Accessibility requirements for systems design to accommodate users with vision impairments

P. Brunet
B. A. Feigenbaum
K. Harris
C. Laws
R. Schwerdtfeger
L. Weiss

New technology tends to be inaccessible to people with disabilities. Accessibility features are frequently added as the technology matures; doing so late in the cycle incurs significant costs. The initial omission of accessibility features often results from systems designers' lack of insight into accessibility requirements. This paper discusses accessibility requirements for accommodating users with vision impairments from the complementary perspectives of the systems architect, the assistive technology developer, and the application developer. The paper concludes with a historical perspective of the evolution of the current Windows™ accessibility features and gives insight into future industry directions.

INTRODUCTION

An accessible information technology (IT) solution is one that is usable by all people, regardless of ability or disability. Although industry awareness of end user accessibility needs is increasing, inaccessible IT solutions continue to be delivered. Often, new technologies fail to address accessibility from the beginning of their development.

Typically, a platform is initially specified and delivered without any support for accessibility. In past years, this happened with IBM OS/2*, Microsoft Windows**, Java**, and the visual presentation systems for the Linux** platforms. In each case, accessibility support was added to the systems architecture as a remedial activity,[1] generally years after the initial releases. Addressing accessibility late in development is costly, and in the interim, the absence of a solid foundation contributes to interoperability issues between the platform, assistive technology (AT) products, and application software.

One root cause of inaccessibility is that software designers lack insight into the fundamental requirements of building accessible solutions. IBM is one of the few companies that have developed both accessibility infrastructures and assistive technologies on a broad range of platforms.[2] As past and

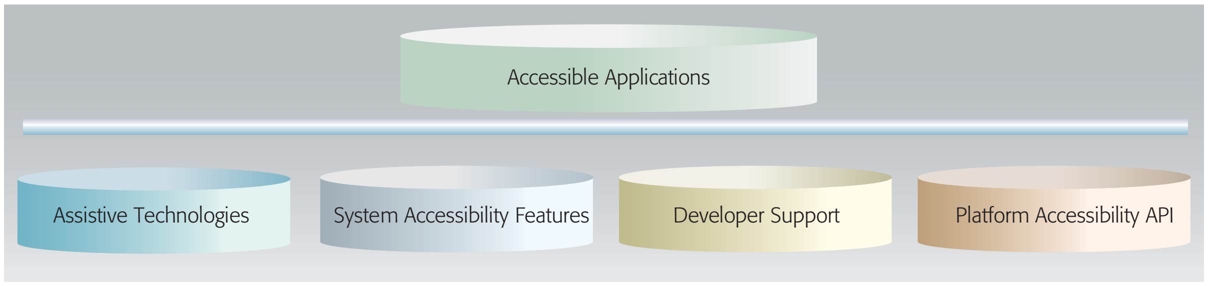


Figure 1
Software components of an accessibility solution

present members of the IBM Accessibility Center, the authors have a unique perspective on the requirements which the various IT communities must satisfy in order to deliver accessible IT solutions.

This paper presents requirements for accommodating people with visual disabilities and addresses non-visual user interfaces from several perspectives, including that of the systems architect, the assistive technology developer, and the application developer. We limit the scope of our discussion to the requirements driven by vision impairments in order to explore these requirements in greater depth. The graphical user interface (GUI) is the dominant human-computer interaction paradigm in today's IT environment. The expressiveness of the GUI is so rich, and the medium is so vision-oriented, that extraordinary efforts are required to translate this UI paradigm into other modalities.

COMPONENTS OF A COMPREHENSIVE ACCESSIBILITY SOLUTION

Accessibility solutions can best be discussed by dividing them into their components, which include the platform accessibility architecture, the assistive technology (including system accessibility features), developer tools support, and the application. *Figure 1* illustrates the various software components required to deliver an accessibility solution.

Platform accessibility architecture

In this paper, the term "platform" describes a software runtime environment in which a software application (made up of one or more executable modules) can run. A platform defines a unique set of application programming interfaces (APIs) and execution protocols (such as the application's behavior for handling an event). For example, the Microsoft Win32 API (available in Windows XP**) defines one platform. Java Standard Edition defines another platform, and Hypertext Markup Language (HTML), Cascading Style Sheets (CSS), and JavaScript** define yet other platforms. An application is usually built specifically for each platform on which it will execute.

Accessible platforms must provide a way for applications to export information about their visual user interface (UI) to assistive technology products, and for AT devices to observe state changes in the UI components. Additionally, they must provide device-independent access to applications. The programming interfaces and communication protocols associated with these facilities are generally known as the *platform accessibility architecture*. Platform architects carry the major responsibility for defining this architecture.

Application, assistive technology, and system accessibility features

In part, an accessible solution is created by enabling an application for accessibility during product design and development. This is analogous to enabling an application for internationalization.

When an accessible application is deployed to people with disabilities, the product is paired with a complementary assistive technology, which provides alternative input and output mechanisms, to create a complete solution. Examples of assistive technology include screen readers that use text-to-speech (TTS) engines to read software to people who are blind, closed captioning displays for people who are deaf or hard of hearing, and special keyboards and input devices for people with limited hand use or mobility impairments.

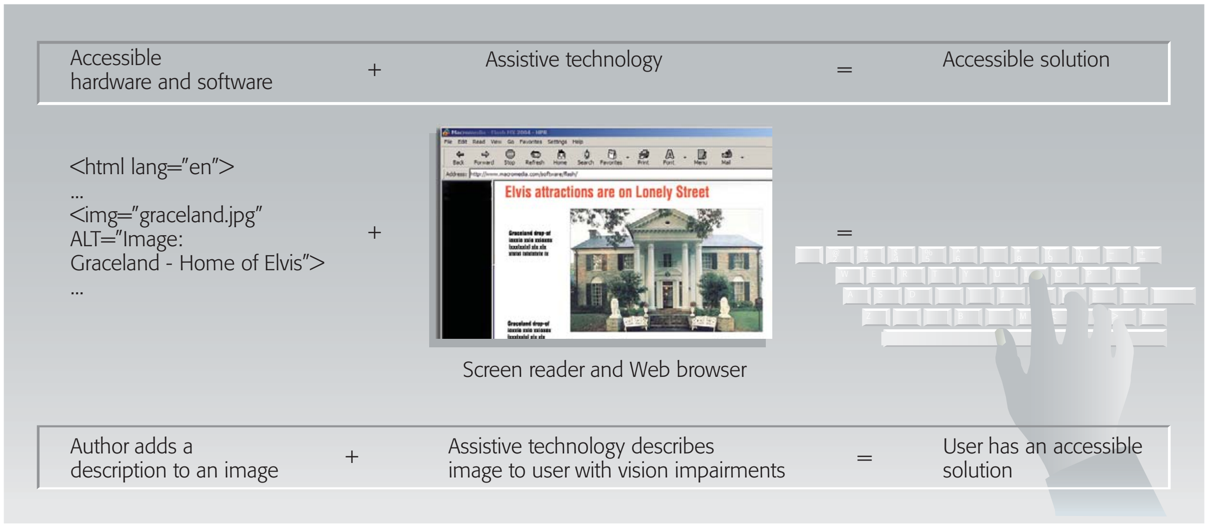

Accessible hardware and software
+
Assistive technology
=
Accessible solution
<html lang="en">
...
<img="graceland.jpg"
ALT="Image:
Graceland - Home of Elvis">
...
+
Elvis attractions are on Lonely Street
=
Screen reader and Web browser
Author adds a description to an image
+
Assistive technology describes image to user with vision impairments
=
User has an accessible solution

Figure 2
Accessibility solution for a Web page

An AT can only be effective when the software and hardware with which it interfaces is accessible. For example, a screen reader cannot read informational graphic images on a Web page unless the Web page author provides an alternative text (`alt`) attribute for those images. If the `alt` attribute is missing, the screen reader cannot provide meaningful information about the image. ***Figure 2*** illustrates this point—the `alt` attribute in the `IMG` image tag enables the AT to make the Web page accessible.

System accessibility features, such as Windows' StickyKeys and high contrast color schemes, are a special class of assistive technologies which a platform must provide or provide access to, and with which an application must be compatible. For example, the Mozilla** browser and Java Swing**, which run on multiple operating-system platforms, both have provisions for responding to system font and color settings. Swing has a pluggable "look and feel" for Windows that allows standard Swing UI components to respect Windows font and color settings. Although early versions of the Java platform did not, the platform now supports StickyKeys for Java applications running on Windows.

Developer support

Enabling applications for accessibility can often be both a tedious and error-prone activity. Developers can easily overlook small details that need to be addressed. As a result, support is needed to assist the developer in producing an accessible solution. Specifically, developers need guidelines, reusable accessible controls, and authoring tools with sample code. For example, Web developers need tools to create accessible HTML and other Web content. The World Wide Web Consortium's[3] (W3C**) Web Accessibility Initiative[4] (WAI) authoring-tools working group[5] created the Authoring Tools Accessibility Guidelines (ATAG) recommendation to define what a Web content creation tool must do to promote the creation of accessible Web content. Developers need similar tools to assist in the creation of accessible rich-client GUIs, such as GUIs built using Java Swing[6] and the Eclipse Standard Widget Toolkit (SWT).[7] For example, the IBM Reflexive Interface Builder advanced technology[8] provides an accessibility validator feature that detects potentially inaccessible GUIs built with either Swing or SWT.

Prior and related work

Extensive work has been done in a number of domains that relate to the subject matter of this paper. Programming requirements and guidelines for building accessible applications are published by a number of organizations. The Trace Center at the University of Wisconsin maintains a list of well-written, representative publications at their Web site.[9] IBM's approach to capturing and enumerating such guidelines can be found at http://

www-3.ibm.com/able/guidelines/index.html. Jacob Nielsen has written guidelines specifically for Web-based applications in Reference 10.

The insights captured in these guidelines are critical to the successful delivery of an accessible application. However, these guidelines generally rely on underlying platform features which must be provided by the systems architects. Within this paper, we have attempted to capture the fundamental principles and features that the platform and AT designers must address.

