Developing accessible software for data visualization

by D. Willuhn
C. Schulz
L. Knoth-Weber
S. Feger
Y. Saillet

An increasing number of countries are establishing regulations that require information technology to be fully accessible by all users. To make software products accessible, they must be designed and implemented so that they can also be used by persons with sensorial or physical impairments. Although checklists and coding guidelines are available, this requirement can be challenging, especially when implementing accessibility for a product that displays information in a graphical format. This paper presents a process-oriented view of addressing accessibility issues in software development. It describes what had to be taken into consideration and how accessibility issues were solved. The experience was gained during the development of the IBM DB2® Intelligent Miner™ Visualization, an application used to visualize data mining results. The following areas are addressed: integrating design for accessibility into the development process, providing special design features that enable users to interact with graphical data, and programming solutions for accessibility features.

A significant number of users of information technology (IT) products are affected by some kind of disability such as impaired vision, hearing, mobility, or cognition.[1,2] To accommodate the special needs of these users, many regulations and standards for accessibility are in effect worldwide.[3] An increasing number of businesses and organizations have purchasing requirements that stipulate accessible products. Hence, software manufacturers must consider accessibility requirements as an important aspect of product development.

When software is accessible, individuals with disabilities can use assistive technology to increase, maintain, or augment their functional capabilities. For example, blind persons can use a screen reader to have the information displayed on the graphical user interface (GUI) of a product read to them. At a minimum, developers must provide the appropriate prerequisites by which assistive technologies can interact with a given product. At best, the product provides special forms of input and output techniques that suit the needs of disabled users.

Principles and guidelines for accessibility design for various types of software are available from standardization organizations, government authorities, and human-computer interaction (HCI) research.[4–6] Software manufacturers provide developers with accessibility checklists based on these regulations. For example, the IBM accessibility checklists[7] specify principles and guidelines for input methods, output methods, and consistency and flexibility in software applications as follows.

For choice of input methods:

0018-8670/03/$5.00 © 2003 IBM

- Support the user's choice of input methods, including keyboard, mouse, voice, and assistive devices via the serial port.
- Provide keyboard access (mouseless operation) to all features and functions of the software application as the primary requirement.
- Provide support, usually by means of the operating system, for input via the serial port, keyboard movement of the mouse pointer, and other keyboard enhancements.

For choice of output methods:

- Support the user's choice of output methods, including display, sound, and print.
- Provide text labels for icons, graphics, and user interface elements and support visual indications for sounds as the primary requirement.
- Implement the accessibility application programming interfaces (APIs), for example, Java** accessibility or Microsoft active accessibility, so that the target operating system meets this principle.

For consistency and flexibility:

- Make the application consistent with the user's choice of system behavior, colors, font sizes, and keyboard settings.
- Provide a user interface that can be customized to accommodate the user's needs and preferences, including fonts, colors, and display layout.

At the lowest level, checklists give specific recommendations for designers and programmers, such as "provide keyboard equivalents for all actions" or "provide a variety of color selections capable of producing a range of contrast levels." In addition, they provide examples and tips for coding and testing accessibility features.

Coding guidelines, such as the *IBM Guidelines for Writing Accessible Applications Using 100% Pure Java*™, [8] provide application developers with essential information about programming practices and how to use the Java accessibility API.

In this paper, we address (1) making designing for accessibility part of the development process, which includes planning for accessibility, additional efforts for coding, testing, and writing software, and implementing multidisciplinary teamwork; (2) providing special design features to enable all users to interact with graphical data, such as with different ways of presenting information, color palettes that are suitable for a wide range of users, keyboard navigation in on-screen diagrams, and assistive technologies such as screen readers; and (3) using programming to produce accessibility features such as high-contrast settings and keyboard handling.

Accessibility in the User-Centered Design process

Merely consulting accessibility guidelines during development is not sufficient because they share the same fate as all other kinds of design guidelines: They are context-independent and thus need to be interpreted in each situation in which they are applied. Even when they are as specific as giving details of implementation, developers cannot be sure if a guideline is at all appropriate in a given context. The same holds for usability. A product may conform to all generally accepted design recommendations but may still lack usability in certain contexts of use. To avoid this problem, a User-Centered Design (UCD) approach must be followed. [9,10]

Developing accessible products requires a process-oriented, multidisciplinary approach. Accessibility requirements must be addressed from the very start, when design and evaluation activities are initiated, through all phases of the development process. Thus, accessibility must be integrated in a UCD process. [11]

In the UCD process, a team of experts works on all aspects in all phases of the development cycle. A typical UCD team is comprised of experts for user interface programming, human-computer interaction design, visual design, information development design, and user feedback. All of these disciplines are also needed when it comes to designing and implementing accessibility. Thus, all team members need a thorough understanding of how accessibility requirements affect their work and how it is related to the work of the other disciplines.

Design objectives. The product under discussion, the IBM DB2* Intelligent Miner* Visualization Version 8.1, presents the results of data-mining functions and statistical functions. Customized visualizers are available for depicting clustering, tree classification, or association analyses. Each visualizer deploys various types of diagrams and color-coding techniques to facilitate the comprehension of complex data and relationships. [12]

The extensive use of graphic representions poses a number of challenges with regard to accessibility re-

quirements. For example, diagrams and color coding provide a powerful means of conveying the meaning of and the relationships among complex data for individuals with normal eyesight. However, the very same means makes this information almost inaccessible to users with impaired eyesight. The objective was to find a solution that (1) serves the needs of disabled users as defined in the applicable accessibility checklist, (2) does not compromise usability for the majority of users without disabilities, and (3) keeps development resources within acceptable limits.

Activities and teamwork. Four phases are involved in the design process.

The planning phase. The IBM Java Accessibility Checklist and coding guidelines were a pivotal resource at the beginning of our UCD process. They helped the team to identify accessibility requirements for user interface (UI) design, visual design, and information design, and to determine which skills, tools, and efforts were needed by team members to work on accessibility requirements in their fields of responsibility. This aspect was crucial, especially because efforts for coding and testing were significantly higher than usual. This need for additional resources could not have been satisfied if it had been identified later in the development process.

The conceptual design phase. At the beginning of the design activities, the UCD team discussed how the accessibility requirements affect the design concept for each visualizer. It was quite obvious that software that aims at explaining complex data and relationships by visualization is very difficult for persons with visual impairments to use. Even if the relevant data in the charts and the diagrams can be accessed by a screen reader, it is very difficult to navigate within diagrams and to understand the meaning of the data.

To solve this problem, we decided on a design strategy that is based on offering alternative presentations, or views, of the data. Besides a graphical view, data are shown in a textual and a tabular form. See the next section on user interface design for a detailed discussion of these views.

