

Design Checks for Java Accessibility

Simon Harper¹, Ghazalah Khan, and Robert Stevens²

¹Information Management Group — ²BioHealth Informatics Group,
School of Computer Science University of Manchester, Manchester, M13 9PL, UK
¹*simon.harper@manchester.ac.uk* — ²*robert.stevens@manchester.ac.uk*

Abstract

The accessibility of application components is key to enhancing the usability, ergo the user experience, of profoundly blind computer users. However, this accessibility is hampered when platform independent applications are developed because the normal operating system / application interface is abstracted. This means that implicit information inferred by the operating system in platform specific builds is not available in platform independent developments. The current most common platform independent language is Java and the key interface environment for Java is a set of Java Foundation Classes (JFC) known as ‘Swing’. Based on faults, identified from conducting systematic testing of Swing applications, we have developed a web-based programming manual for writing accessible Java applications and checking their accessibility. Here, we present the major points from this manual as an abridged set of programming checks to help programmers overcome many of the technical errors that lead to accessibility faults when programming Swing.

Keywords: Visual Impairment, Java, Java Foundation Classes, JFC, SWING, Design Checks, Programmer, Programming Resource.

1. INTRODUCTION

The widespread use of information technology in organisations and in everyday life, along with socio-political (moral) issues supported by legal enforcement has forced a reengineering of Operating System (OS) interfaces [20]. This reengineering has involved the creation of Application Programming Interfaces (API) to support access by users with disabilities. When software is designed or modified to take advantages of this usability reengineering it is considered to be ‘accessible’ [6]. In this sense, accessibility is the successful access to information and use of information technology by people who have disabilities. In these cases accessibility is achieved by the use of assistive technologies¹ that support a users specific interaction needs, for example, screen-readers² use text-to-speech to provide spoken access in an attempt to support profoundly blind users.

It has long been established that interaction with visual user interfaces, potentially complex and difficult when employed by sighted people [7, 12], is complicated further if the user is profoundly blind (here used referring to the World Health Organisation (WHO) definition; being the inability to distinguish fingers at 3 metres [13]) [8]. Work, including ours, has shown that profoundly blind users are hindered in their efforts to access computer resources, even when these resources are ‘accessible’ [8, 11, 15, 10]. Furthermore, profoundly blind users are at a severe disadvantage when moving around applications components and dialogues compared to their sighted counterparts. We suggest that the ‘playing field’ is not yet level because the usability³ community has typically concentrated on sensory translation and the implementation of optional technological fixes that focus on this translation [15]. The absence of complete guidelines, design and evaluation methodologies, and work on holistic views of accessibility, all hinder profoundly blind users because the fact that profoundly blind people *must* interact with their environment in a markedly different way from that of sighted individuals has been missed [1]. Now that Java ‘supports’ accessibility, programmers may perceive that the accessibility ‘problem’ in Java has been fixed. However, this is not the case, indeed this viewpoint has lead to many Java applications having broken accessibility.

¹Hardware or software that is used to increase, maintain or assist the functional capabilities of people with disabilities.

²Screen-readers are special applications that vocalise the onscreen data. Pages are typically read from the top left to the bottom right, one word at a time.

³... and therefore the HCI community.

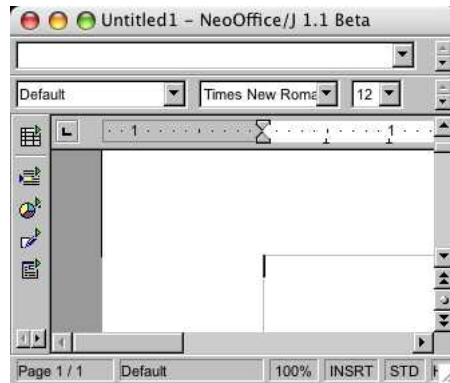


FIGURE 1: Accessibility Problems (NeoOffice J)

To provide solutions to this ‘broken accessibility’ we needed to ask the questions:

“What didn’t work?” and “Why didn’t it work?”

This paper aims to answer these questions by formulating a set of accessibility principles derived from informal observational experiments with profoundly blind users. These principles are then codified as tests and applied in a systematic manner to Java Swing applications. Application components that fail a test are then examined⁴ to identify what (if anything) the programmer has missed. These failures are then used to create our programming manual⁵ for writing accessible Java applications, and our abridged set of programming checks (presented in this paper).