Accessibility is one component of the broader domain of "universal usability," which has been described as "a focus on designing products so that they are usable by the widest range of people, operating in the widest range of situations, as is commercially practical."[11] Many researchers in this field have worked to capture requirements, as found in Reference 11 and Reference 12. These works have captured fundamental requirements from an end user perspective, but provide less guidance on software architectural constructs that must be built into the operating system (or equivalent programming platform such as the Java API) as a foundation for the needed user interface features.

We believe that this paper is unique in treating these fundamental principles from the systems programming perspective. While there are works which discuss system API sets or features within the domain of a specific platform (such as HTML or a single operating system such as Linux[13]), we are not aware of recent work which assesses the fundamental software architectural requirements that must be consistently met, regardless of the specific programming platform used.

THE NEED TO ACCESS SEMANTIC INFORMATION

Visual user interfaces have traditionally focused on how information is presented on a display, with little regard for making the underlying information of an application available through alternative programming interfaces. Consider a GUI that presents an image of a bar chart. In that image, there is no information about the meaning of the various bars in the chart and their values; the image is just an array of pixels. The term *semantic information* refers to the underlying information in an application, as opposed to the presentation of that information. It is essential to maintain semantic information separately from the visual presentation and to make the semantic information accessible through programming interfaces.[11,12,14]

Capturing semantic information may be easier when only character I/O technology is employed. Character I/O consists of plaintext content, often limited in scope (such as an 80×25 grid). Examples include the command-line interface provided by most operating systems, where old content scrolls off the screen. In this environment, the AT is frequently able to directly observe the encoded character data. If the application consists simply of prompts and responses, all the semantics of the interface are encoded in these text strings. The AT does not need to perform any additional interpretation of this information; rather, the text strings can simply be presented through an alternate physical interface such as a Braille device.

User interfaces that exploit a graphical paradigm are not as easy to deal with. The physical medium for these interfaces consists of a grid of pixels (often quite large, such as 1024×768) on which content is drawn. Text content is rasterized into a small grid of pixels, but other content, such as images, is displayed directly. No restrictions are imposed on font size or content position. Unlike character I/O interfaces, the encoding of the character strings may not be available to the AT. In this environment, an assistive technology must capture information drawn on the screen and attempt to reverse-engineer the text and semantics[15] if that facility is available.

GUIs (or character I/O interfaces which emulate a graphical paradigm) can be quite complex. In addition to traditional components, such as images, buttons, lists, tables, and trees, GUIs now incorporate multimedia content, such as sounds, music, videos, and conference calls. Assistive technologies must be able to present this content in a form acceptable to the user, regardless of the user's abilities. This places a higher burden on the application and content developer to provide additional semantic information about the content and its presentation and on the accessibility framework to make it available through programming interfaces.

Often assistive technologies have difficulty in presenting a high-fidelity alternate user interface. Although many reasons exist for this, the most significant one is that the GUI focuses on how it must present information rather than on exposing the appropriate content and semantics to an assistive technology that provides alternative renderings.

In our initial bar-chart example, the graphic provided no information about the meaning of the various bars in the chart; the image was simply an array of pixels. The chart was probably created from information in a spreadsheet or database table. If the assistive technology had access to this spreadsheet or database, rather than only the image generated from it, the assistive technology could provide an alternative rendering of the data appropriate for the accessibility needs of the user.

State-of-the-art design for accessibility today is based on a Model-View-Controller (MVC) design pattern[16] that allows developers to store valuable semantic meta-data without a UI dependency and that lays the foundation for exposing this information to assistive technologies. In the next section, we review the MVC design pattern and explore its applicability to the design of accessible solutions.

The MVC design pattern

The MVC architecture is a classic approach for a variety of interactive applications. It separates the tasks of maintaining an application's internal model from the tasks of presenting a user interface and processing user input and output.

The MVC design pattern partitions an application in the following way. The *model* represents the application data, the state of that data, and the operations that permit this application information to be accessed and changed. The *view* renders the contents of the model. Traditionally, the view is the GUI representation of the application. The *controller* handles events generated when the model changes, such as when user input from the mouse or keyboard modifies the data (i.e., by entering new data) or changes state (i.e., by changing focus from one control to another). ***Figure 3*** depicts the three parts of the MVC design pattern and the interactions among them.

As seen in Figure 3, the model encapsulates the application state, responds to state queries, exposes application functionality through programming interfaces, and notifies the appropriate views when changes occur. The view component renders a presentation for the model, requests updates from the model, sends user input events to the controller, and allows the controller to select views. The controller component defines the application behavior, maps user actions to model updates, and selects views.

The view accesses the model to obtain the information needed to render the model's contents. For example, for a word-processing application, the view obtains text strings and style information from the model; when the model changes, the view renders the appropriate visual text. This design pattern maintains a careful separation of content from presentation.

The controller must update the model when user input events result in some state change in the model. Following the word-processing example, when the controller captures keystroke events, it might insert new text characters into the model of the document being edited, triggering the view to visually render the change in content.

MVC in the context of assistive technology

The applicability of the MVC design pattern to the assistive technology interoperability problem is straightforward: an AT must create alternative views for the application's model. Reference 12 describes the construction of "dual user interface" applications, in which a single authoring tool is used to capture the semantics of the interface and then generates multiple user interfaces. Each generated UI is optimized for the abilities of the target user community. The dual user interface is an excellent illustration of the utility of the MVC-based approach. More generally, Reference 17 describes the desirability of "multilayer" UI design, in which a collection of view-controllers of varying complexity and functionality levels is built for a single application.

Sometimes these new views are very tightly bound to the original visual rendering of the application on the computer display. For example, a screen magnifier such as ZoomText**[18] builds its unique view by enlarging the contents of a subsection of the display. In other cases, however, there may be a looser binding between the view created by the AT and the original visual rendering of an application's

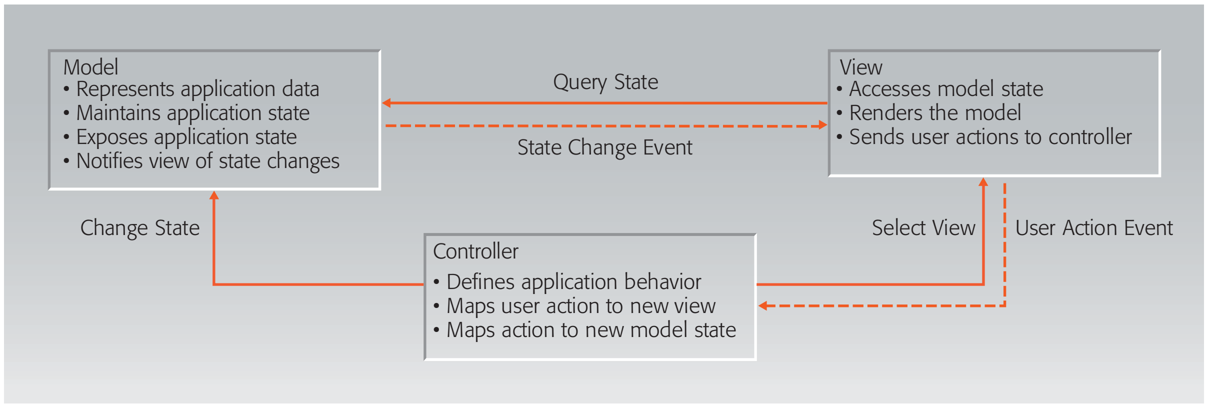

Model
• Represents application data
• Maintains application state
• Exposes application state
• Notifies view of state changes
Query State
State Change Event
View
• Accesses model state
• Renders the model
• Sends user actions to controller
Change State
Select View
User Action Event
Controller
• Defines application behavior
• Maps user action to new view
• Maps action to new model state

Figure 3
The MVC design pattern

model. IBM Home Page Reader[19] (HPR), a talking Web browser, illustrates the latter approach.

HPR provides the traditional graphical rendering of the Web page for sighted users by embedding the Microsoft Internet Explorer** (IE) Web browser control.[20] HPR supplements the graphical view of the Web page with synthesized speech to deliver aural renderings of the page, which include information not explicitly rendered in IE's graphical view. Such extra information includes row and column numbers within a table and announcements of the top and bottom of a form.

The AT, in addition to generating new views appropriate for people with disabilities, must in many cases deliver a new controller for the application. The controller component of HPR, for example, captures all keystroke and mouse user-input events and redefines the handling of many of these events. For example, a mouse click on a paragraph of text causes HPR to speak that text and reposition the speech cursor to that location.

REQUIREMENTS FOR A PLATFORM ACCESSIBILITY ARCHITECTURE

We have argued that an application's semantic information must be maintained separately from the presentation of that data. An AT can then create alternative views for the application's model. The need for new views and controllers aligns nicely with the widely accepted MVC design pattern, and may present opportunities to exploit earlier investments in MVC architectures. These new views and controllers are uniquely designed to meet the needs of people with disabilities. With this in mind, we present the first requirement for a platform accessibility architecture; that is, that an IT application must be structured in such a way that the application model, views of that model, and control functions which modify the state of the model, are well isolated from one another.

The platform accessibility architecture defines a standardized API, or "contract," between the application (which maintains the model) and the assistive technology (which provides new views and controllers). The architecture defines the scope and the semantics of the information exchange. Consequently, the architecture ultimately limits the expressiveness, or the "richness," of the user interfaces that can be built for end users with disabilities. This section presents our view of the architectural semantics that must be supported by any platform accessibility architecture. Later, in the section "Assistive technology requirements," we present the features required in the AT and focus on the needs of alternate view-controllers. See "Summary of accessibility requirements for applications and tools" in the Appendix for a discussion of the issues concerning an application's interaction with the platform accessibility architecture.

The platform requirements mostly concern isolating and communicating information about the application model. A robust object model containing the necessary semantics must be maintained by the application and communicated through the platform accessibility API. We present some representative object model semantics which are common to many applications. We discuss the requirements that the platform architecture must provide for describing the relationships among the objects in the model and explain the need for an event system to communicate user input and changes in the model.

A challenging trade-off exists in making object-model semantics both generic and expressive, as each of these objectives works against the other. In the following subsection, we explore some approaches toward resolving this conflict.

Object model

The platform accessibility architecture must enable programmatic access to an object model for an application through a well-understood contract. Assistive technologies use this feature to access the semantic information in an application.