To ensure that users have mouseless access to all features, a special design specification for keyboard operations was written. We intended to follow existing conventions as defined in the Java and Microsoft Windows** design guidelines as closely as possible. Additionally, we wanted to introduce supplemental features for keyboard operation where needed.

The detailed design phase. The main activity of this phase was to elaborate the design of views, interactions, and navigation techniques. Elaboration was done by developing prototypes in an iterative fashion. Each prototype was evaluated against the requirements of the accessibility checklist. Necessary design changes were identified and implemented in the next prototype. The results are shown in detail in the next two sections of this paper.

During the detailed design phase, the keyboard operation specification needed several revisions. Although keyboard operations were increasingly implemented and tested, we discovered that a number of the assignments could not be realized. Usually this was a result of technical limitations and side effects in the Java development environment. In these instances, the UI programmer and the UI designer worked together to find new keyboard assignments that fitted within the overall concept.

Also in this phase, we started to test the accessibility. Recommended test tools such as screen readers and code inspection programs were used to check that requirements were met. Screen readers especially turned out to be tricky to use. At the time of development, no single screen reader was available that could work smoothly with all features of Java applications. We decided to test with two readers in parallel, JAWS for Windows 4.02, and the IBM Self-Voicing Kit for Java Version 1.3. Other recommended Java accessibility test tools that we used were Java Accessibility Helper, Ferret, and Monkey.[13]

The evaluation phase. In this phase, a series of accessibility evaluations was conducted with the test tools mentioned in the detailed design phase. During function verification test (FVT), test cases covered all features of the user interface. We followed the approach that is recommended in the IBM Java Accessibility Checklist:

- Test using the keyboard only
- Test using assistive technology
- Test using accessibility testing tools

This activity resulted in about 40 accessibility defects—approximately twice the number of usability defects found. These accessibility defects were mainly related to keyboard operation (navigation and interaction in the user interface) and screen reader use

Table 1 Responsibilities and activities

Responsibility	Planning	Conceptual Design	Detailed Design	Evaluation
User experience design lead	Establish accessibility objectives and make an overall plan for activities and resources.	Ensure early appropriate interlocks among all disciplines in the UCD team.	Negotiate issues affecting accessibility and usability.	Make final accessibility assessment. Document accessibility status
UI programming	Understand requirements of accessibility checklist and plan resources for implementation and test	Work with HCI designer to determine technical feasibility and effort for accessibility features.	Implement accessibility features. Work with HCI designer and visual designer on technical issues.	Test code with regard to high-contrast settings and use of screen readers
HCI design	Understand requirements of accessibility checklist and identify need for specialized designs.	Create concepts for presentation of information, navigation, and interaction.	Work with visual designer and UI programmer to resolve issues found during initial evaluations	Understand evaluation results and map into final design change proposals.
Visual design	Identify product externals and define visual identity/brand elements.	Create design concepts, mock-ups, storyboards using visualization techniques; gather feedback from intended audience and stakeholders.	Finalize concepts considering technology requirements, applicable standards, and legal regulations.	Use simulation tools and user feedback to validate accessibility of visual product externals.
Information development design	Plan content of deliverables and determine accessibility features to be documented.	Create outline of the on-line help system and the manuals.	Write on-line help and documentation in accessible format.	Verify accessibility of on-line help and documentation.
User feedback	Understand test requirements of the accessibility checklist. Determine time and resources needed for evaluations.	Become familiar with accessibility test tools and assistive technologies. Inspect early designs using accessibility checklist.	Perform initial accessibility evaluations with recommended tools.	Perform accessibility tests with recommended tools.

(information such as object names and values not accessible). Again, a few instances were found where planned keyboard assignments had to be changed.

The final assessment took place during system verification test (SVT) when we repeated all test cases. At this time, only a small number of defects were found. Again, the UI programmer and the UI designer worked together to solve these issues. The specification for keyboard operation was finalized and could now be used by the information developer to document accessibility features.

Table 1 summarizes the responsibilities and activities of the multidisciplinary UCD team with regard to accessibility requirements.

The sections that follow describe the accessibility design features that were produced in this UCD process.

User interface design

In designing the user interface, we looked at alternative formats and interaction techniques.

Figure 1 The graphics view

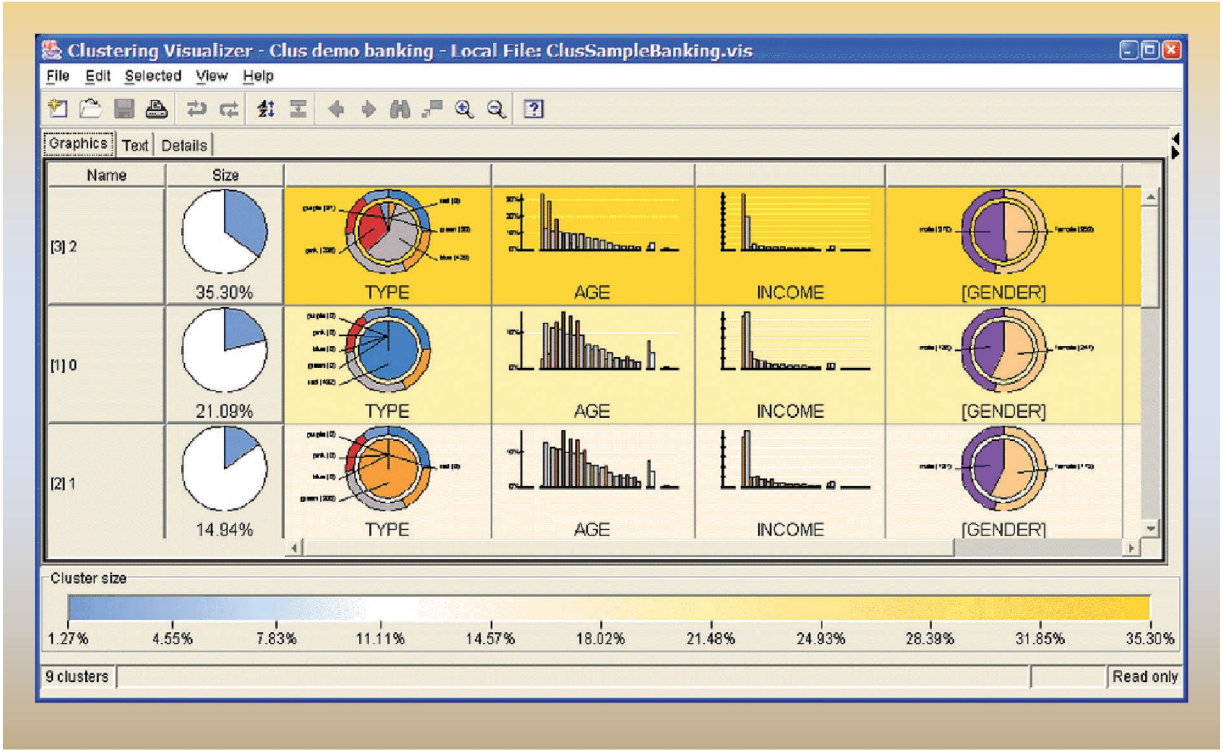
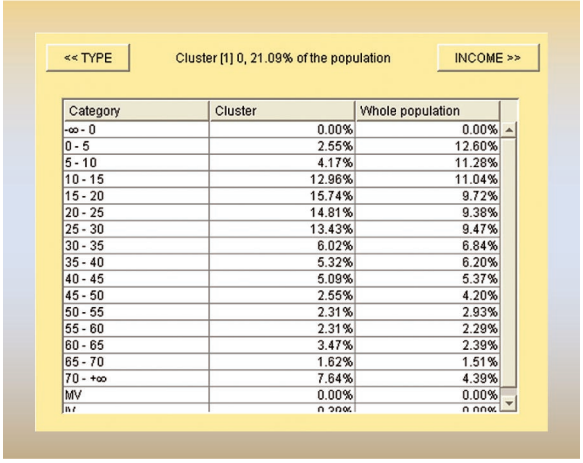


