typical in that it requires such a user to employ a screen reader's mouse emulation capability to accomplish that task – something that should never be necessary in solutions that provide usable access.

In this case, since the static text is near the top of the dialog and since there are no other UI elements inside the dialog to the left or right of the text that could interfere with comprehension, screen reader users with intermediate to advanced skills will not likely have much trouble reading the text with an emulated mouse. Locating the beginning of

the text with the mouse requires three key strokes and reading will not be interrupted or confused by other objects. This assumes that the screen reader's mouse movement restrictions are properly set. On the other hand, many screen reader users with basic to intermediate skills may struggle, especially if the screen reader's mouse movement capability is either too restricted or not adequately restricted. In either case, the user may never locate the text.

Regardless of screen reader user skill level, reading with the mouse is unintuitive and inefficient. Alternatively, if a reading cursor were automatically placed at the beginning of the critical information when the dialog first appears to a screen reader user, the text would always be discovered, could be more easily understood by allowing the user to read by word or line, and access would be as efficient as it is for other users.

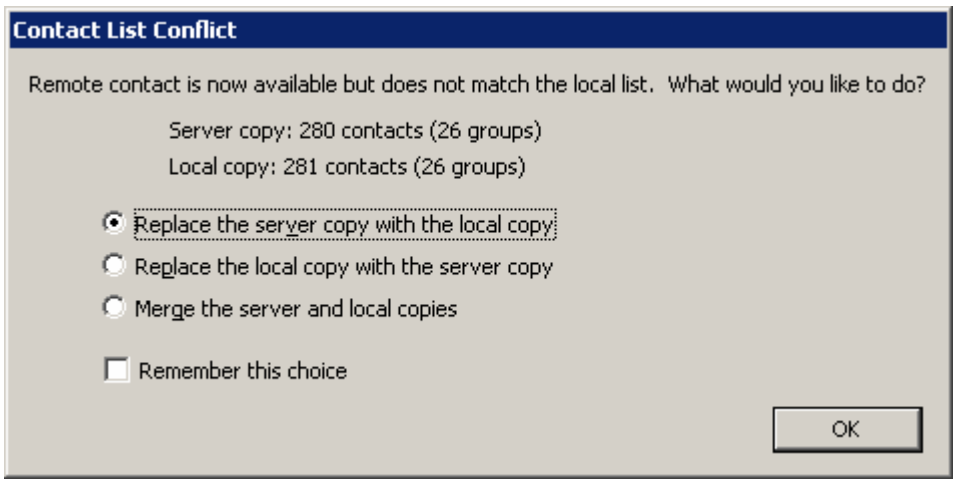

Figure 1: Lotus Sametime Connect contact list conflict dialog

One remedy for many simple message dialogs is to display the message in a widget that supports a read-only cursor, e.g., the typical read-only edit control. If the blind user can tab from the OK button to the read-only message widget, any screen reader will allow the user to read and consume the information with the same efficiency and discernment capabilities that other users have. Additional advantage can be afforded to all users if the message display widget also supports selection and copy to clipboard functionality. However, this may introduce unwanted changes to the appearance of some dialogs and adds keystrokes to the tab order. While in simple dialogs the additional keystrokes do not negatively impact sighted keyboard-only users, in other circumstances they do. Thus, putting read-only edit controls in the tab order is not an acceptable general approach to resolving static text issues.

Example 2 -- typical wizards and complex dialogs

Wizards that step users through a process frequently have significant amounts of explanatory text mixed in with a few controls that support the options in that step. In such cases, not only is the text itself important, but the reading order and possibly the structure

as well. Today, unlike the web standards, there are no requirements that the structure or reading order be clearly exposed to the screen reader user. There is no requirement that the screen reader user be able to navigate the text to facilitate comprehension.

Specific examples of this problem are abundant in products from nearly every software vendor. For example, consider the auto-summarize dialog in Microsoft® Word shown in Figure 2.