There is often more than one object model that might be useful to an assistive technology. Screen readers typically build richer functionality by integrating the information obtained from several models. The application model, from a pure MVC perspective, consists of the data which makes up the application and the operations that modify it. Before standardized contracts, or accessibility APIs, were created, a model needed to be reproduced by capturing the necessary data before it was rendered on the screen.

Off-screen model

Before the advent of GUIs, visual user interfaces were based on character I/O. Typically, the currently running program—a single application—controlled the entire display device. A screen reader for the MS-DOS[21] platform had two designs available for reading the displayed text for alternate presentation to the user. The first design intercepted the DOS interrupts that drew text on the screen. The second design polled the text buffer for changes.

The first design is closer to the modern approach, but was invalidated when character I/O applications circumvented the DOS API by writing directly to the text screen buffer. By polling the screen buffer, the second design indirectly introduced the concept of a "model" of the application contents. Within the MVC framework just described, the text screen buffer provided the point of interface between the application model and its view (the text and its attributes). IBM Screen Reader* for DOS,[22] an example of the second design, used the screen buffer as a model of the application's content.

The modern GUI renders text and attributes in bit patterns that do not contain the actual characters. As a result, an AT for a GUI must rely on an alternate model of the displayed content. Because the first GUIs provided no standard model of the content, ATs created their own model by intercepting text-drawing calls and then caching the merged text in an off-screen model (OSM).[15] IBM Screen Reader/2 for OS/2 was one of the first ATs to create and use an OSM.

The OSM technique continues to be widely used by assistive technology vendors in the current Microsoft Windows environment. Among other advantages, the OSM enables the AT to present and interpret spatial relationships among most visual user interface objects, allowing the AT to present a richer view to the non-visual end user.

However, the OSM is also problematic. The AT must infer application semantics from information that is intended for visual display. This is error-prone: for example, the juxtaposition of two objects on the visual display does not ensure that the two objects are related. Furthermore, AT vendors will not be able to create OSMs in newer operating systems with more robust security features and sophisticated vector graphics technology. In the absence of an OSM, systems architectures must still provide the equivalent spatial information and other visual attribute information to an AT.

Generic model or accessibility API

Rather than infer the semantics of an application from its OSM, it would be better to build new views directly from the application's internal model. An assistive technology benefits when a single, standard object model can be used to access all applications and all user interface objects that might appear on the visual display. Platform architects have defined a number of important architectures to address this need: Microsoft Active Accessibility**[23] (MSAA), which is available on Microsoft Windows desktop environments ranging from Windows 98 through Windows XP, the Java Accessibility API[24] (JAAPI), available in the Java SDK (see ***Figure 4*** for a graphic depiction of this API); and the Accessibility Toolkit API[25] (ATK), a component of the

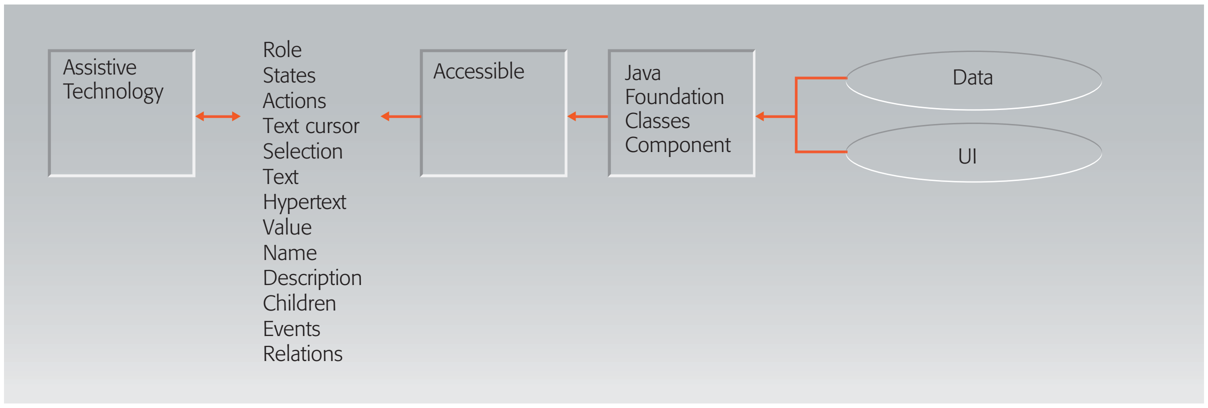


Figure 4
Java Accessibility API

Gnome Accessibility Project[26] (GAP), available in Gnome Version 2.4 and later.

Each of these architectures represents user interface objects as nodes in a tree structure, in the same way as an XML (Extensible Markup Language) document represents the information in the document as a tree of nodes. The customary set of navigation methods are provided so that assistive technologies can traverse (or "walk") the tree to the top object and recursively enumerate the children. Without methods to access parent and child objects, it is impossible for an AT to access all the objects and their accessibility information in an application, and it is difficult for an AT to provide logical navigation of the document structure.

The semantics in each of these architectures defines a discrete number of UI types, attributes, and states for each node, which usually corresponds to UI objects in the application. For example, each framework defines object types, called *roles*, such as "button," "list box," and "text," and attributes such as "name," "value," "parent," "child," and "screen location."

The ability to access any application through a small, predefined set of object definitions simplifies the AT's work. However, this standardization comes at a cost. The industry continually creates new, custom types of GUI objects. Whereas the simple, standardized accessibility API promotes interoperability with well-known UI objects, it is frequently difficult to build an adequate non-visual view of a custom GUI object. The standard semantics provided by an accessibility architecture frequently lack the expressiveness needed to describe new, innovative UI objects or document structure. In such cases, the richness of the user experience is lost in exchange for the simplicity of a standard interface.

For example, to obtain a rich interface for an ActiveX** GUI control in the current Microsoft Windows environment, AT developers usually circumvent the accessibility architecture and directly access the ActiveX control's methods. Access to the particular methods of a specific UI object may allow the AT to provide greater functionality, but using many different API sets to access a wide range of application components is a development burden. Doing so requires a specific development effort on the part of each AT vendor. If this specific investment is not made by an AT vendor, an interoperability issue results, and the new GUI object is inaccessible. In high security environments, it may not be possible to access the object's methods.

We explore one strategy toward enabling the extension of the generic model in the section "Generic model extensibility."

Application-specific Document Object Model

There is a significant industry trend toward representing application data in XML accessed through a parsed Document Object Model (DOM). The DOM API provides the contract to access the information. The "schema" for the object model is tailored for each specific application. For example, Microsoft Word has a different object model from a Web service or Web page.

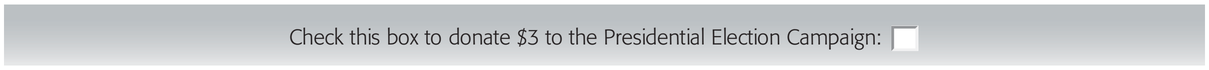

Check this box to donate $3 to the Presidential Election Campaign:

Figure 5
A simple label/object relationship

Because the object model is designed for a specific application, an AT with access to this model may often build a richer view than those which are available only through the platform architecture's generic model. Unfortunately, requiring the AT to develop application- and DOM-specific algorithms for a given class of applications is costly. For example, to present a usable Microsoft Word interface for blind users, the AT must develop specific algorithms that understand the semantics and schema of the Word DOM.

Support for device-independent actions

An application object model must provide an accessibility API through which an AT can obtain the set of actions and descriptions for each object in the application and make all actions available through programmatic access. The AT can convey this information to the user so the full range of features in an application can be used. By providing this programmatically through a predefined accessibility API, an assistive technology could also invoke any function by using alternative input mechanisms rather than having to simulate a set of key strokes or mouse clicks to perform the same task. The Java and Gnome accessibility APIs provide this capability through their AccessibleActions interface.

For example, a Web-based application might attach several JavaScript handlers to a single UI object, associating each handler with a unique mouse event (such as "mouseOver" and "mouseClick"). In order to make these features accessible, the AT must be able to interrogate the application as to what each handler does and be able to programmatically invoke each handler.

Relationships among user interface objects

An object model of an application's user interface by itself does not provide enough information for an AT to build an adequate view of an application. The AT also must be able to determine the relationships among the user interface objects.

As a simple illustration of this requirement, consider the association of an input object, such as a checkbox, with its text label, as shown in ***Figure 5***. The purpose of the checkbox is obvious in the visual presentation because the text and checkbox are displayed next to each other. In the case of more complex collections of elements, where an object controls one or more other objects, such as the formula field of a spreadsheet that controls multiple cells, the associations may not be readily seen. This potential source of end-user confusion is resolved when the application specifies the explicit associations using facilities in the platform accessibility architecture.

The ability to describe spatial relationships among objects is important for similar reasons. Any experienced software developer appreciates the value of correct indentation for a block of computer code. When a user of a visual interface wishes to collaborate with a user of a non-visual interface, it is helpful if the two users can refer to the location of one object with respect to another object, for example, "I'm interested in object B, in the upper-right corner of the display." Finally, assistive technologies may also interpret implied associations among objects, such as labels for user inputs, based on relative spatial positioning.

The ability to specify sophisticated relationships among objects is a key challenge in building accessible IT solutions. The industry has made significant progress in IT accessibility for applications such as online forms, in which simple text and user inputs are the only UI elements. However, many of the more knowledge-intensive applications use maps, graphs, and drawings as the user interface paradigm.

In light of these challenges, it is clear that a robust facility for specifying the relationships among objects is a core requirement for any platform accessibility architecture.

Events

We have considered the requirements for an object model of the user interface elements and for the relationships among those elements. The final requirement for any platform accessibility architecture is the specification of an *event protocol* and *event semantics*, through which the AT is informed of state changes in the current application model, and of user inputs.

For example, in an online form on a Web page, if a user makes an error, the form validation algorithm typically presents an error message and then sets the keyboard focus to the form field containing the incorrectly entered data. Ideally, the platform accessibility architecture should broadcast events to provide notifications of error messages in addition to the new location of the application's keyboard focus.

In addition to events triggered by state changes in the UI model, the platform accessibility architecture must supply the AT with one or more features which permit it to monitor and modify user input events. The keyboard is a particularly critical input device; it is thus essential that the platform accessibility architecture allow the AT to intercept and modify the stream of events generated by the keyboard.