1.1. Problem

To fully realise the problems users with low-vision⁶ encounter we suggest that our sighted readers start a Java application and limit the window size to the top left fifth of the screen (see Figure 1). Now interact with the application using only the screen area and disregard everything that is not visible. In our example we have used NeoOffice J running on an Apple Mac. Now note the problems you have; we believe you will find that:

1. You cannot get an overall feel for what is off the viewable window.
2. You do not know where you are in the window or if you’ve been there before.
3. Consequently, you become disoriented.
4. You cannot tell where you are once the movement arrows are selected.
5. There is too much detail for the viewing area and it is too complex;
6. You cannot tell if the information on the target window is the information you are expecting or require.
7. You often find you tab from control to control all with the same name (or left blank) and in effect become lost in the repetition.
8. The whole experience is both time consuming and frustrating.

Indeed, these problems are exactly those faced by sighted people using a Personal Digital Assistant (PDA – like a ‘Palm’). The only difference here is that applications have been especially built from the ground up to overcome the limited screen space [3].

The problems above are compounded if the reader is profoundly blind and must rely on text-to-speech technology. In this case the software starts to speak when a window / dialogue is loaded. In the case of a typical application: descriptions of the components may or may not be spoken (based on their explicit descriptions); components not selectable may not be read at all; buttons and fields may be tabbed in the incorrect order, or not at all. All these problems are plainly unsatisfactory.

1.2. Synopsis

The scope of our work is restricted to the analysis of the accessibility of Java Swing applications for profoundly blind users who require the use of a screen-reader to access computers. This work reports on the

⁴... therefore we only test Open Source applications otherwise we cannot examine the code.

⁵<http://augmented.man.ac.uk/documents.shtml>

⁶Users with low vision using screen magnification technology, which enlarges the information on the screen by pre-determined incremental factor [for example, 1x magnification, 2x magnification, 3x magnification, etc.]. Magnification programs run simultaneously and seamlessly with the computer’s OS and applications.

formative testing and analysis of the accessibility of Swing; achieved by designing a testing tool to conduct systematic testing of Swing applications using JAWS. Based on faults, identified from this process, we develop a web-based programming manual for writing accessible Java applications. Here, we present the major points from this manual as an abridged set of programming checks to help programmers overcome many of the technical errors that lead to accessibility faults when programming Swing.

- 2. Java Accessibility Primer** ⁷ Here we provide background information relating to Java Accessibility including a brief history and the current state of accessibility. We discuss why platform independence often makes accessibility more difficult and investigate how the Java Accessibility Bridge was intended to solve these problems; we also see why this was not completely successful.
- 3. Testing for Accessibility** Java Swing accessibility has not been completely successful (as observed in section 2). Here, we describe a testing framework (derived from informal observational experiments with profoundly blind users) that allowed a number of applications to be investigated with regard to their accessibility. An analysis of the source code was conducted upon failure of any one of these tests. In this way common programmatic mistakes could be identified.
- 4. Design Checks** These programmatic mistakes were then formulated into a series of design checks which attempt to support enhancements aimed at filling the current gaps in Java Swing accessibility and enabling better usability for profoundly blind users. Our chief goal, was the design of appropriate programming techniques so that components intended to be accessible were, indeed, accessible. Such checks assist in the transformation of 'broken accessibility' and actively support the programmer in making sure all design requirements are fulfilled.
- 5. Conclusions** Finally, we focus on our conclusions from the work undertaken and look at future work including tool automation.

2. JAVA ACCESSIBILITY PRIMER

Since the 1980s, visually impaired people have used computers via screen-readers. During the 1980s, such applications relied on the character representation of the contents of the screen to produce spoken feedback [20]. In the past few years, however, the widespread trend towards the development and use of the Graphical User Interface (GUI) has caused several problems for profoundly blind users. The introduction of GUIs has made it virtually impossible for blind people to use much of the industry's most popular software and has limited their chances for advancement in the now graphically oriented world. GUIs have left some screen-readers with no means to access graphical information because they do not use a physical display buffer, but instead employ a pixel-based display buffering system [8]. A significant amount of research and development has been carried out to overcome this problem and provide speech-access to the GUI. Companies are now enhancing screenreaders to work by intercepting low-level graphics commands and constructing a text database that models the display. This database is called an Off-Screen Model (OSM); and is the conceptual basis for GUI screen-readers currently in development.