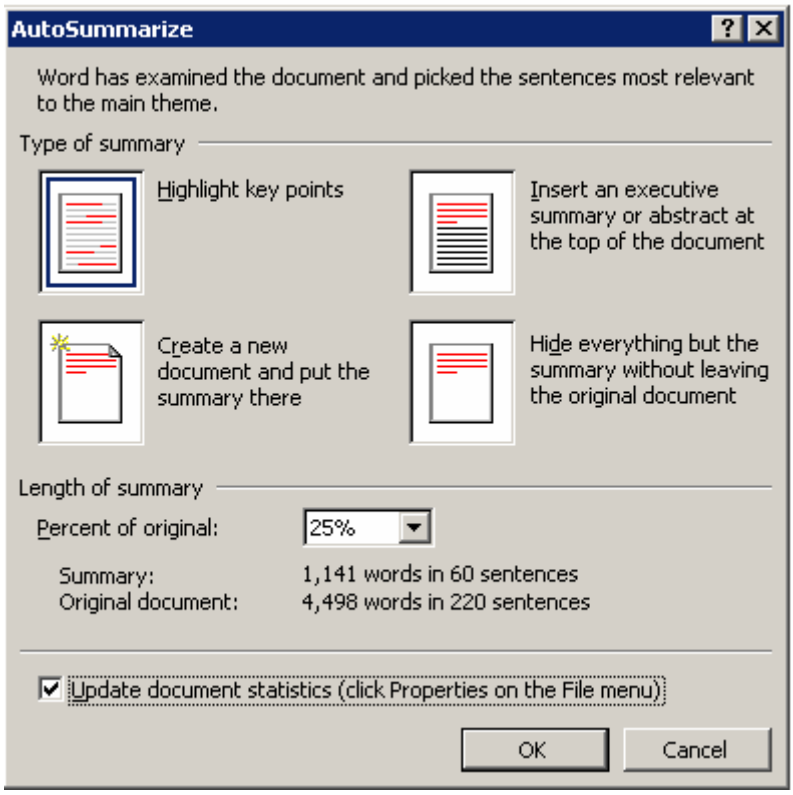

Figure 2: Microsoft Word Auto Summarize Dialog

Using JAWS®, it is almost impossible to make sense of this dialog because JAWS has no way of determining the correct reading order when it is read with the INSERT+B command. JAWS reads this dialog as follows:

Table 2: JAWS speech output for Microsoft Word auto summarize dialog

AutoSummarize Word has examined the document and picked the sentences most relevant to the main theme. Type of summary Length of summary Summary: Original Document: Underline Highlight key points Insert an executive summary or abstract at the top of the document C underline reate a new Hide everything but the document and put the summary without leaving summary there the original document

Highlight Radio button Condense Radio button Insert Radio Button checked Create radio button Percent of original: 25% 1,166 words in 62 sentences 4,612 words in 233 sentences checked Update document statistics (click Properties on the File menu) OK button Cancel button

It would be difficult to argue that hearing the text in Table 2 spoken, especially if it is spoken in one continuous stream of speech, provides a blind user with a complete, let alone a clear and understandable, picture of the dialog. Reading the dialog with the JAWS cursor isn't much more revealing. Then, if the user changes the length of summary and wishes to determine how many words are in the resulting summary, the user needs to either switch to the JAWS cursor and find that information or re-read the entire dialog.

Similarly, the Symantec™ AntiVirus installer pictured in Figure 3 is typical of many wizards. Most blind users will never be aware of the descriptions of the radio buttons unless they happen to press their screen reader key that reads the entire dialog. If they do so, the descriptions are not read coherently.

Figure 3: Symantec Antivirus Installer

Unfortunately, making the non-editable text accessible in most dialogs and wizards is not so easily remedied as it is in simple message dialogs. It could be included in the tab

order, but now we have interfered with the interaction model for all users, adding extra key presses and reducing the efficiency of those who do not have to see by pressing keys.

Example 3 -- windows containing a variety of widgets

An example that is quite different from any of the previously mentioned dialogs and wizards is the IBM Lotus Sametime Connect 7.5 chat window shown in Figure 4. It contains a business card widget that displays a business card for the person with whom the user is chatting. This widget is included the tab order. When the focus lands on the business card, a screen reader will read the entire business card in one long continuous stream of speech. There is no way to hear a portion of the card.