An AT usually defines many unique *key sequences* that can be entered by the user and intercepted and interpreted by the AT without being delivered to the application. For example, these key sequences may provide access to application features that are usually available only by use of the mouse. AT key sequences also provide the user with the ability to invoke AT-specific features, such as reviewing information displayed on the screen, obtaining additional information about the current object with focus, and accessing and changing the AT's own features, help, and settings.

The Windows operating system (up to and including Windows XP) satisfies this requirement by providing a system-level API that allows an application to intercept all keystroke events.

Generic model extensibility

The section "Generic model or accessibility API" introduced the problem of the limited expressiveness available in a generic object model. In this section, we explore approaches to extending a generic model. We first consider only the "role" attribute, which is one of the key standard attributes found in all current accessibility architectures. Roles, which are associated with visual UI components, allow an AT to determine the function, behavior, and alternate presentation for the UI component. Today's operating-system accessibility infrastructures assign roles statically and do not address "role extension" very well. Adding new roles requires updating and redistributing the infrastructure, making it very difficult for groups other than the operating-system development team to create new UI components with accessible information.

Microsoft, in its upcoming Longhorn operating system,[27] has taken steps to improve this situation by providing a set of behaviors called UI Automation[28] Control Patterns, which any UI object can support. Although this model is more flexible than those using predetermined roles, it still uses a static set of known behaviors.

A more complete approach to creating an extensible model is under development, as part of the work on the semantic Web. The semantic Web (an extension of the current Web in which information is given well-defined meaning) provides a common framework that allows data to be shared and reused across application, enterprise, and community boundaries. It is a collaborative effort led by the W3C with participation from a large number of researchers and industrial partners. It is based on the Resource Description Framework (RDF), which integrates a variety of applications using XML for syntax and URIs (uniform resource identifiers) for naming.[29]

RDF provides a rich semantic language to describe the function of a UI component. The AT can interpret this language and provide an alternate UI for the component if the language defined is rich and pervasive enough. The industry has started to provide this definition through the semantic Web—the first real opportunity to provide semantics through a semantic-extensible framework.

RDF can be used to create a schema (describing the properties, semantics, and types of resources) for taxonomies that specifies how to represent objects and their relationships. These taxonomies can describe a previously unknown object, identify ancestor objects from which the new object inherits its behaviors, how it interacts, what states it can achieve, and the events that it generates.

All of these semantics allow an AT to discover a semantic interaction model and create a knowledge base for a previously unknown object. The AT can use this knowledge base to determine how to interact with the new object (or "widget"). More important, middleware can use this information to better adapt Web content to different devices, users, and environments. Reference 14 describes one application that uses RDF-encoded semantics to construct multiple views of Web pages for a diverse user community.

Today, developers provide speech interfaces (e.g., voice recognition commands) for commercial applications that are not specifically created for people with disabilities. If knowledge of how the objects were used to create a user interface were available, it would be possible to create speech interfaces directly for users who are blind or who have mobility impairments.

An upcoming XHTML 2 (Extensible Hypertext Markup Language Version 2) meta-specification module[30] will include a new attribute called `role`, which will define the content type of the object, regardless of the element with which it is associated. In essence, it allows the creation of an extensible object model that can be queried to determine how to interact with the targeted element.

REQUIREMENTS FOR AN ASSISTIVE TECHNOLOGY

Developing an assistive technology, such as a screen reader or magnifier for users with visual impairments, requires creating one or more new view-controllers for applications and perhaps for the platform itself. The views must provide alternatives that allow a person with a disability to interact with all the features and functions of the platform and applications, allowing him or her to be as productive and efficient as a nondisabled peer.

To meet this objective, an assistive technology developer must consider and accommodate alternative-input navigation to all features and content, rendering of all information with alternative output, customization through settings and scripting, performance, and compatibility and interoperability.

Navigation to all features and content

The platform must make all content accessible through an object model or a programming interface. The AT must provide navigation to all features and content using at least one alternative mode of input other than a pointing device. The alternative input method depends on the type of disability for which the AT is being created and the type of device. The primary alternative input for navigation for most users with a disability, especially for users with visual impairments, is the keyboard, but voice recognition or single switch input with an on-screen keyboard is sometimes a better choice for users with mobility impairments.

It is surprisingly challenging to design a user interface which is purely keyboard-driven that enables a screen reader to access all features and content. An initial, overly simple strategy might rely on following the system focus for keyboard input. Usually, this approach provides navigation only to elements that can be activated or edited, such as links, controls, and documents. This navigation strategy skips the content, called static text, which does not normally receive keyboard input focus.

The navigation strategy delivered by the AT must provide access to static text. Doing so usually requires the AT to define and maintain at least one additional "virtual" cursor to track current position within static text. This feature is often referred to as a "content focus" or "point of regard." In addition, the AT must define additional keystroke entry commands to control the point of regard. Because most simple keystroke sequences are used by some application, it can be difficult to identify keystroke combinations that do not conflict with other features in the platform or application. An example of such an AT is the JAWS** for Windows[31] product. JAWS (Job Access with Speech) is a general-purpose screen reader that reads most Windows applications. It defines several cursor modes. One of these modes, called the "virtual PC cursor," lets the user access and review any of the static text currently displayed within the application's graphical window, regardless of any input fields that might currently have keyboard input focus. JAWS uses unique key combinations—some quite complex—to avoid key conflicts with the system and application keys. Instead of following keyboard focus, HPR, which is a foreground Web browser application, provides its own reading-mode key sequences.

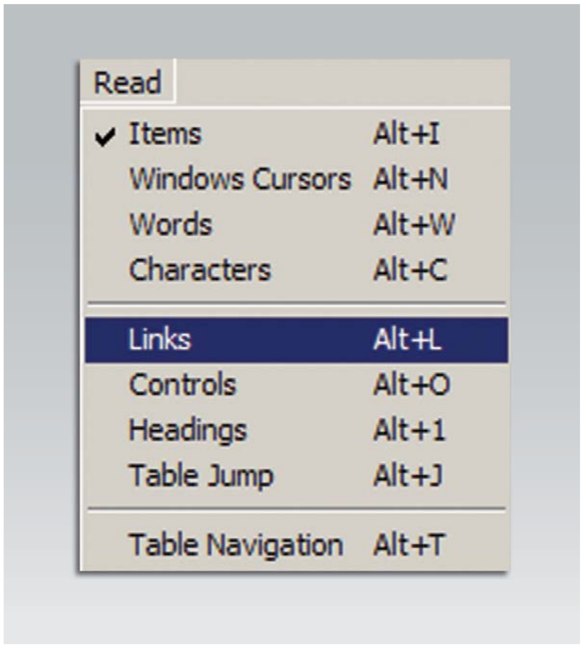


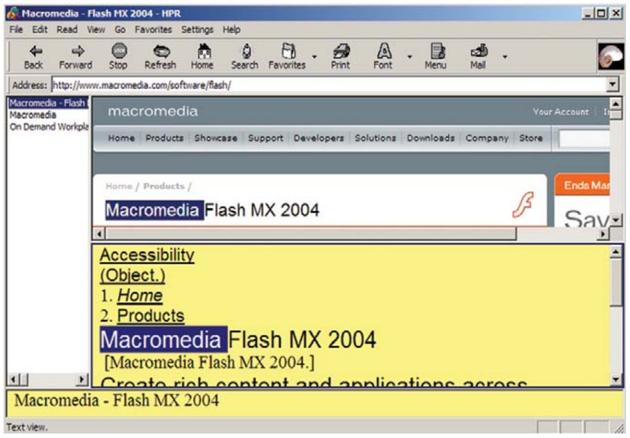
Figure 6
IBM Home Page Reader reading modes

In contrast to general-purpose screen readers, HPR is a single application that delivers a multimodal user interface for Web browsing. HPR maintains just one cursor mode for this limited application domain, simply ignoring the system keyboard input focus and maintaining its own point of regard for the user. HPR also defines its own navigation and command key sequences, as follows:

Basic sequences
Stop (Ctrl or Esc)
Read page (Spacebar)
Next/previous link (Tab/Shift + Tab)
Activate link/control (Enter)
Next word (Ctrl + Right Arrow)
Previous word (Ctrl +Left Arrow)
First/Last item (Ctrl + Home/End)
Up/down 10 items (Page Up/Page Down)
Begin/end of entity (Home/End)

Sequence for items and most reading modes
Read previous, current, next entity (Left Arrow, Down Arrow, Right Arrow)

Sequence for the Windows Cursors reading mode
Previous character (Left Arrow)
Next character (Right Arrow)
One line up/down (Up/Down arrows)


Macromedia - Flash MX 2004 - HPR
File Edit Read View Go Favorites Settings Help
Back Forward Stop Refresh Home Search Favorites Print Font Menu Mail
Address: http://www.macromedia.com/software/flash/
macromedia
Your Account
Home Products Showcase Support Developers Solutions Downloads Company Store
Home / Products /
Macromedia Flash MX 2004
Ends Mar
Sav
Text view.

Accessibility
(Object.)
1. Home
2. Products
Macromedia Flash MX 2004
[Macromedia Flash MX 2004.]
Macromedia - Flash MX 2004

Figure 7
Text view of Home Page Reader

The keystroke sequences are simpler than those required for a general-purpose screen reader; this is possible because the problem domain is much smaller. ***Figure 6*** shows some of the items that the user can direct HPR to read.

Rendering of all information

When meaningful information is not available through the platform architecture or when developers create custom controls, AT developers must develop creative, but non-error-prone heuristics for finding and producing the object information they need to present to the user. Assistive technologies today use an "object model plus" system and APIs to ask for both visual and structural information about the content being displayed. Once the AT has the information for which it is looking, it builds text strings to deliver to alternative output devices. These devices, which are now more sophisticated, include text views with highlighting, color contrast, and magnification; software TTS engines; and Braille displays with a range of Braille character output and navigational input features. ***Figure 7*** shows the text view of HPR.

As an example of both standard API usage and heuristics, Home Page Reader takes advantage of the many HTML 4.01 accessibility features and implied HTML tag semantics that are available through the Microsoft Internet Explorer DOM interfaces, such as the LABEL element with the `for` attribute to explicitly label form controls, the `headers` attribute to provide headers for table cells, and the `alt` attribute to describe images and map areas. In cases where the Web author provides no accessible information, or where the HTML standards provide insufficient ways for developers to supply accessible information, HPR and other assistive technologies resort to heuristics, such as looking for text that is nearby but not explicitly associated with a control as a label for that control.[32]

Customization through settings and scripting

ATs present application content that is normally displayed to the end user in an alternate form to make the application accessible. Because no two users have exactly the same abilities, experience, or style of working, the AT must offer mechanisms such as settings and scripting to customize the presented content, as described next.