Figure 2 The tabular presentation of a chart in the graphics view



Category	Cluster	Whole population
-∞ - 0	0.00%	0.00%
0 - 5	2.55%	12.60%
5 - 10	4.17%	11.28%
10 - 15	12.96%	11.04%
15 - 20	15.74%	9.72%
20 - 25	14.81%	9.38%
25 - 30	13.43%	9.47%
30 - 35	6.02%	6.84%
35 - 40	5.32%	6.20%
40 - 45	5.09%	5.37%
45 - 50	2.55%	4.20%
50 - 55	2.31%	2.93%
55 - 60	2.31%	2.29%
60 - 65	3.47%	2.39%
65 - 70	1.62%	1.51%
70 - +∞	7.64%	4.39%
MV	0.00%	0.00%

Presenting information in alternative formats. Providing text labels for icons and graphics is a basic requirement for accessibility. It is fairly simple to implement and provide the information that is necessary for alternative output devices, such as speech or braille. However, this technique is suitable only for applications that use static graphics (icons or pictures). It does not work for applications that use dynamic graphical objects such as diagrams or charts.

Our solution to this problem is to provide alternative presentations. Each visualizer presents information not only in a graphical format (pie charts, bar charts, or nodes and arrows), but also in textual or tabular format. This variation makes the data that are shown by a graphic accessible in a format appropriate for assistive technologies.

The graphics view shown in Figure 1 is the primary view of a visualizer. It presents data in different types of charts and in two dimensions of color coding, foreground colors and background colors.

Alternatively, every chart can be displayed in a tabular presentation, making the chart data accessible to screen readers (Figure 2).

Figure 3 The text view

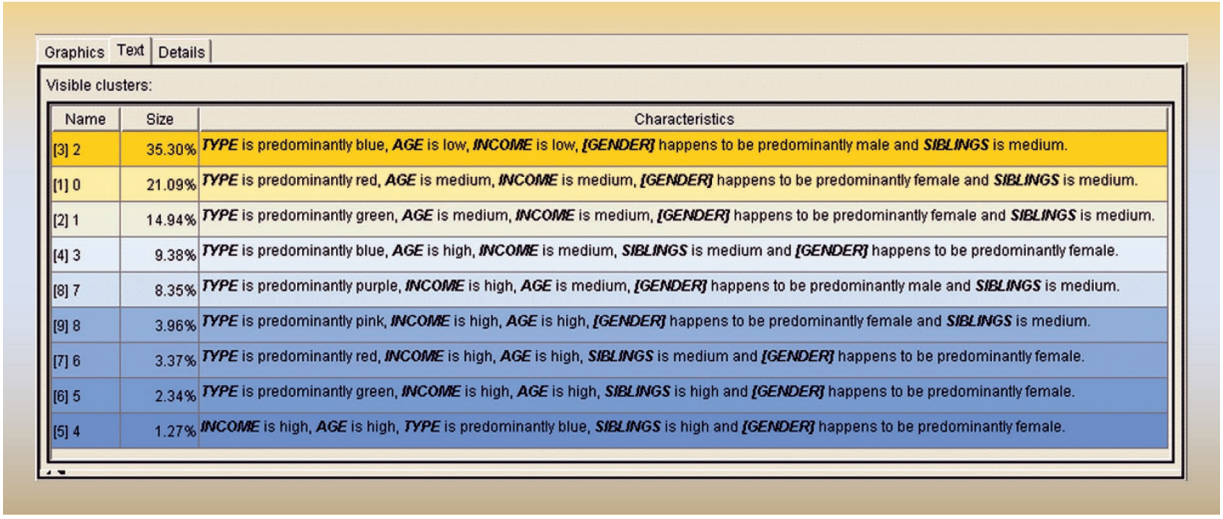
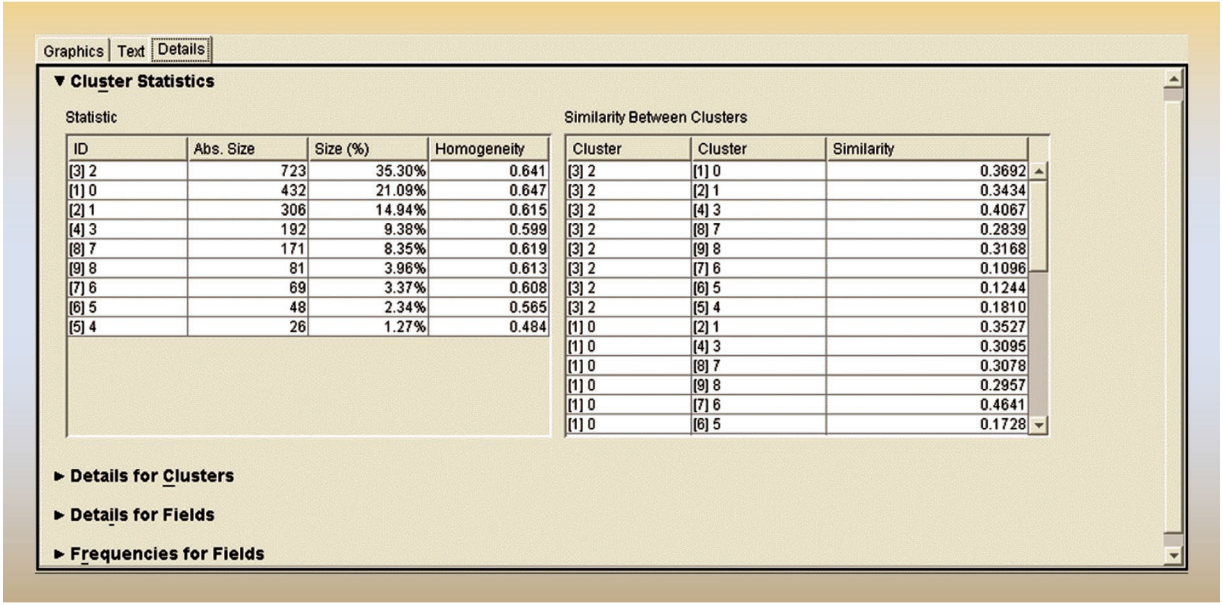