2.1. The Off-Screen Model

An OSM is a database reconstruction of both visible and invisible components [16]. A database must manage information resources, provide utilities for managing these resources, and supply utilities to access the database. The resources the OSM must manage are text, off-screen bit maps, icons, and cursors. Text is obviously the largest resource the OSM must maintain. It is arranged in the database relative to its position on the display or to the bit map on which it is drawn. This situation gives the model an appearance like that of the conventional display buffer. Each character must have certain information associated with it: foreground and background colour; font family and typeface name; point size; and font style, such as bold, italic, strike-out, underscore, and width.

Merging text by baseline (i.e. position on the display) gives the model a 'physical-display-buffer' like appearance with which current screen-readers are accustomed to working. Therefore, to merge associated text in the database, the model combines text in such a way that characters in a particular window have the same baseline [16]. Each string of text in the OSM has an associated bounding rectangle used for model placement, a window handle, and a handle to indicate whether the text is associated with a particular bit map or the display. GUI text is not always placed directly onto the screen. Text can be drawn into memory in bit-map format and then transferred to the screen; or can be clipped from a section of the display and saved in memory for placement back onto the screen later.

⁷Please feel free to jump this primer if you already conversant with the issues.

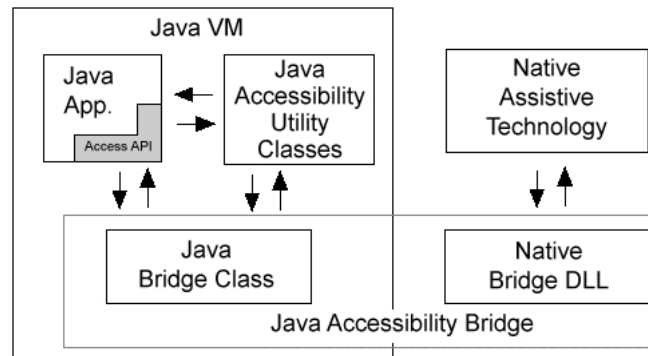


FIGURE 2: How the JVM, Java Accessibility API and Access Bridge work together with the host system to provide the relevant information to an assistive technology

In windowing systems, more than one application can be displayed on the screen. However, keyboard input is directed to only one active window at a time, which may contain many child windows; the applications cursor⁸ is then placed in at least one of these child windows. The OSM must keep track of a cursor's window identification (i.e. handle) so that when a window becomes active the screen-readers can determine if it has a cursor and vocalize it. Additionally, the OSM must keep track of the cursor's screen position, dimensions, the associated text string (i.e. speak-able cursor text), and string character position. If the cursor is a blinking insertion bar, its character position is that of the associated character in the OSM string. In this case, the cursor's text is the entire string. A wide rectangular cursor is called a selector, since it is used to isolate screen text to identify an action. Examples of selector cursors are those used for spreadsheets or drop-down menus. The text that is enclosed within the borders of the selector is the text that the screenreader would speak.

Modifying current screen-readers to accommodate the new GUI software is no easy task. Unlike DOS, leading GUI software operates in multitasking environments where applications are running concurrently [2]. The screen-reader performs a juggling act as each application gets the user's input.

2.2. The Java API [18]

In an attempt to solve the accessibility problems with screen-readers and Java Swing, Sun Microsystems (Sun) introduced the Java Accessibility API in 1997 which was designed as replacement for the OSM. With a Java application that fully supports the Java Accessibility API, no OSM model is necessary because the aim of the API is to provide all of the information normally contained in an OSM [19].

The Java Accessibility API defines a contract between individual user-interface components that make up a Java application and an assistive technology that is providing access to that Java application, through the Accessible interface. This interface contains a method to get the `AccessibleContext` class, which contains the core common set of accessibility information that every user-interface object must provide. If a Java application fully supports the Java Accessibility API, then it should be compatible with assistive technologies such as screen-readers. The Java Accessibility API package currently consists of eight Java programming language interfaces and six Java programming language classes [17].