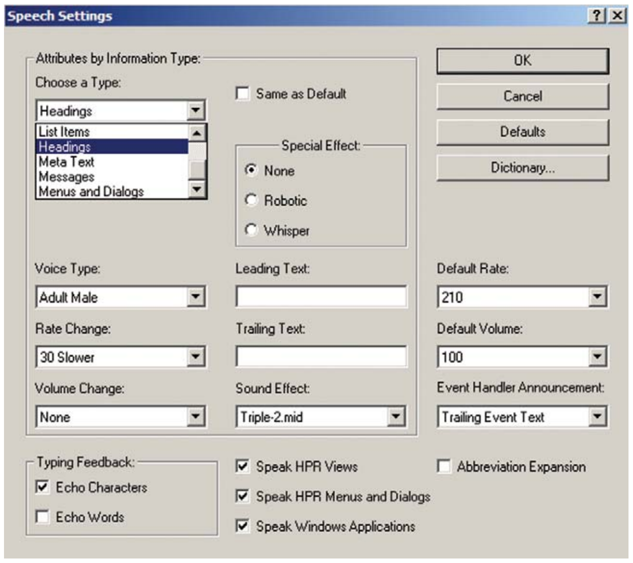
Settings

As a prime example of customization through settings, AT products that use TTS engines to announce visual content take advantage of TTS engine features, such as different speech rates, voices, multiple languages, and dictionaries, to distinguish different types of information and increase understanding and productivity. Customization through settings can be helpful but also more complex for the end user, as illustrated in the Speech Settings dialog for Home Page Reader (see *Figure 8*), which uses the IBM TTS engine.[33]

Scripting

When an AT works with more than a single type of application, it needs to provide a more powerful mechanism than settings to customize the output. General purpose ATs, like JAWS for Windows and the Java Self-Voicing Development Kit[34] (SVDK), must provide both a settings interface and a scripting language.

Scripting languages can be unique to the AT, like that used in JAWS for Windows, or they can be in the form of a library of specialized functions in a standard language, as in the Java SVDK. Ideally, as in both of these ATs, the custom script can apply to all applications; alternatively, they can apply to a single application. The scripting language should provide a means to define new key sequences and to respond to application events like focus changes and text cursor movement. Furthermore, it should provide a way to query information about the application like the contents of selected text or the state of a control.



Attributes by Information Type:
Choose a Type: Headings
List Items
Headings
Meta Text
Messages
Menus and Dialogs
☐ Same as Default
Special Effect: [✓ None] Robotic Whisper
Voice Type: Adult Male
Leading Text:
Default Rate: 210
Rate Change: 30 Slower
Trailing Text:
Default Volume: 100
Volume Change: None
Sound Effect: Triple-2.mid
Event Handler Announcement: Trailing Event Text
Typing Feedback:
✓ Echo Characters
☐ Echo Words
✓ Speak HPR Views
✓ Speak HPR Menus and Dialogs
✓ Speak Windows Applications
☐ Abbreviation Expansion

Figure 8
Speech Settings dialog for Home Page Reader

Performance

A major concern for users of an assistive technology is system responsiveness to end-user requests. Because an AT user must navigate and process much more textual information and use slower alternative input and output mechanisms, system responsiveness and quick rendering of output, such as text spoken by TTS engines, is essential for end-user productivity. When a screen reader encounters a complex UI, the responsiveness of the speech output can dramatically degrade due to the computing resources used by the AT to process a large volume of information.

To improve performance, ATs may buffer large amounts of content. For example, if a user requests to read an entire Web page, Home Page Reader collects and sends just a small unit (or "chunk") of information to the speech engine to initiate speech instead of making the speech engine wait for HPR to gather all Web page information. While the engine is speaking the first chunk of information, HPR repeats this process until all information has been collected and sent.

Still another performance trade-off consideration is the quality of the TTS engine. Most English and European assistive technologies use a formant TTS engine for speech output, that is, one which uses resonances of the human vocal track. The engine uses a rules-based synthesis process that produces a voice that sounds mechanical. This speech engine has minimal system requirements for memory and disk space and offers a broad range of voices, languages, speech rates, and other speech characteristics. Specifically, it is very quick to start and stop speaking, and due to its strict rules-based output, it is consistently intelligible to experienced listeners, even at high speech rates.

Used mostly for telephony applications that run on a server, a "concatenative" TTS engine provides natural-sounding synthesized speech that is produced by putting together sound bits from a library created from professionally recorded sentences and phrases. This type of speech engine requires significantly larger amounts of memory and disk space than does formant speech and offers a smaller range of speech characteristics than a formant speech engine. In addition, its limitations include a slower maximum speech rate, using only one language engine at a time and only one or two voices. It also has greater system requirements to achieve start and stop responsiveness comparable to formant TTS engines.

Event handling can be handled in an *in-process* or *out-of-process* manner, and this design choice can affect AT performance. In-process assistive technologies, such as Home Page Reader, operate in the same operating-system process as the GUI that they are servicing. Out-of-process assistive technologies, such as any MSAA client like the JAWS screen reader, operate in a different operating-system process than the application they are serving. Although Windows ATs have improved performance by finding mechanisms to access MSAA in an in-process manner, this is not the case on other systems. The overhead incurred in crossing process boundaries has a significant impact on the design and performance of the accessibility APIs.

Compatibility and interoperability

Compatibility and interoperability between assistive technologies and the applications they support present significant problems. Many assistive technologies are competing for the same system resources, accessing the same APIs and events, and often trying to speak the same information using different TTS engines. In addition, the AT may only work well with a selected number of applications with which it may nevertheless have keyboard and feature conflicts.

To handle compatibility problems, ATs must offer settings and key sequences to silence, modify, or turn off their speech, keys, or other features on demand or when a specified application has focus.

An AT should conform to standards to minimize interoperability problems. Due to problems with keyboard conflicts, ATs often do not follow platform user-interface design standards or comply with software accessibility guidelines like Section 508[35] and the W3C User Agent Accessibility Guidelines[36] (UAAG).

ACCESSIBILITY REQUIREMENTS FOR APPLICATIONS AND TOOLS

The accessible object model, which is made available through the platform accessibility architecture, is useless without the active participation of the application. An application must make its information available through the model so that an AT can present it. Enabling applications for accessibility and verifying that enablement is a significant task. Enablement challenges include the lack of knowledge about how to develop accessible user interfaces, the cost of developing the user interface, and the costs to repair an incorrectly constructed user interface.

Developing an accessible user interface without accessibility enablement tools is difficult and error-prone. Tools help to encapsulate knowledge of good accessibility design and implementation. Their use reduces errors and can automate accessibility enablement, thus reducing application cost and time to market. Based on our experience, we believe that the single most important factor in a large organization's ability to consistently deliver accessible solutions is the availability of robust accessibility enablement tools.

Often, the technical difficulties of developing an accessible user interface depend on the underlying accessibility architecture of the platform. IBM primarily focuses on these platforms: Microsoft Windows,[23] UNIX**, Linux using the Gnome GUI toolkit and desktop,[26] the Web,[37] Java Swing,[24] and Eclipse and Eclipse-based environments.[38]

Within IBM, Eclipse is now supplanting Windows and Swing as the primary rich-client development environment. Eclipse offers a level of host platform independence that reduces development costs and improves time to market. All enablement in Eclipse is currently done at the source level through APIs in the org.eclipse.swt.accessibility package. Although Eclipse has no standard accessibility enablement tools, the IBM Accessibility Center has published a validation tool, the IBM Reflexive User Interface Builder from IBM alphaWorks* (http://www.alphaworks.ibm.com/tech/rib).

Accessibility enablement occurs on at least three levels: platform, infrastructure, and application. Accessibility tools are important at the platform level and critical to achieve success at the application level. All tools must assist the developer in providing accessibility-enabled code and content. Because accessibility enablement is primarily related to user interface development, and in particular, GUI development, most accessibility tools should be integrated into GUI development tools.

As an analogy, accessibility enablement resembles the localization of products. Developers should design enablement into their products from the beginning, when the cost to do so is lower. They need a complete accessibility development infrastructure to support their efforts, including tools and practices. A development organization must define processes that ensure that accessibility is addressed in all phases of the development process. To develop accessible products, developers need accessibility development guidelines and checklists, appropriate schedule time and resources, and management "buy-in."

Accessibility enablement tools are needed to support both the creation of accessible code and content and the repair of existing but inaccessible code and content. Tools to create accessible, rich GUIs and Web content should provide for a palette of accessible controls, prompts that appear when application content is being created, verification of content, and repair of inaccessible content.

Palette of accessible controls

The tool should provide a set of known reusable and accessible user interface controls (also known as "widgets") for the user to select. If the standard controls provided by the host platform are not accessible, the tool should offer some custom controls that have been enabled. The tool should also prompt the developer to specify any accessible properties for these controls when they are selected.

Prompts while creating the content

The tool should guide the developer to make the best possible enablement decisions. It should automate or eliminate many low-level coding tasks. An example of these prompts is a dialog box for inserting an image into a GUI. The dialog box needs to provide a way to add alternative text for that image. For example, the Microsoft PowerPoint** tool provides the dialog box shown in ***Figure 9*** for all images.

Verification of the content

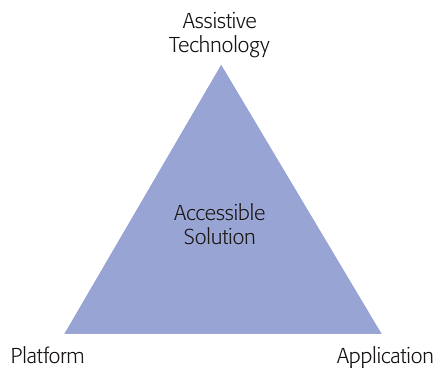
The tool should detect and present to the user any accessibility enablement problems. All accessible tools, which should run during both authoring and build times, must ensure that the generated GUI meets the defined minimum levels of enablement. Some critical requirements for enablement include: GUIs must be fully keyboard accessible; any non-text object must have alternative text; fonts, colors, and timing of events in GUIs should be configurable; and GUIs should respect the system accessibility settings. For example, a Web content creation tool should check each `<img src="...">` tag in the document for an `alt` attribute and report any missing `alt` attributes.