Figure 4 The details view

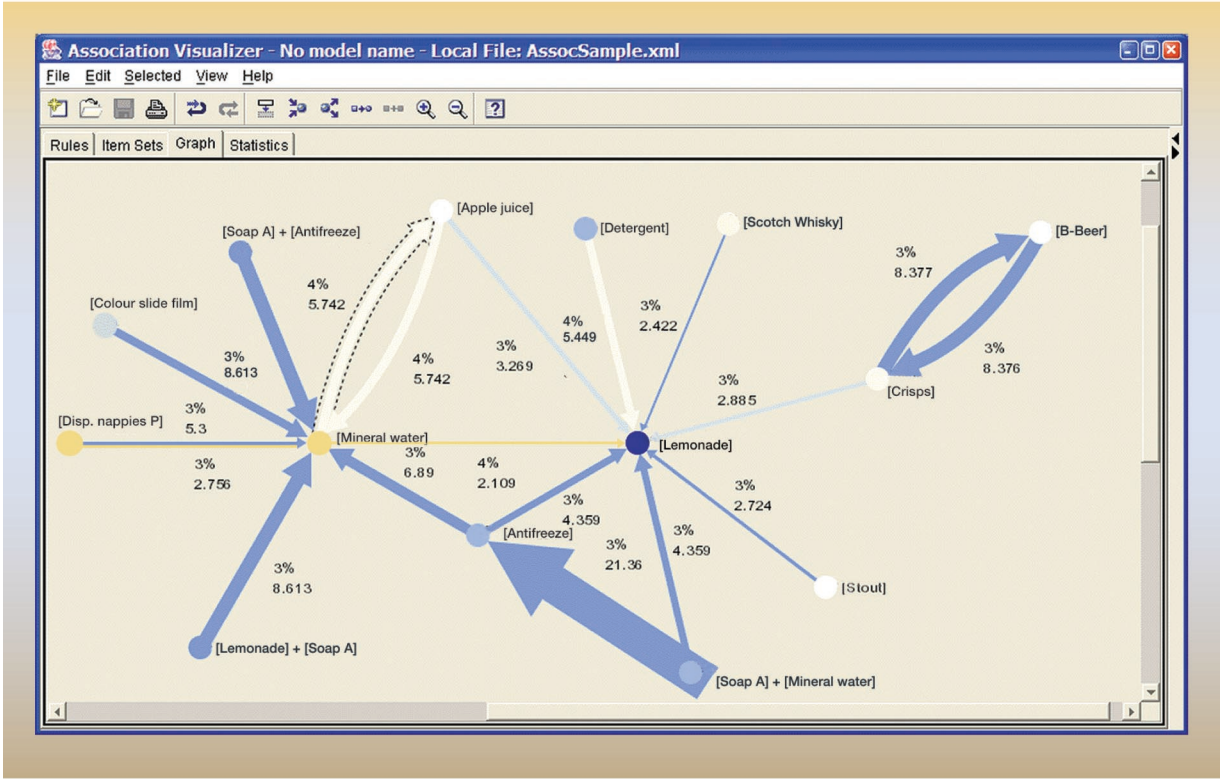


In the text view, the most important characteristics of each cluster are described in sentences (Figure 3). This format is especially useful for individuals who rely on speech output, because this representation reduces the cognitive load by summarizing the most important characteristics of the underlying data.

Finally, the details view shows all numerical data of the mining model in tables (Figure 4).

Interaction techniques. The associations visualizer uses graphs that not only display information, but also allow many interactions. For example, users can

Figure 5 The graph view of the associations visualizer



move elements (nodes or arrows) or select particular elements to display only those elements that are of interest.

How can this activity be done without using a mouse? We designed a simple mechanism for keyboard navigation and selection. As in a table or a list, each element can receive the keyboard focus. Figure 5 shows that the arrow pointing from [mineral water] to [apple juice] has the keyboard focus (indicated by a dotted line). Users can navigate in this graph as follows:

- To move the keyboard focus across the arrows that are connected to [mineral water] in a clockwise direction, press the Right Arrow key or the Down Arrow key.
- To move the keyboard focus counterclockwise, press the Left Arrow key or the Up Arrow key.
- To move the focus to the node [apple juice], press the End key.
- To move the focus back to [mineral water], press the Home key.

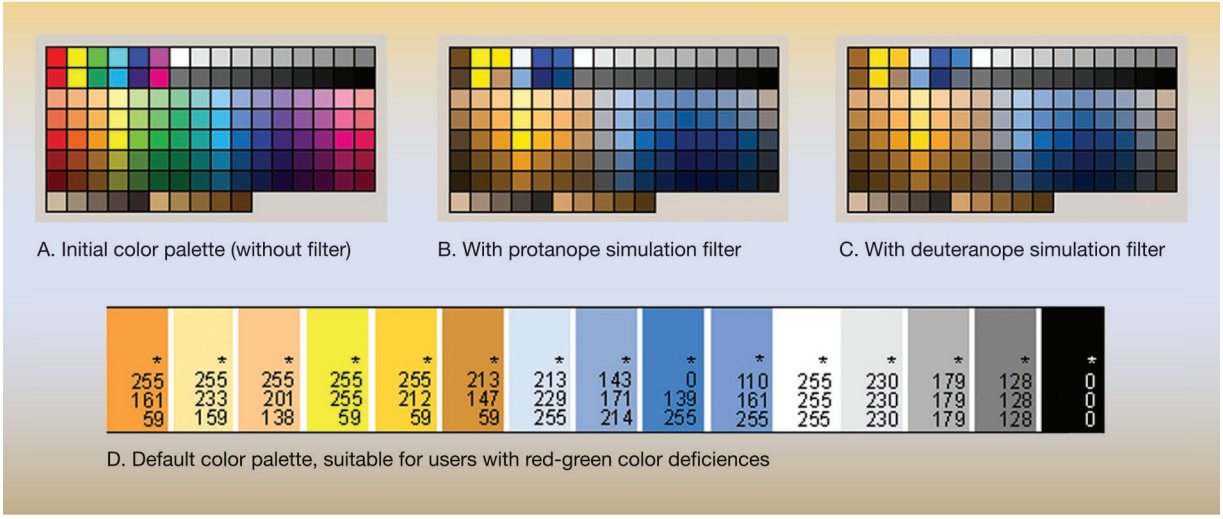
Elements in this graph are selected or deselected with the Space bar. Together with a few additional keyboard functions, association graphs are fully enabled for mouseless operation.