2.3. The Java Accessibility Bridge

In order for existing assistive technologies available on host systems (e.g. Microsoft Windows, Macintosh, OS/2) to provide access to Java applications, they need some way to communicate with the Java Accessibility support in those applications. The Java Accessibility Bridge supports that communication [14]. The Access Bridge for Microsoft Windows makes it possible for a Windows based assistive technology like JAWS to interact with the Java Accessibility API [5]. The Java Accessibility API is implemented in the Swing user interface components. The Access Bridge is a class which contains 'native methods', indeed, part of the code for the class is actually supplied by a dynamically linked library (DLL) on the host system [14]. The assistive technology running on the host (e.g. a Macintosh screen-reader) communicates with the Macintosh native DLL portion of the bridge class. This bridge class in turn communicates with the Java Virtual Machine, and from there to the Java Accessibility utility support and on to the Java Accessibility API on the individual user interface objects of the Java application it is providing access to (see Figure 2).

⁸A screen-reader cursor is the area on the display where the next potential action will occur or where users can enter their next text.

For example, in order for JAWS to provide access to Java applications running on Microsoft Windows, it would make calls to the Java Accessibility Bridge for Windows. When the user launches a Java application, the bridge would inform JAWS of this fact. Then JAWS would query the bridge and the bridge would in turn forward those queries on to the Java Accessibility Utilities that were loaded into the Java Virtual Machine. When those answers came back to the bridge, the bridge would forward them on to JAWS [17, 16].

2.4. The Problem of Platform Independent Software

The problem of platform independent software is easily surmountable by using technologies like the accessibility API and Bridge, but faults still persist. The nub of the problem of platform independent developments is the fact that an additional level of indirection is present when compared with native developments. With a native development information regarding text, titles, window, and component status are implicitly accessible by the native OS and so in many cases this information does not need to be explicitly provided. This is not the case with platform independent software (as Fig 2 shows). In this case the bridge component provides an intermediary step by implementing an OS native end and a conduit to the Java native end. However, the application still cannot correctly resolve accessibility information unless it is made explicit. In summary then, the Java Virtual Machine (JVM), Java Accessibility API, and the Java Access Bridge work together with the host system to provide the relevant information to an assistive technology. Because 'implicit' hooks are not available a failure, at the programming level, to make information explicit can invalidate the entire pipeline and break the accessibility of the application.

3. TESTING FOR ACCESSIBILITY (A RATIONALE)

The accessibility work discussed in the previous section suggests that the inaccessibility of Java applications has to a great extent been overcome. Our objective was to see if this was really the case or if anomalies were present. To accomplish this we conducted a set of informal observational experiments [4] with profoundly blind JAWS users focusing on formative testing and analysis of the accessibility of different Java Swing applications.

In each case we compare these techniques to the expected outcome in an effort to understand user cognition via a standard preference indicator [9]. The problems encountered were then classified and a set of principles evolved. These principles formed the basis for a set of questions which could be delivered in a systematic way. A testing tool to enable non-experts to conduct the systematic testing of Swing applications was then created. This tool was used to test a set of applications (such as 'JEdit'⁹ and 'OilEd'¹⁰) and based on faults, identified from this process, we developed a web-based programming manual, and a set of design checks to assist programmers develop accessible Java applications. We can break each stage of this process down as follows:

1. **Informal Observational Experiments:** Informal observational experiments were conducted with both sighted and profoundly blind JAWS users. These experiments focused on testing applications to see what spoken information was necessary for both good accessibility, and good usability. Java Accessibility API-friendly components offer up much more of themselves than just mouse clicks. One of the other strengths of the API is that an assistive technology can actually query components. It can say 'Give me all the information about you; your name, what you do, are you a button, are you a scroll bar, are you checked, are you unchecked, what is your colour, what type of fonts can you use?'. The Accessibility API is also extensible in its capabilities. Accessibility is constantly evolving but the API has been designed so that greater granularity can be added as time goes by without breaking anything in the process. However, our experiments identified a number of failures in this support when evaluated with subjects using JAWS. Given the comprehensiveness of the API the question we asked ourselves was 'Why?' Records were kept as to what was missing from each failed interaction in an attempt to answer this question. These were later used to formulate a set of questions (a checklist) for testing other applications.
2. **Formulation of Questions (for systematic delivery):** Eventually, a set of eight questions were formulated so that, when delivered systematically, the results could inform a consistent accessibility metric of the application; and suggest a path to the rectification of the accessibility failure. These took the form of 'closed' questions where a simple Yes/No answer was expected as a response. In this way semi-automated tools could deliver the testing framework and non-expert users would find the questioning sequence both fast and easy.