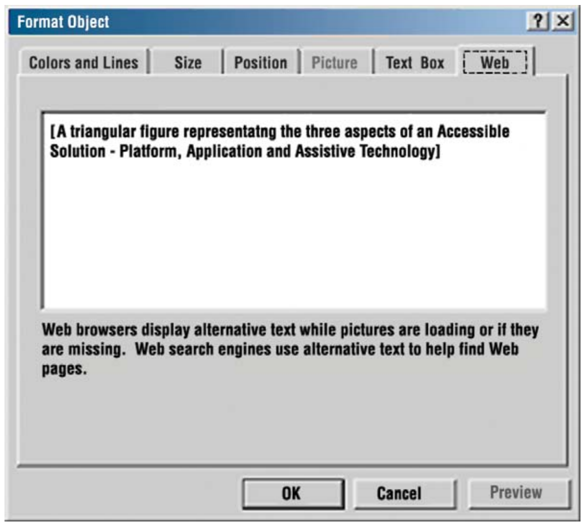
Repair of inaccessible content

The tool should present options for correcting accessibility defects. Automatic repair should only be attempted when it is certain that the repair action is appropriate. When possible, accessibility tools should prevent the creation of inaccessible GUIs and at least warn the developer if one is being created. Prompts should be provided to help the developer create the most accessible GUI possible.

As an example, the Web content creation tool can display a prompt for each `IMG` tag that does not have an `alt` attribute. For greater ease of use, the tool can remember each image source it has seen and automatically provide the `alt` text for repeated use of the same image.

Higher-function tools should go beyond the code-level tasks described previously in supporting accessible design. They should ensure that the GUI is well organized and that the design is properly reviewed for both usability and accessibility enablement.





[A triangular figure representatng the three aspects of an Accessible Solution - Platform, Application and Assistive Technology]

Web browsers display alternative text while pictures are loading or if they are missing. Web search engines use alternative text to help find Web pages.

Figure 9
Microsoft PowerPoint dialog for entering alternative text for images

Numerous commercial vendors provide Web content (HTML) accessibility verification tools, such as the ParaSoft tool, WebKing.[39] Commercial tools to validate rich-client GUIs are less prevalent.

LESSONS LEARNED—MICROSOFT WINDOWS TODAY

We will complete our survey of systems design requirements by providing a perspective on the development of the Windows accessibility architecture. This architecture has evolved over the past decade. It is an example of addressing accessibility after initial release that has resulted in interoperability problems between assistive technologies (ATs) and Windows applications[40] and higher support and enablement costs for both.

Figure 10 is a comprehensive view of the Windows accessibility framework. Applications map to an assistive technology platform layer, which is accessed by assistive technologies. Primarily, we focus on screen reader access because most of the accessibility information used by screen readers is used by other assistive technology solutions. Note that it is not important to understand all the technologies depicted in Figure 10, only that the accessibility environment in Windows is a complex one.

The first accessibility framework for Windows, Microsoft Active Accessibility**[41] (MSAA), was introduced in May 1997. Prior to this date, no accessibility framework existed for Windows, resulting in the loss of access by blind users who instead used the pervasive Disk Operating System[42] (DOS) systems for access. To address this problem, screen reader developers used low-level graphical interfaces, such as the Graphics Device Interface (GDI) in Windows, to read the display[15] in a method called "GDI hooking." This process was slow and resulted in interoperability problems in early GUI AT solutions, such as Screen Reader/2 from IBM and SlimWare Windows Bridge[43] for Windows 3.1 from Synthavoice Computers, Inc. Additionally, mobility access features found in the Apple Computer MacIntosh**[44] were not integrated into Windows 3.1. The Massachusetts commissioner for the blind, Charles Crawford, threatened to boycott Microsoft products[45] as a result. Subsequently, in 1995, Microsoft formed an Accessibility and Disabilities group,[46] which resulted in the inclusion of mobility and low-vision features in Windows 95.

A first initiative of the Microsoft Accessibility and Disabilities group was to create a new accessibility API called Microsoft Active Accessibility (MSAA), which is a Windows architecture that creates Common Object Model (COM) servers for each GUI control. These servers provide functions (through the `IAccessible` and `IDispatch` interfaces) that ATs can use to get information about a control. The values returned are inferred by the MSAA runtime

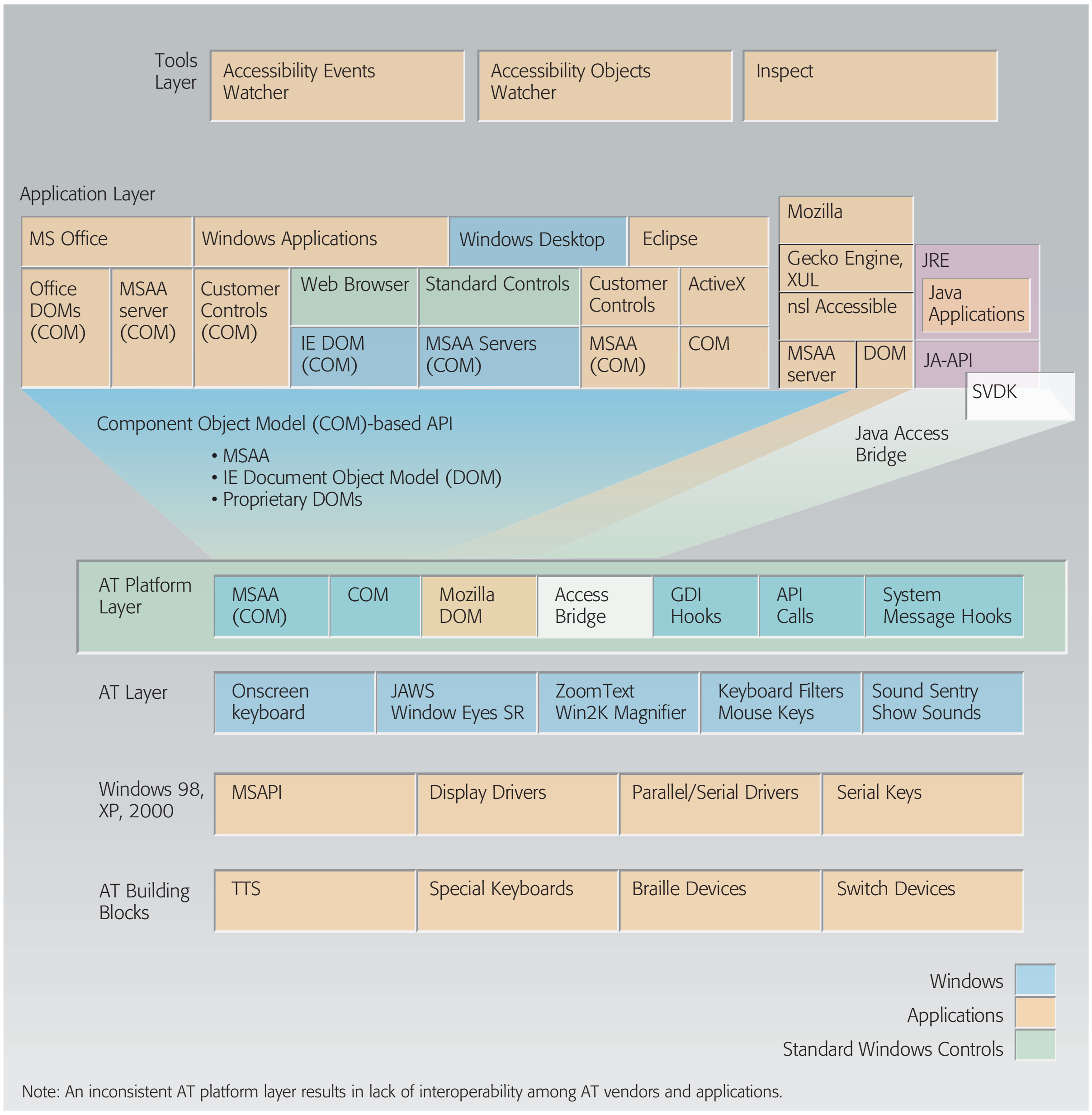


Figure 10
A comprehensive view of the Windows accessibility framework

but, with significant effort, can be augmented by developer-written code.

During MSAA's creation, Microsoft held meetings to discuss the requirements and specification of the API[43] with AT vendors, but the meetings did not include external application developers. Unfortunately, the APIs were inadequate for developing assistive technologies and accessible solutions. Because Microsoft did not build assistive technologies such as screen readers, it was difficult to test the MSAA designs and implementation. Also, MSAA was not implemented to its full capabilities in many of Microsoft's own applications, such as Word.[47] Few developer guidelines existed, causing a lack of support and implementation of the architecture by AT vendors. Additionally, the MSAA architecture[48] lacked support for rich text, documents, complex tables, and languages other than English.[49] As a consequence, AT vendors relied on alternate APIs, such as the Microsoft Word Object Model,[50] to access documents and text.

When Pure Java applications were introduced in 1998 in the Java Foundation Classes, also known as Swing 1.0, they incorporated the Java Accessibility API[51] (JAAPI) as a result of collaboration between IBM and Sun.[52] Because JAAPI, like the JVM** (Java Virtual Machine), needed to run on multiple operating systems, it could not rely on OSM technology. This required the JAAPI to incorporate APIs for rich text and tables and for more comprehensive relationships among GUI objects.[53] IBM developed the Self-Voicing Kit for Java, a pure, cross-platform Java-accessible application reader, to test JAAPI, but it could not read Windows applications. Because MSAA 1.3 had no API for these constructs, Sun created the Java Access Bridge[54] in August, 1999, so that native Windows ATs could access Java applications from a Windows shared library, resulting in yet another accessibility API for Windows.

MSAA's lack of support for these complex controls and constructs extended beyond Java—requiring ATs to continue to use OSM technology to reverse-engineer the information not provided in MSAA. OSM technology is presentation-dependent, and each new application revision would cause screen readers to malfunction. Also, OSM technology is not adequate to handle rich documents, as it does not convey document structure.