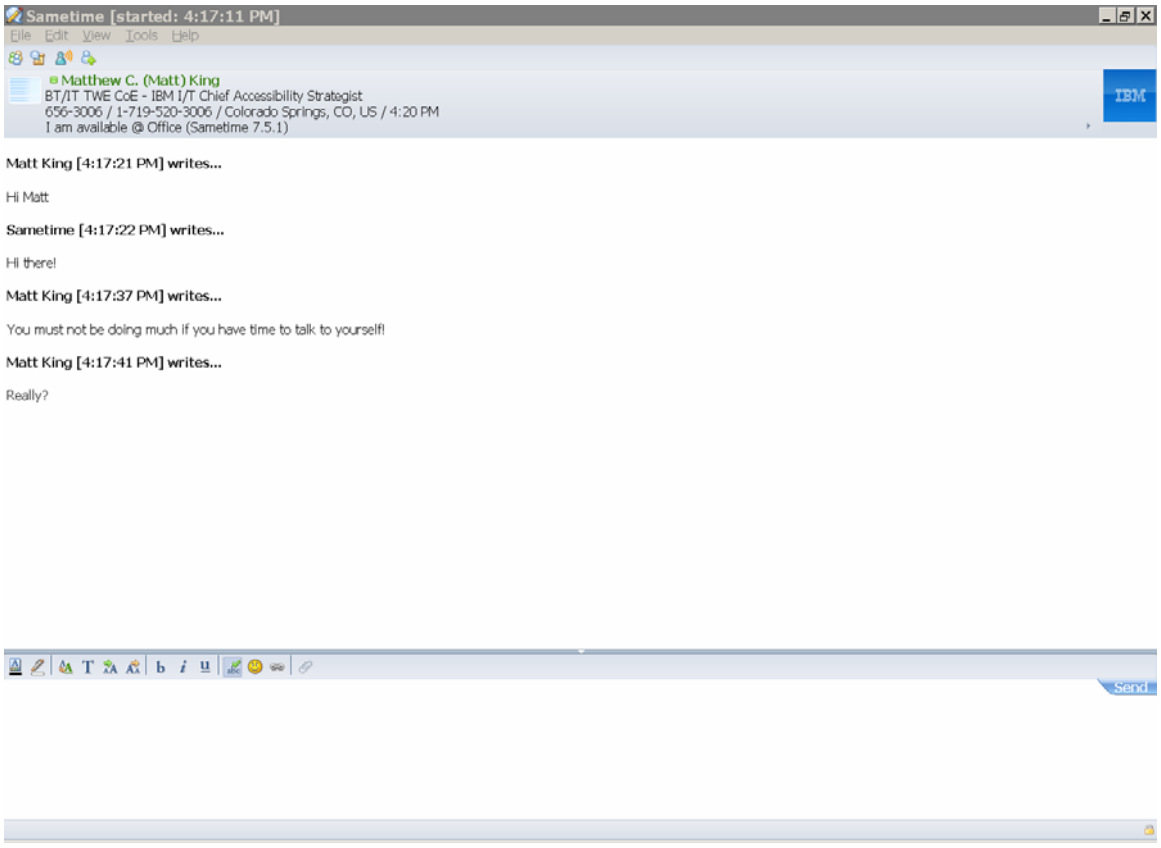

Figure 4: IBM Lotus Sametime Connect chat window with business card

Technically the business card widget is compliant with most current standards, but a blind user has nowhere near the same access to the information that a sighted person has. For example, if a sighted person wishes to read only the telephone number in order to dial it or write it down, the task is trivial. Whereas, a blind person has to listen to all of the text on the card and be prepared to memorize the telephone number at the point in time when it is read. For many users this may take several repetitions.

Since this widget is already in the tab order, the remedy could be very simple: give it the additional properties of a read-only edit. This would not only give blind users full access but also provide all other users with the useful feature of being able to copy/paste the information.

Example 4 - forms

Forms that are a mix of large amounts of read-only text and interspersed input fields represent the most important test case for accessible non-editable text, and are an especially pervasive problem. Per current accessibility standards, it is adequate to give the user directions and cues to complete the form. US section 508 says, for example, "allow people using assistive technology to access the information, field elements and functionality required for completion and submission of the form, including all directions and cues."[1] This is not generally understood to mean, "Provide full access to all the information in the form."

For example, if you create a 10 page contract in Microsoft Word and provide a few input fields at various places in the text, the form will be technically compliant if you provide appropriate labels for the input fields. Yet, there is no way for the screen reader user to reliably read the input values in context. Further, the only way that the screen reader user can read the entire form is to "unprotect" it and read it in edit mode. That, of course, means acquiring a password in many cases, which is usually not possible. Similarly, it is impossible to create a form in IBM Lotus Notes where the instructional text between input fields can be easily navigated.

Necessary characteristics of a usable solution

We need industry conventions or standards accompanied by accessibility API support that:

1. Allow static text to be structured on any platform in any context;
2. Place static text in the navigation sequence only for assistive technology users who require it and not for other users, such that:
    a. The placement can match the natural flow of the task or information being presented;
    b. The key pressed to gain access can match the application's navigation, e.g., if the text is between 2 items in the tab order, tab gives access to the text;
    c. The text can be read automatically;
    d. The text can be logically navigated, e.g., by character, word, line, etc.;
    e. If the text contains an object, e.g., a link, that can gain application focus, it can be read in context.