Besides special presentation and interaction techniques, choosing appropriate colors is a critical aspect. Color selection is the task of the visual designer.

Visual design

An important task of visual design for accessibility is to define a scheme for color coding that accommodates the needs of individuals with color-vision deficiencies. About 8 percent of the male population and less than 1 percent of the female population are afflicted by anomalous color perception or color blindness.[14] The most common forms of color-vision deficiencies are related to difficulties in distinguishing red and green colors (protanopia, deuteranopia). As Meyer and Greenberg point out,[14] these forms have similar characteristics in how they

Figure 6 Application of deuteranope and protanope filters to an RGB color palette



cause confusion in colors, so that it is possible to find a set of colors that is suitable for almost all color-defective users.

Using a color-defect simulation tool to identify suitable colors. To select colors that are distinguishable for users with red-green deficiencies, visual designers have to know how these individuals perceive different colors. Brettel et al.[15] have developed an algorithm for simulating anomalous (dichromatic) vision for persons with normal (trichromatic) color perception. A number of color simulation tools are available that are based on this method. We used VisCheck[16] to construct a color palette.

Applying the deuteranope and the protanope filters to a red/green/blue (RGB) color palette led to the results shown in Figure 6. This set of suitable, "filtered" colors was further refined according to the following requirements: distinguishability, visual appeal, and readability of foreground text. The resulting palette of selected colors is shown in Part D of Figure 6 (numbers indicate the RGB values for the colors). In cases where this default color set is not suitable, users can still turn to the standard Java color chooser dialog to modify all color coding according to their preferences.

For contexts where color coding is not desirable at all—for instance, complete color blindness—we provided an option to code charts with monochrome textures. This feature is also useful when charts are printed in black and white.

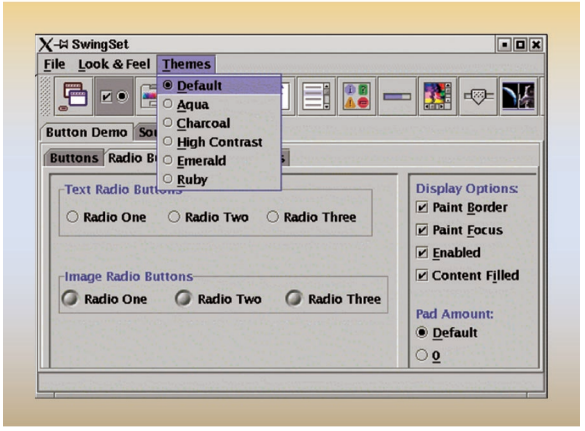
Software development

So far, we have described how we addressed accessibility issues with regard to user interface design. We will now look at the solutions that programmers developed to meet two important accessibility requirements: high-contrast settings and keyboard handling.

Accessibility schemes and pluggable look and feel. Users with visual impairments need specific fonts or specific color schemes with a high contrast between text and background to be able to read information on the screen. Therefore, the IBM Java Accessibility Checklist[7] specifies that high-contrast settings must be supported by all user interface objects.

On Microsoft Windows operating systems, users can modify the display settings to use specific color schemes and fonts. They can also use predefined schemes that are designed for users with visual deficiencies. By selecting a scheme in the display settings of the desktop, the palette of system colors and system fonts is modified. This modification usually has an impact on the appearance of all the native applications of Microsoft Windows.

Figure 7 Default theme



Java applications work differently because they do not rely on the graphical libraries that are used by the native applications. Swing-based Java applications can use their own pluggable look and feel (PLAF) that works independent of the underlying operating system. The standard Java platform includes the following PLAFs:

- The Microsoft Windows PLAF emulates the appearance of the Microsoft Windows operating system.
- The Motif PLAF emulates the appearance of a Motif application.
- The Metal PLAF provides its own look and feel that is independent of any existing operating system.

Programmers can force the application to use a particular PLAF or allow users to select the PLAF that they want to use.

Applications that use the Windows PLAF of Java normally inherit the color settings of the underlying operating system automatically. This fulfills a part of the IBM Java Accessibility Checklist. However, the font settings are not properly inherited if versions earlier than Java 1.3.1 are used. In this case, if Microsoft Windows is set up to use a high-contrast color scheme with large fonts, the Java application uses the correct color scheme, but it does not change from the standard small fonts to the large fonts. This behavior makes it inaccessible for users with a vision deficiency.

Furthermore, inheriting the color settings of the underlying operating system works only with the Windows PLAF, which can be used only on a Microsoft Windows operating system. For any other operating system, such as Linux**, AIX*, or the Solaris** Operating Environment, there is no way to inherit the settings of the underlying operating system.

The IBM Java Accessibility Checklist recommends the creation of one's own PLAF and the use of native methods to obtain the system preferences if the Windows PLAF cannot be used. Unfortunately, this is difficult to realize. Even if work is derived from an existing PLAF, the creation of one's own PLAF is a huge task that is beyond the scope of a development project. Furthermore, using native methods breaks the operating system independency for which Java was designed. It also means that one's own PLAF must be created for each different operating system or for each different windowing system, because some operating systems can use different windows managers, each of them having its own API and its own way to define a color scheme.

IBM DB2 Intelligent Miner Visualization had to be delivered as a stand-alone product for five different operating systems and as an applet for any Java-enabled browser. Therefore, these technical recommendations were not acceptable to us. We had to find another solution to enable users with deficiencies in vision to use the product, even on a non-Windows operating system.

We decided to exploit a feature of the Java Metal PLAF. It allows the use of different color and font schemes. The SwingSet demos that are delivered with the Java software development kit demonstrate this feature. Figure 7 shows the default theme, and Figure 8 shows the high-contrast theme.

It is a relatively easy task to develop a software application that:

- Allows users to select among different PLAFs
- Additionally provides one or more PLAFs that are based on the Java Metal PLAF
- Provides several high-contrast color schemes

It represents an alternative for users with visual impairments who are working on a non-Windows operating system. In fact, it does not automatically use the settings that users might use for their desktops. However, users can set up the software application with similar settings, and this compromise is acceptable.