⁹<http://www.jedit.org/>

¹⁰<http://oiled.man.ac.uk/>

3. **Encoding of Question Within a Semi-Automatically Delivered Framework:** We then created a simple testing tool to deliver our questions and record the responses. For each component eight questions would be asked and these details along with the components information was processed into an accessibility profile for that specific component. these questions are based upon the actions a screenreader needs to be able to support in order to carry out its task. for example, being able to move to and from a widget with the keyvaord; being able to determine widget state; etc. These questions comprised: (1) Does the screen-reader read the component name? (2) Does the screen-reader read the component label? (3) Does the screen-reader read the current component state? (4) Can you navigate to component with keyboard? (5) Can you navigate from component with keyboard? (6) Can you navigate within component with keyboard? (7) Can you select the component with keyboard? and finally, (8) Can you edit component state with keyboard?
4. **Systematic Application of Framework:** By applying the framework to a number of open source applications it was possible to investigate the source code to ascertain the programming difficulties that lead to broken accessibility. A record of these difficulties was added to each profile and this enabled us to rank the most common difficulties and propose techniques for their solution. The Java Accessibility API consists of a rich set of Java interfaces and classes to effectively deal with almost any accessibility issue with Swing applications. The Accessibility API builds a range of capabilities directly into the individual user interface components and thereby providing direct communication with objects. In fact, the only problem for which a solution could not be found due to a lack of Java functionality was the keyboard navigability of disabled components. This issue is currently being investigated by Sun.
5. **Formulation of Design Checks:** The Java windowing interface presents information in the way that the user chooses, be it tactile, audio, or visual. By using the Java Accessibility API, components enter into a 'contract with screen-readers, offering their internal workings via a simple, well-defined, and secure interface. Unfortunately, many programmers do not enter into a 'contract' with the Java Accessibility API. By formulating solutions and presenting them as part of a set of design checks we hope to encourage more programmers to fully engage with Java accessibility.

4. DESIGN CHECKS (ABRIDGED)

The JFC library, which implements the Swing components, supports accessibility by implementing both basic keyboard accessibility and the Java Accessibility API. This allows JAWS to extract and interact with the information that visually disabled users need to access often complex information. As a result, the millions of developers using the JFC APIs to build user interfaces for their applications are automatically gaining enabling technology in their development environment. Due to the accessibility support built into the Swing user interface components, even developers with no expertise in special needs or assistive technologies can easily build highly accessible applications. However, we have discovered that there are a few extra rules developers need to keep in mind to build basic support for accessibility into a Swing application. We call these 'Design Checks' and these are summarised below:

Provide descriptive component text: Call `setAccessibleDescription()` to set the accessible descriptions on all accessible components in an application. In the Java Accessibility API, `AccessibleDescription` is a text description of an object's purpose. For example, if the object is a Web page link containing an image, this should be set to the URL string of the target. Some JFC components use tool tips to provide this information, even if the developer doesn't explicitly set the `AccessibleDescription` of an object. Therefore, if a program uses tool-tip text, this check may already be complete.

Provide descriptive text for icons and other graphics: For `JLabels` and `JButton` classes that contain only images, call `setAccessibleName()` to set the object's `AccessibleName` property. In the Java Accessibility API, the accessible name of an object is simply the object's name.

To use an icon in a Swing program that supports accessibility, call JFC's `setIcon()` method to create a `JFC ImageIcon`. Then the `ImageIcon` class's `setDescription()` method can be used to provide text descriptions of the icons to visually disabled users. Swing uses `ImageIcon` objects to place images in many different kinds of components, ranging from JFC buttons to labels. `ImageIcons` should also be used when images need to be inserted into a `JTextComponent` object or into JFC components subclassed from `JTextComponent`.