The most challenging aspect of these requirements is creating an architectural model that can be implemented on any platform and allows the text to be included in the navigation

sequence for screen reader users while it is left out of the navigation sequence for other users. For the architecture to be robust, it should not require the application being accessed or the platform on which the application runs to be aware of the assistive technology and then alter its keyboard behavior. On the other hand, it is generally considered beyond the scope of an assistive technology to modify or amend the behavior of the application being accessed. So, whose behavior will change: will it be the assistive technology (AT) or the information technology (IT)? Answering this question may require more clearly defining the AT/IT division of responsibilities. In addition, consideration must be given to what roles, states, or properties in the application will trigger the desired behavior.

What about aria-describedby?

The IAccessible2 accessibility API specification includes a relationship called "describedby" that enables a UI developer to specify that a particular control is described by another control. In the WAI-ARIA specification, this is supported by the *aria-describedby* attribute. The intent of this feature is to enable static text to be associated with the element it describes. It also has the side effect of requiring that the descriptive text be contained within a discrete element, which provides additional semantic structure within the UI.

The provision of the "describedby" relationship begs the question, "Isn't this precisely what is required to resolve static text issues?" While it is helpful, it is not sufficient. The presently expected screen reader behavior as described in the CodeTalks wiki[8] is that when an element with the *aria-describedby* attribute is focused the description should be read after the element with focus is read. While this makes the description more discoverable and improves understanding, it is not very usable unless the description is short. In general application, it has the following shortcomings:

1. The description is read after the element it describes. This is the reverse of the experience of the visual user and will reduce understanding in many circumstances. Of course, this is necessary in current implementations because the reading of the element being described and the reading of the description are combined into a single operation. If the description were read first, than the user who does not need the description will suffer dramatic productivity loss.
2. The description is not navigable. Because a reading cursor is not placed in the description, the blind user can not read by character, word, line, etc. Of course, the screen reader could provide a feature to virtualize the description but the presence of that feature will not be intuitively obvious and would require additional steps.
3. If the description is a DIV element that spans links, the user will have already experienced the links out of context and will be hearing them for a second time.

It is possible the describedby relationship is part of the answer, but by itself it does not provide a generally applicable means for creating usable access to static text.

Summary

Much is known about designing usable accessibility into software, but there remain substantial and long-standing fundamental gaps in our knowledge – including how to provide usable access to static text. We can make tremendous progress toward providing more usable access by utilizing UI design heuristics for usable access. But, as an industry, we also need to initiate a discussion to establish conventions or standards that will solve the accessibility and usability problems posed by static and other non-editable text that is mixed with interactive elements in a user interface.

Acknowledgements

Special thanks goes to the core team of the IBM 2008/2009 usable access working group for their dedication to development of usable access UI design heuristics, especially Mary Jo Mueller, Colin Crehan, Sueann Nichols, and Cathy Laws. I also appreciate the reviews of this paper provided by Rich Schwerdtfeger, Brian Cragun, Mary Jo Mueller, Cathy Laws, and Colin Crehan.

References

1. "WebAIM: EWG:Draft Jan 7." WebAIM: Web Accessibility In Mind. TEITAC Editorial Working Group. Mar. 2009 <http://webaim.org/teitac/wiki/EWG%7EDraft_Jan_7.php>.
2. Hagler, Keates, Ice, Kunzinger, Johannesen, Lovelace, Sacco, and Trewin. "Conducting user evaluations with people with disabilities." IBM Accessibility Center. Nov. 2005. IBM. 2009 <http://www-03.ibm.com/able/resources/userevaluations.html>.
3. "Accessible Rich Internet Applications (WAI-ARIA) 1.0." World Wide Web Consortium - Web Standards. W3C. Mar. 2009 <http://www.w3.org/WAI/PF/aria/>.
4. "WAI-ARIA Best Practices." World Wide Web Consortium - Web Standards. W3C. Mar. 2009 <http://www.w3.org/WAI/PF/aria-practices/>.
5. "Web Content Accessibility Guidelines (WCAG) 2.0." World Wide Web Consortium - Web Standards. W3C. Mar. 2009 <http://www.w3.org/TR/WCAG20/>.
6. "Understanding the Four Principles of Accessibility." World Wide Web Consortium - Web Standards. W3C. Mar. 2009 <http://www.w3.org/TR/UNDERSTANDING-WCAG20/intro.html#introduction-fourprincs-head>.
7. "IAccessible2." The Linux Foundation. The Linux Foundation. Mar. 2009 <http://www.linuxfoundation.org/en/Accessibility/IAccessible2>.

8. "Set of ARIA Test Cases - CodeTalks." CodeTalks. Mar. 2009 <http://wiki.codetalks.org/wiki/index.php/Set_of_ARIA_Test_Cases>.

Trademarks