Building a customizable high-contrast scheme. To ensure flexibility, we decided to build a generic PLAF that inherits the functionality of the standard Java Metal PLAF and that additionally can be customized by editing an external resource file in text format. In this way, the information about the colors and the font size are not compiled in the PLAF itself, but are read from a text file that can be modified with a simple text editor.

The implementation of a generic PLAF is relatively easy because most of the functionality is inherited from the standard Metal PLAF. The details about the implementation are beyond the scope of this paper. For more information, see the paper by Saillet.[17] From this paper, developers can download the code of the PLAF to use or modify for any kind of swing-based application.

The class name of the customizable PLAF that can be downloaded from the Saillet paper is "HighContrastLAF." A search can be made for the information about the color scheme and the font size in the file "HighContrastLAF.properties." The following example shows the external resource file for the black-on-white look-and-feel, large fonts. The font size for this PLAF is set to 30 (line 5). The icons are scaled to 250 percent of their original size (line 3). The basic colors are set to black and white (lines 6–15). Figure 9 shows a sample dialog with this PLAF.

```
HighContrastLAF.properties
1   name = High Contrast Look And Feel
2   description = Black on white, large
    fonts
3   iconMagnificationFactor = 2.5
4   fontName = Dialog
5   fontSize = 30
6   backgroundColor = 000000
7   foregroundColor = FFFFFF
8   primaryColor1 = FFFFFF
9   primaryColor2 = 000000
10  primaryColor3 = 000000
11  secondaryColor1 = FFFFFF
12  secondaryColor2 = 808080
13  secondaryColor3 = 000000
14  selectionForeground = 000000
15  selectionBackground = FFFFFF
```

In the same way, a high-contrast look and feel with white text on black background and normal font size can be created just by providing another resource file with appropriate color settings.

Figure 8 High-contrast theme

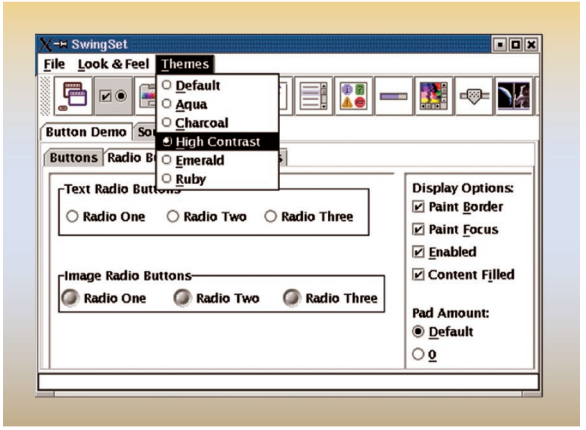
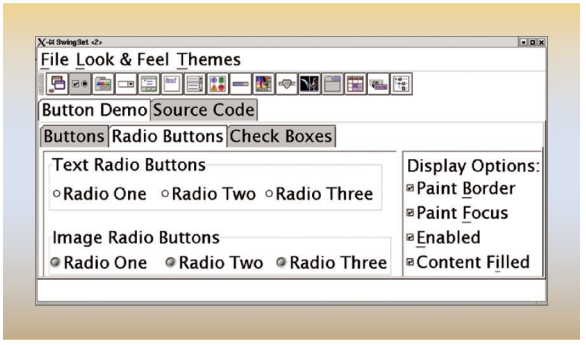


Figure 9 Black text on white background, large font



The PLAF can be used for any swing application by invoking the following line in the main method before the first graphical object is created:

```
UIManager.setLookAndFeel
 ("HighContrastLAF");
```

For IBM DB2 Intelligent Miner Visualization, we implemented an entry in the Preferences dialog that lets the user select a PLAF. If such a dialog cannot be implemented, an easier solution might be to use the system properties to allow users to specify the PLAF by starting the program from the command line.

```
String plaf = System.getProperty
 ("plafName");
if (plaf!=null) try {
  UIManager.setLookAndFeel(plaf);
```

```
} catch (Exception e)
{ System.out.println("Error loading
 PLAF "+plaf+":"+e);}
```

Users can select a specific PLAF by using the following syntax in the command line when they are starting the application:

```
java -DplafName=HighContrastLAF
 -classpath . . . . Main . . .
```

Aspects to consider during the development of the application. The previous subsection presented a generic PLAF that can be modified by editing a resource file so that an application fulfills the accessibility requirements concerning the high-contrast settings. There are, however, rules to respect to ensure that the application whose look and feel is customized really uses the settings that are defined by the PLAF. These rules are basically simple to follow if they are adhered to from the beginning of the project. This statement sounds as though it should be self-evident; however, it is often forgotten during the implementation.

Rule 1—Do not use hard-coded colors or fonts. Hard coding is a mistake that often happens when the background of a screen view, or "widget," must be set to white or its foreground to black. In this case, product developers tend to program something like the following examples, forgetting that other PLAFs might have a completely different color scheme:

```
someWidget.setBackground(Color.white);
someWidget.setForeground(Color.black);
```

If a PLAF uses white text on a black background, this might result in widgets that have both their foreground and their background set to white or black. Or it results in widgets with a completely different color scheme from the rest of the application.

The same applies to the font. Using the following command to set the font of a widget to boldface might result in an inconsistency if a PLAF with a large font is used:

```
someWidget.setFont(new Font("Dialog",
 Font.BOLD, 14))
```

All widgets would use large fonts except the specific widget.

When colors or fonts are needed, the correct technique is to use the methods getColor(Object key) and getFont(Object key) of the class javax.swing.UIManager. These methods take a key that identifies a color or a font as a parameter, and they return the corresponding color or font as it is defined by the current PLAF. Unfortunately, the list of the available keys is not documented. However, the keys are included in the source code of the class javax.swing.plaf.basic.BasicLookAndFeel.

The keys that are available for the colors are listed in Table 2. The keys that are available for the fonts are listed in Table 3.

If the keys that are listed in Table 2 or in Table 3 are used, the correct technique to set the colors or font is:

```
myWidget.setBackground(UIManager.getColor
 ("text"));
myWidget.setForeground(UIManager.getColor
 ("controlText"));
myWidget.setFont(UIManager.getFont
 ("Button.font"));
```

If a specific font style or size is needed to highlight a control, an existing font must be derived, and a value relative to the normal font size must be used for the new size of the font. The following line creates a boldface font with a size 20 percent larger than the normal font size independent of the font size that the PLAF is using.

```
Font f = UIManager.getFont
 ("Button.font"); // reference font
myWidget.setFont(f.deriveFont
 (Font.BOLD, 1.2*f.getSize())) ;
```

Rule 2—Regular tests of how the application looks with a different look and feel. It is a good idea to regularly test the application with a different PLAF during the development phase. In this way, hard-coded colors or fonts can be detected and fixed faster as described in Rule 1. For the tests, use a PLAF with a completely different color scheme and font size than the PLAF that is used during development. The customizable PLAF presented earlier in this paper may be the one to use.