The effort to compensate for the missing MSAA function began a general trend by application developers, including developers of Internet Explorer, Mozilla, and other applications, to export their proprietary COM interfaces or shared libraries. Although no platform accessibility API may be all-encompassing, the multiple API sets used by application and AT developers for sharing accessibility information on Windows created an enormous interoperability problem (see Figure 10).

AT vendors began looking at applications that bundled different technologies using different accessibility APIs and object models. Often, the AT vendor did not know which one to use. Even though Microsoft directed developers to use MSAA to make their applications accessible, little information was available on how to implement it properly.[40] As a result, Windows developers expended considerable expense enabling their applications, but they still did not work well with assistive technologies, in part because AT vendors often reverted to their "GDI hooking" reverse-engineered approach.[15]

Microsoft later addressed some deficiencies by introducing MSAA 2.0 extensions for text services, including rich text, and documentation on how to implement MSAA.[55] The updated documentation has helped application developers, but many AT vendors have not adopted the new APIs. In fact, MSAA 2.0 is not even offered by all versions of Microsoft Windows. Consequently applications like Eclipse limit their support to MSAA 1.3. Many AT vendors continue to use the alternative solutions they developed and have not yet adopted the new APIs because they have found, for example, the Microsoft Word DOM to provide more information than MSAA 2.0.

SUMMARY AND CONCLUSIONS

This paper has documented systems design requirements for accessibility. These are summarized in the Appendix.

IBM's extensive background in all fundamentals of accessibility provides a comprehensive and unique perspective for this field. Incorporating accessibility in any platform or any product requires a comprehensive strategy that goes beyond enablement. To ensure a working solution, it is critical that an accessible MVC architecture is in place which supports an extensible yet comprehensive accessibility API set, and at the same time maintains responsiveness to user interactions.

Developer support, including authoring tools that enforce accessibility, sample code, and robust documentation, must be provided. System accessibility features must be identified to ensure that the platform supports them. A comprehensive set of assistive technologies must be provided.

We have focused on solutions for users who are blind or have low vision because of our extensive experience in this area and because most of the features used to produce a working solution for these customers can be applied to other kinds of accessibility solutions.

The evolution of the accessibility of the Windows platform today clearly shows how failure to address these requirements early dramatically affects the cost to develop and support accessible solutions both by the application developer and the developer of the assistive technology.

APPENDIX

Systems design requirements for accessibility covered in this paper include requirements for a platform accessibility architecture, requirements for an assistive technology, and accessibility requirements for applications and tools.

Summary of requirements for a platform accessibility architecture

These include the following:

1. An IT application must be structured so that the application model, views of that model, and control functions that modify the state of the model, are well-isolated from one another. (See "MVC in the context of assistive technologies" and "Requirements for a platform accessibility architecture.")
2. A robust object model containing the necessary semantics must be maintained by the application and communicated through the platform accessibility API. (See "Object model" and "Accessibility requirements for applications and tools.")
3. The object model must provide an API through which an AT can obtain a set of actions and descriptions for each object in the application, and the object model must make all actions available through programmatic access. (See "Object model.")
4. The platform architecture must provide methods for describing the relationships among the objects in the model. (See "Relationships among user interface objects.")
5. The architecture must provide methods for event protocols and event semantics through which the AT is informed of state changes in the current application model and of user inputs. (See "Events.")
6. The architecture must permit the AT to monitor and modify user input events. (See "Events.")

Summary of requirements for an assistive technology

These include the following:

1. The AT must provide for navigation to all features and content using alternative input devices. (See "Navigation to all features and content.")
2. All information must be rendered using alternative output devices. (See "Rendering of all information.")
3. Customization must be supported through settings and scripting. (See "Customization through settings and scripting.")
4. The AT must respond swiftly to user interactions. (See "Performance.")
5. The AT must be compatible with other ATs. (See "Compatibility and interoperability.")

Summary of accessibility requirements for applications and tools

These include the following:

1. The application's object model containing the necessary semantics must be maintained by the application in the context of the accessibility architecture. (See "Accessibility requirements for applications and tools.")
2. Application development tools must provide a palette of accessible controls, supply prompts when application content is being created, verify content, and repair inaccessible content. (See "Accessibility requirements for applications and tools.")

*Trademark, service mark, or registered trademark of International Business Machines Corporation.

**Trademark, service mark, or registered trademark of Microsoft Corporation, Sun Microsystems, Inc., Linus Torvalds, Netscape Communications Corporation, Massachusetts Institute of Technology, Algorithmic Implementations, Inc., The Open Group, or Apple Computer, Inc.

CITED REFERENCES AND NOTES

1. OS/2 Version 1.0 was released in December 1987; IBM Screen Reader/2 Version 1.0 was made available in December 1992. Java Version 1.0 was available in January 1996, and the Java Accessibility API was available as part of the Java Foundation Classes, also known as JFC or Swing, in March 1998. GNOME 1.0 was available in March of 1999, and the GNOME Accessibility API was made available with GNOME 2.4 in September 2003.
2. *History of IBM Accessibility,* IBM Accessibility Center, IBM Corporation, http://www.ibm.com/able/access_ibm/history.html.
3. World Wide Web Consortium (W3C), http://www.w3.org.
4. Web Accessibility Initiative (WAI), World Wide Web Consortium (W3C), http://www.w3.org/WAI/.

5. Authoring Tool Accessibility Guidelines Working Group (AUWG), Web Accessibility Initiative, World Wide Web Consortium, http://www.w3c.org/WAI/AU/.

6. Java Foundation Classes (JFC/Swing), Sun Microsystems, Inc., http://java.sun.com/products/jfc/.

7. The Eclipse Platform Subproject, The Eclipse Foundation, http://eclipse.org/platform/index.html.

8. B. A. Feigenbaum and M. A. Squillace, "IBM Reflexive User Interface Builder", IBM alphaWorks (July16, 2004), http://www.alphaworks.ibm.com/tech/rib.

9. The Trace Research and Development Center at the University of Wisconsin maintains a list of accessible software guideline documents at http://trace.wisc.edu/world/computer_access/software/.

10. *Beyond ALT Text: Making the Web Easy to Use for Users with Disabilities: 75 Best Practices for Websites and Intranets, Based on Usability Studies with People Using Assistive Technology,* Nielsen Norman Group Report (2001), http://www.nngroup.com/reports/accessibility/.

11. G. Vanderheiden, "Fundamental Principles and Priority Setting for Universal Usability," *Proceedings of the ACM 2000 Conference on Universal Usability*, Arlington, VA, ACM Press, New York (2000), pp. 32–37.

12. A. Savidis and C. Stephanidis, "Developing Dual User Interfaces for Integrating Blind and Sighted Users: The HOMER UIMS," *Proceedings of the ACM SIGCHI Conference on Human Factors in Computing Systems*, Denver, CO, ACM Press, New York (1995), pp. 106–113.

13. J. Goldthwaite, "Accessibility Standards for Operating Systems," *ACM SIGCAPH Newsletter* **75**, 2–3 (January 2003).

14. L. Seeman, "The Semantic Web, Web Accessibility, and Device Independence," *Proceedings of the ACM International Cross-Disciplinary Workshop on Web Accessibility* (2004), pp. 67–73.

15. R. S. Schwerdtfeger, "Making the GUI Talk," *Byte Magazine* (December 1991), ftp://ftp.software.ibm.com/sns/sr-os2/sr2doc/guitalk.txt.

16. G. E. Krasner and S. T. Pope, "A Cookbook for Using the Model-View-Controller User Interface Paradigm in Smalltalk-80," *Journal of Object-Oriented Programming* **1**, No. 3, 26–49 (August 1988).

17. B. Shneiderman, "Promoting Universal Usability with Multi-layer Interface Design," *Proceedings of the ACM Conference on Universal Usability* (2003), pp. 1–8.

18. ZoomText, Ai Squared, Inc., http://www.aisquared.com/index.htm.

19. IBM Home Page Reader 3.04, IBM Accessibility Center, IBM Corporation, http://www.ibm.com/able/solution_offerings/hpr.html.

20. *WebBrowser Control Overviews and Tutorials*, Microsoft Corporation, http://msdn.microsoft.com/workshop/browser/webbrowser/browser_control_ovw_entry.asp.

21. MS-DOS (then known as PC-DOS) from Microsoft Corporation was introduced in August 1981.

22. IBM Screen Reader (for DOS) was announced in 1987.

23. Microsoft Accessibility, Microsoft Corporation, http://www.msdn.microsoft.com/library/en-us/dnanchor/html/accessibility.asp.

24. Java Accessibility, Sun Microsystems, Inc., http://java.sun.com/j2se/1.4.2/docs/guide/access/index.html.

25. *GNOME Accessibility for Developers*, The GNOME Project, http://developer.gnome.org/projects/gap/guide/gad/index.html.

26. *Disability Access to GNOME*, The GNOME Project, http://developer.gnome.org/projects/gap/.

27. Technical information for the Microsoft Longhorn project is available at http://msdn.microsoft.com/Longhorn/.

28. Technical information for UI Automation can be found by searching or browsing the reference document at http://winfx.msdn.microsoft.com/.

29. This description is taken from the World Wide Web Consortium's home page for semantic Web activity at http://www.w3.org/2001/sw/. For an introduction to RDF, refer to the "RDF Primer W3C Recommendation," World Wide Web Consortium (February 2004), http://www.w3.org/TR/2004/REC-rdf-primer-20040210/.

30. This specification is a work in progress by the W3C and not yet externally available.

31. JAWS for Windows Overview, Freedom Scientific, Inc., http://www.freedomscientific.com/fs_products/software_jaws.asp.

32. IBM Home Page Reader algorithms are described in the document "IBM Home Page Reader 3.04—Online Help for Developers," IBM Corporation (January 2005), http://www-306.ibm.com/able/solution_offerings/hpr4devhelp.html.

33. Scott Clark, "Java Talks the Talk with IBM's Self-Voicing Kit," *EarthWeb Developer News* (January 1999), http://news.earthweb.com/dev-news/article.php/56261.