Always set the focus: Make sure that a component in your application has the input focus at all times. If no component has the focus, a screen-reader program cannot relate the applications status to a blind

or sight-impaired user. Most Swing components implement focus-setting mechanisms to support keyboard navigation, therefore, when a Swing application is keyboard-enabled the Swing API should take care of all focus-setting requirements.

Set mnemonics on components: Mnemonics are keyboard equivalents for component activation. Whenever there is a component that supports a `setMnemonic()` method, set the mnemonic to a unique key sequence to activate the object. In a Swing application, it is important that the mnemonic is always set for the first `JMenuBar` entry to allow a disabled user to get there using the keyboard.

Set keyboard accelerators in menus: `JMenuItem` components provide a `setAccelerator()` method to set a keyboard accelerator key. This key lets the user activate a menu item without having to go through the menu. When possible, it is a good (though not mandatory) practice to provide keyboard accelerators.

Label components properly: When using Swing's `JLabel` class to provide a label for a component, call `JLabel`'s `setLabelFor()` method to associate the component with its label. If both the `displayedMnemonic` and `labelFor` properties of a Swing component are set, the component's label calls the `requestFocus()` method of the component specified by the `labelFor` property when the appropriate mnemonic is activated.

Group objects inside named panels: When objects appear in groups, it is important to group those components logically and to assign each group of components a name. This practice makes it easier for users with disabilities to navigate your application. One way to group a set of components is to add the group to a `JPanel` and then set the `JPanel`'s `AccessibleName` property. You can save yourself some trouble by using a `JPanel` with a `JFC TitledBorder`. Then you won't have to set the accessible name because the JFC will do that for you.

Make sure custom components support accessibility: Developers must be aware that subclasses of `JComponent` are not automatically accessible. Custom components that are descendants of other Swing components should override inherited accessibility information as necessary.

These are the core checks that developers must follow to ensure that their Java applications meet at least the basic accessibility requirements. However, there is much more that can be done using the tools and software available from Sun to make Java technology even more accessibility-friendly. More detailed checks with code examples can be found in the Online Programming Manual (OPM)⁵. This is a web document, designed for Java programmers, providing extensive guidelines for developing accessible applications including creating accessible custom components if the standard Swing components do not fulfil requirements. On a final note, it is important to stress the importance of considering accessibility issues from the initial design stages of the software life cycle, as all too often issues of accessibility are an afterthought in the development of new technology.

5. CONCLUSIONS

If Java accessibility has come so far, why are basic accessibility features still compromised in Swing applications today? The main reason lies with Java developers, either due to ignorance of the issues which disabled users face with their software, or wrongfully assuming that it is difficult to implement accessibility features into their Java applications. Following the simple accessibility guidelines for custom components as outlined in our Design Checks and programming manual could have prevented many accessibility bugs. Java developers must realise that it is relatively simple to incorporate accessibility into applications that make use of JFC because the Java Foundation Classes implement accessibility methods in Swing's user interface components. Application developers should take advantage of this simple way to incorporate accessibility into their designs.

Sun has spent several years developing an extensive range of accessibility tools and software for Java technology developers, but they are only useful if developers are actively taking advantage of them. What most Java developers fail to realise is that implementing accessibility techniques into their software is not nearly as difficult as they may think; little effort is required at no extra cost. A suggestion could be for Sun to provide awareness education and training sessions for Java developers around the globe on accessibility issues and their relation to the use of Java applications by all users. This will emphasise to the Java developer that accessibility means building the services and support into an application that enable people with disabilities to use the software as well as anyone else.

Based on faults, identified from conducting systematic testing of Swing applications, we have developed a web-based programming manual for writing accessible Java applications. Here, we have presented the

major points from this manual as an abridged set of programming checks to help programmers overcome many of the technical errors that lead to accessibility faults when programming Swing. Our work, however, is not yet complete. We propose that further work needs to be undertaken along 2 paths: create a set of formal observational experiments with profoundly blind users to confirm our initial informal findings; and extend the system such that an automated analysis tool can be created.