Table 2 Keys that are available for the colors

Key	Description
desktop	Color of the desktop background
activeCaption	Color for captions (title bars) when they are active
activeCaptionText	Text color for text in captions (title bars)
inactiveCaption	Color for captions (title bars) when not active
inactiveCaptionText	Text color for text in inactive captions (title bars)
inactiveCaptionBorder	Border color for inactive caption (title bar) window borders
window	Default color for the interior of windows
Menu	Background color for menus
menuText	Text background color
text	Text background color
textText	Text foreground color
textHighlight	Text background color when selected
textHighlightText	Text color when selected
textInactiveText	Text color when disabled
control	Default color for controls (buttons, sliders, etc.)
controlText	Default color for text in controls
controlLtHighlight	Highlight color for controls
controlShadow	Shadow color for controls
controlDkShadow	Dark shadow color for controls
scrollbar	Scrollbar background
info	Background color of the tool tips
infoText	Background color of the tool tips

Table 3 Keys that are available for fonts

Button.font	Label.font	PopupMenu.font	Table.font	TitledBorder.font
ToggleButton.font	List.font	OptionPane.font	TableHeader.font	ToolBar.font
RadioButton.font	MenuBar.font	Panel.font	TextField.font	ToolTip.font
CheckBox.font	MenuItem.font	ProgressBar.font	PasswordField.font	Tree.font
ColorChooser.font	RadioButtonMenuItem.font	ScrollPane.font	TextArea.font	
ColorChooser.font	RadioButtonMenuItem.font	Viewport.font	TextPane.font	
Label.font	Menu.font	TabbedPane.font	EditorPane.font	

Rule 3—Do not assume a size for fonts or the space required for text or widgets. This recommendation is not only valid for accessibility issues, but also for the globalization of the application. Because a PLAF can use any font size, it is impossible to know definitely how much space a widget or the text will need. This implies that no widget position and size should be hard-coded and that layout managers consequently should be used.

Keyboard handling

The Java swing library offers a rich set of graphical components that are easy to use and appropriate for most standard applications. These components not only comprise the graphical part, but also provide the handling, the navigation, and the accessibility for the functionality.

In complicated software products, very often it is not sufficient to use standard components. Either extensions to existing components from a library or a new component must be written. If a new component is written, the accessibility of the new functionality must be ensured, and it also must be ensured that the handling and navigation are working correctly.

During the development of IBM DB2 Intelligent Miner Visualization, we had to deal with various problems. The following subsections describe two of them.

Mixing standard components with newly created components. We embedded the newly created component association graph that is shown in Figure 5 into a scroll pane that is provided by the Java swing library. The association graph consists of two graph-

Figure 10 Example of action map and input map

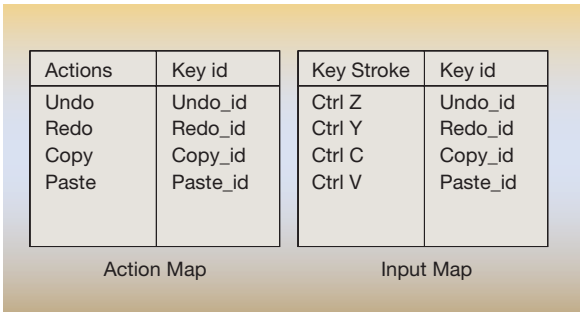
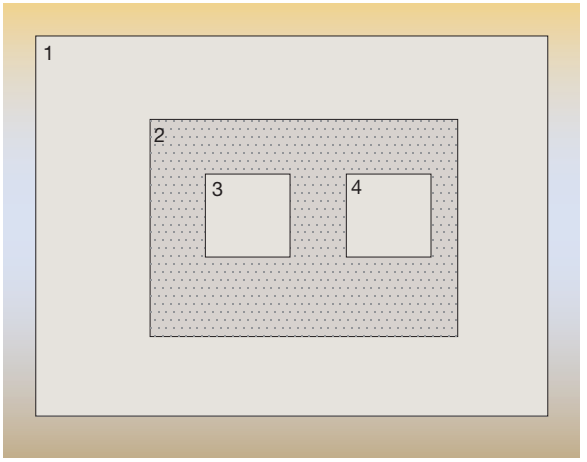


Figure 11 The WHEN_FOCUSED scope



ical elements: circles and arrows. The circles and the arrows are used to represent associations between two elements. These elements can be moved manually in the graph. If the canvas, or screen area, is too small, a scroll bar can be used to scroll to the part that is not visible initially.

Because the association graph is a new component, its navigation keys must first be defined. It must be possible to navigate as to where the focus is placed in the graph, and to select and move the elements. We defined keys for the following actions:

- Focus navigation—Arrow keys, Home key, and End key
- Selecting elements—Space bar
- Moving elements—Shift key and Arrow keys

For more information, see the earlier subsection on interaction techniques.

From an ease-of-use point of view, it is vital to provide intuitive focus navigation. Therefore, we defined the Arrow keys to move the point of focus. However, this generates a conflict with the standard scroll bar component. As the default, the Arrow keys are used to move the scroll bars. As a result, the scroll bar could only be moved with the Page Up key and the Page Down key. The solution from the ease-of-use viewpoint is to redefine the scroll bar keys to be the Shift Arrow keys, because focus navigation is used more often than the scroll bar and therefore is more important.

It is good practice not to hard-code the key binding but to use a more flexible approach.

In the Java language, keyboard actions are managed with the help of input and action maps. An action map is a structure that stores all the possible actions that can be executed with a key identifier (ID), so that the actions can be retrieved easily and assigned to any key stroke combination or menu item. In contrast, an input map is a structure that maps a specific key stroke combination with a key ID of an action stored in an action map (Figure 10).

Every swing component owns one action map containing all the actions that can be executed on this component, as well as three input maps. Each of the three input maps is used in a different range, or scope, where the action should take place. That means an action can be executed in a different context. Every scope is identified by a constant.

Only the key events coming from the component owner of the input map are evaluated. Key events coming from the parents or children of the component are ignored. If a key stroke is registered in the WHEN_FOCUSED scope of Component 2, the key stroke only has an effect if the focus is on Component 2. If the focus is on one of the other components, it has no effect. The placement of the focus is indicated by the area that contains the texture, as shown in Figure 11.