34. R. S. Schwerdtfeger and P. D. Jenkins, "IBM's Self-Voicing Kit Technology for Java: IBM's Solution to Bring Cross-Platform Accessibility to Mainstream Computing," *California State University at Northridge Center on Disabilities 1999 Conference (CSUN 99)* (March 1999), http://www.dinf.ne.jp/doc/english/Us_Eu/conf/csun_99/session0098.html.

35. *Section 508 of the Rehabilitation Act: Electronic and Information Technology Accessibility Standards*, The Access Board (December 2000), http://www.access-board.gov/508.htm.

36. *User Agent Accessibility Guidelines 1.0 W3C Recommendation*, World Wide Web Consortium (December 2002), http://www.w3.org/TR/2002/REC-UAAG10-20021217/.

37. *Web Guidelines References and Resources,* IBM Accessibility Center, IBM Corporation, http://www.ibm.com/able/guidelines/web/webreferences.html.

38. K. Harris, "Making Eclipse Accessible to People of all Abilities," *Presentation at EclipseCon 2004* (February 2004), http://www.eclipsecon.org/2004/EclipseCon_2004_TechnicalTrackPresentations/06_Harris.pdf.

39. ParaSoft WebKing, http://www.parasoft.com/jsp/products/home.jsp?product=WebKing&itemId=105.

40. J. J. Lazzaro, "Taking the Mystery Out of Microsoft Active Accessibility," *AccessWorld* **1**, No. 4, (July 2000), http://www.afb.org/afbpress/pub.asp?DocID=AW010404.

41. D. Klementiev, "Software Driving Software: Active Accessibility-Compliant Apps Give Programmers New Tools to Manipulate Software," *MSDN Magazine* (April 2000), http://msdn.microsoft.com/msdnmag/issues/0400/aaccess/default.aspx.

42. *Disk Operating System,* free-definition, http://www.free-definition.com/disk-operating-system.html.

43. P. Schroeder, "A Brief History of Microsoft and Accessibility," *AccessWorld* **1**, No. 4 (July 2000), http://www.afb.org/afbpress/pub.asp?DocID = AW010402.

44. P. Corr, "Macintosh Utilities for Special Needs Users," http://homepage.mac.com/corrp/macsupt/columns/specneeds.html.

45. R. Bellinger, "Microsoft Grudgingly Given Credit for Features in Windows 95 for Disabled," *EE Times* (March 22, 1995).

46. D. Kendrick, "Inside Microsoft: An Accommodating Workplace for People Who Are Blind?" *AccessWorld* **1**, No. 4 (July 2000), http://www.afb.org/afbpress/pub.asp?DocID = AW010406.

47. *Accessibility Aids May Not Work with Office Programs*, Microsoft Corporation, http://support.microsoft.com/kb/q169975/.

48. *Active Accessibility 1.3 SDK*, Microsoft Corp. (April 2003), http://www.microsoft.com/downloads/details.aspx?FamilyId = 4179742F-1F3D-4115-A8BA-2F7A6022B533&displaylang = en.

49. "Microsoft Active Accessibility Version 1.2 is Now Available!" http://teddy.fcc.ro/articles/MSAA.html.

50. *Automating Word Using the Word Object Model*, http://msdn.microsoft.com/library/default.asp?url = /library/en-us/dv_wrcore/html/wroriAutomatingWordUsingWordObjectModel.asp.

51. "Get Ready to Swing (1.0)," *JavaWorld* (March 1998), http://www.javaworld.com/javaworld/jw-03-1998/jw-03-swinggui-p2.html.

52. R. S. Schwerdtfeger and P. D. Jenkins, "100% Pure Java: IBM's Focus to Develop Accessibility for the Next Generation," *California State University at Northridge Center on Disabilities 1998 Conference (CSUN 98)* (March 1998), online proceedings, http://www.dinf.ne.jp/doc/english/Us_Eu/conf/csun_98/csun98_051.htm#_Toc412382694.

53. R. S. Schwerdtfeger, *IBM Guidelines for Writing Accessible Applications Using 100% Pure Java*, IBM Accessibility Center, IBM Corporation (August 2000), http://www-306.ibm.com/able/guidelines/java/snsjavag.html.

54. P. Korn, "JDK 1.2.2, JFC 1.1.1, and Access Bridge EA1 released!" JAVA-ACCESS accessibility interest mailing list archives (August 1999), http://archives.java.sun.com/cgi-bin/wa?A2 = ind9908&L = java-access&F = &S = &P = 69.

55. *Accessible Copy of the Microsoft Active Accessibility SDK Documentation*, Microsoft Corporation (April 2003), http://www.microsoft.com/downloads/details.aspx?FamilyId = 8BC82E65-DBEB-4BC4-9F27-8AC7DF6B7B77&displaylang = en.

Accepted for publication February 28, 2005.
Published online August 3, 2005.

Peter Brunet
IBM Software Group, 11501 Burnet Road, Austin, TX 78758 (brunet@us.ibm.com). Mr. Brunet is a senior software engineer in the Emerging Technologies area of the IBM Software Group. He has been a primary developer or team lead on accessibility products including Home Page Reader, the IBM Java Self-Voicing Development Kit, Speech Viewer III, PhoneAide, Thinking Out Loud, THINKable™, and PhoneCommunicator™. Prior to developing accessibility products, he developed robotics control language interpreters for IBM robots and IBM Series 1™ compiler runtime libraries for BASIC, FORTRAN 77, COBOL, and PL/I. Prior to his IBM career, he developed engineering software for superminicomputers. He has received an Outstanding Innovation Award and an Entry Systems Division Award, has been awarded several patents, and has authored several publications. He holds a BSECE degree from the University of Michigan and an MASCS degree from Florida Atlantic University.

Barry Alan Feigenbaum
IBM Research Division, IBM Accessibility Center, 11501 Burnet Road, Bldg 904, Austin, Texas 78758 (feigenba@us.ibm.com). Dr. Feigenbaum is an architect in the IBM WorldWide Accessibility Center where he provides architectural and development support for IBM accessibility tools. He is the IBM representative to the W3C Web Accessibility Initiative authoring-tools working group. He created the Server Message Block (SMB) network protocol used in IBM and Microsoft networks and the Samba product, and he served on the design team for the industry standard NETBIOS interface. He received a NCSD TeamWork Recognition award, an IBM Outstanding Innovation Award, an IBM Outstanding Technical Achievement Award, five Invention Achievement Awards, and numerous author recognition awards. He has co-authored several award-winning books, several IBM technical reports, and numerous articles in IBM and external publications. He holds a Ph.D. degree in computer engineering from the University of Miami, an M.E. degree in electrical engineering from Florida Atlantic University, and a B.S. degree in electrical engineering from the University of Florida.

Kip Harris
IBM Research Division, IBM Accessibility Center, 11501 Burnet Road, Bldg 904, Austin, Texas 78758 (hkip@us.ibm.com). Mr. Harris is a member of the IBM Worldwide Accessibility Center in Austin, Texas, where he develops and evaluates assistive technology. His most recent projects include a lead role in the development of IBM Home Page Reader. He has been recognized with a variety of awards and patents. Prior to joining the Accessibility Center, Mr. Harris worked with a wide variety of software technologies, including both system and application product development and research work in robotics. He holds a B.S. degree in computer science from Tufts University and an M.S. degree in computer science from the University of Texas at Austin.

Catherine Laws
IBM Research Division, IBM Accessibility Center, 11501 Burnet Road, Bldg 904, Austin, Texas 78758 (claws@us.ibm.com). Ms. Laws is a senior software engineer in the IBM Worldwide Accessibility Center where she is the technical team lead and user interface designer for developing assistive technologies and tools such as IBM Home Page Reader. She also led development teams for the IBM Screen Reader, SpeechViewer, and THINKable projects, holds two patents related to SpeechViewer, and is the IBM representative to the W3C user agent accessibility guidelines working group. She has a B.A. degree in computer science and business administration from Texas Christian University and an M.S. degree in computing technology in education from Nova Southeastern University.

Richard Schwerdtfeger
IBM Software Group, Emerging Internet Technologies, 11501 Burnet Road, Bldg 902, Austin, Texas 78758 (schwer@us.ibm.com). Mr. Schwerdtfeger is a Senior Technical Staff Member, the Software Group Accessibility Strategist and Architect in Emerging Technologies, chair of the IBM Accessibility Architecture Review Board, and an IBM

Master Inventor with over 43 patent filings. His responsibilities include overall accessibility architecture and strategy for IBM Software Group. He is a working member of the working groups for W3C WAI Protocols and Formats and HTML, and previously for the User Agent Accessibility Guidelines, which are now a W3C recommendation. Mr. Schwerdtfeger joined IBM at the Thomas J. Watson Research Center in 1993, where he helped design and develop IBM Screen Reader/2. He later led Java accessibility development at IBM, including the Java accessibility collaboration between IBM and Sun Microsystems and the Self-Voicing Kit for Java, co-designed the Java Accessibility API, and was architectural lead on the Web Accessibility Gateway, which was a transcoding gateway for seniors. He is the co-author of *Secrets of the OS/2 Warp Masters* and the author of *IBM Guidelines for Writing Accessible Applications Using 100% Pure Java™*. He has also published articles, such as "Making the GUI Talk" for *Byte* magazine. He has a B.S. degree in engineering from the University of Connecticut.

Lawrence Weiss
IBM Software Group, Emerging Internet Technologies, 11501 Burnet Road, Bldg 902, Austin, Texas 78758 (lweiss@us.ibm.com). Mr. Weiss is a senior software engineer and the current architect of the Self-Voicing Development Kit for Java, an IBM internal tool that facilitates the development of accessible talking Java applications. He began developing products for people with disabilities when he joined Special Needs Systems in 1988, contributing to several releases of Screen Reader/DOS, Screen Reader/2, Screen Magnifier/2, and SpeechViewer™. In the Accessibility Center, he helped design and develop the Java accessibility API in collaboration with Sun Microsystems, and Home Page Reader, the talking Web browser. Mr. Weiss has presented and demonstrated these and other IBM technologies at many disabilities conferences. He currently holds 16 patents in the field of accessibility with more pending, and is a member of the Austin Software Group Invention Evaluation Board to evaluate and include accessibility in IBM intellectual property. He has a B.S. degree in business administration from the University of North Carolina at Chapel Hill. ■