REFERENCES

- [1] Mark H. Ashcraft. *Fundamentals of cognition*. New York ; Harlow : Longman, 1998.
- [2] Eric Bergman and Earl Johnson. Towards accessible human-computer interaction. *Advances in Human-Computer Interaction*, 5(1), 1995.
- [3] Orkut Buyukkokten, Hector Garcia Molina, Andreas Paepcke, and Terry Winograd. Power browser: Efficient web browsing for PDAs. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 430–437. ACM Press, 2000.
- [4] Samuel G. Charlton and Thomas G. O'Brien. *Handbook of Human Factors Testing and Evaluation*. Lawrence Erlbaum Associates, London, UK, 2nd edition, 2002.
- [5] Dan Clark. JAWS for Windows and Java Accessibility. *Archives of the National Federation of the Blind of California*, 1999. <http://www.nfbcal.org/nfb-rd/1565.html> - circa 2005.
- [6] Ian Darwin. Gui development with java. *Linux J.*, 1999(61es):4, 1999.
- [7] Alan Dix, Janet Finlay, Gregory Abowd, and Russell Beale. *Human-computer interaction*. Prentice-Hall, Inc., 1997.
- [8] W. Keith Edwards, Elizabeth D. Mynatt, and Kathryn Stockton. Access to graphical interfaces for blind users. *interactions*, 2(1):54–67, 1995.
- [9] Valerie J. Gawron. *Human Performance Measures Handbook*. Lawrence Erlbaum Associates, London, UK, 1st edition, 2000.
- [10] Simon Harper, Carole Goble, and Robert Stevens. Traversing the Web: Mobility Heuristics for Visually Impaired Surfers. In Tiziana Catarci, Missimo Mercella, John Mylopoulos, and Maria E. Orłowska, editors, *Proceedings of the Fourth International Conference on Web Information Systems Engineering (WISE'03)*, pages 200–209, Los Alamitos California, USA, December (10–12) 2003. IEEE Computer Society.
- [11] Christophe Ramstein, Odile Martial, Aude Dufresne, Michel Carignan, Patrick Chassé, and Philippe Mabillean. Touching and hearing gui's: design issues for the pc-access system. In *Assets '96: Proceedings of the second annual ACM conference on Assistive technologies*, pages 2–9. ACM Press, 1996.
- [12] Jef Raskin. Looking for a humane interface: will computers ever become easy to use? *Commun. ACM*, 40(2):98–101, 1997.
- [13] Various RNIB. A short guide to blindness. Booklet, Feb 1996. <http://www.rnib.org.uk>.
- [14] Mary Smaragdis. Bridging the Gap: Java Access Bridge Links Windows-based Assistive Technologies to the Java Platform. *Sun Java Accessibility Documentation*, 2000. <http://java.sun.com/features/2000/03/accessbridge.html> - circa 2005.
- [15] Hironobu Takagi, Chieko Asakawa, Kentarou Fukuda, and Junji Maeda. Accessibility designer: visualizing usability for the blind. In *ASSETS '04: Proceedings of the ACM SIGACCESS conference on Computers and accessibility*, pages 177–184. ACM Press, 2004.
- [16] Various (Sun Accessibility Team). Accessibility Support for the Java Platform. *Sun Java Accessibility Documentation*, 1998. <http://www.sun.com/access/articles/java.access.support.html> - circa 2005.
- [17] Various (Sun Accessibility Team). The Basics of the Java Platform: A User-Focused Discussion. *Sun Java Accessibility Documentation*, 2000. <http://www.sun.com/access/articles/wp-csun00/> - circa 2005.
- [18] Various (Sun Accessibility Team). Java Accessibility API. *Sun Java Accessibility Documentation*, 2003. <http://java.sun.com/j2se/1.4.2/docs/api/index.html> and <http://java.sun.com/j2se/1.4.2/docs/api/javax/accessibility/package-sum%mary.html> - circa 2005.
- [19] Various (Sun Accessibility Team). A Primer on the Java Platform and Java Accessibility. *Sun Java Accessibility Documentation*, 2005. <http://www.sun.com/access/articles/wp-caped/> - circa 2005.
- [20] Carlos A. Velasco and Tony Verelst. Raising awareness among designers accessibility issues. *SIGCAPH Comput. Phys. Handicap.*, (69):8–13, 2001.