The WHEN_ANCESTOR_OF_FOCUSED_COMPONENT scope is wider than the WHEN_FOCUSED scope. Key events coming from the component owner of the input map or from one of its descendants are evaluated. Events coming from one of its parents are ignored. If a key is registered in the scope of Com-

ponent 2, the key stroke only has an effect if Component 2 or one of its children, Component 3 or Component 4, have the focus on them. In this case, the placement of the focus is indicated by the area that contains the texture in Figure 12.

The WHEN_IN_FOCUSED_WINDOW scope is the widest. The key events coming from all components contained in the same window as the owner of the input map are evaluated. If a key is registered in the WHEN_IN_FOCUSED_WINDOW scope of Component 2, the key stroke only has an effect if the window of Component 2 is where the focus lies or contains the component that has the focus. The focus is indicated by the area that contains the texture shown in Figure 13. This input map is commonly used for mnemonics or accelerators, which need to be active regardless of where the focus is in the window.

When a key is pressed on the keyboard, the key stroke is identified in the input map, and the corresponding key ID is used to look up the action in the action map. If the key ID is found in the action map, the action is executed. The maps have a default initialization depending on the selected look and feel (Figure 14).

To solve the problem with the scroll bar described earlier, the Shift Arrow key binding is inserted into the input map of the scroll pane in the WHEN_ANCESTOR_OF_FOCUSED_COMPONENT scope. The keys can be used when the focus is on the scroll bar and in the whole association graph.

Extending standard components to a new component. We extended a standard component to realize a new functionality. The accessibility API is already implemented in the standard component but must be adapted for the new functionality. For a detailed description, see the Java accessibility utilities.[13]

As a standard component, a JPanel is used. (A Jpanel is a Java class that is a generic lightweight container.) For the application, an expandable panel was needed. The panel has a title and can be expanded or collapsed to show or hide information as shown in Figure 4.

To support the accessibility API, extra code is needed for the new functionality. Every component must return its accessibility context. Because the expandable panel inherits most of the accessibility features from the JPanel, only the changed methods must be implemented. The two important items for the accessibility tools are the name of the component and its current state. For example, a screen reader is able to inform a blind person about the focused component and its state. But the user also needs information about what can be done with the component. Thus, the possible actions for the components must be listed. Here new functionality is added to a component, and therefore, new actions are defined. The expandable panel has two actions: expand and collapse. (A detailed reference to accessibility can be found in Reference 8.)

Figure 12 The WHEN-ANCESTOR_OF_FOCUSED_COMPONENT scope

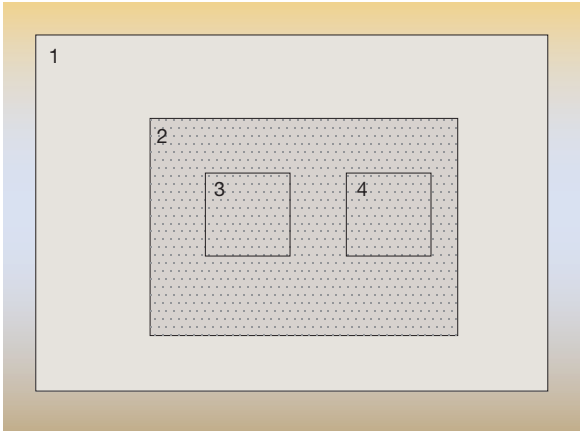


Figure 13 The WHEN_IN_FOCUSED_WINDOW scope

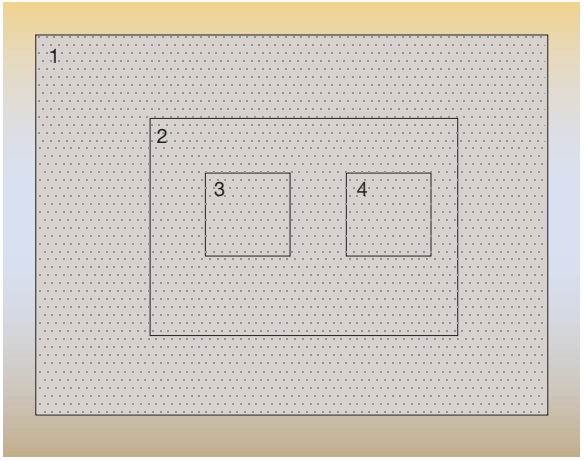
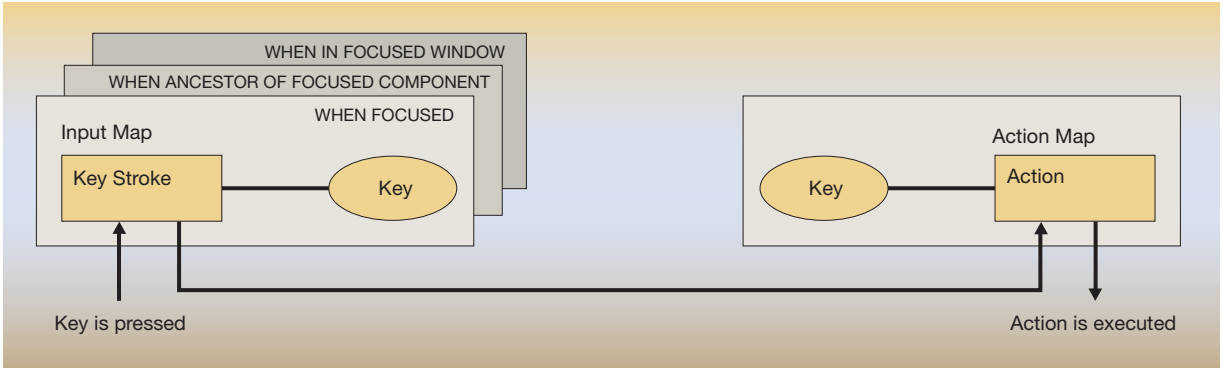


Figure 14 Example of default initialization for input map and action map



Information development

Last, but not least, product-related documentation such as user manuals and on-line help must be produced in an accessible format. To provide accessible documentation, the following requirements must be met:[7]

- Provide documentation in an accessible format.
- Provide information on all accessibility features including keyboard access.
- Test for accessibility using available tools.

Developing accessible documentation. For the manuals, at least one of the available accessible electronic formats must be provided, for example, Hypertext Markup Language (HTML) or Portable Document Format (PDF).

- Manuals in HTML format must follow the *IBM Web Accessibility Checklist*.
- Manuals in PDF format must be created according to the Adobe Systems, Inc. guidelines for accessibility.

Documenting accessible features. The accessible features of a product must be described in the manuals and in the on-line help system. In a user's guide or in an administration guide, the accessible features are described in a separate chapter in the body of the book or in the appendix. In an on-line help system, the accessible features are described in a separate topic that includes subtopics on the different accessibility features.

Furthermore, the following keywords must be added to the index of the manual and in the index of the on-line help system: accessibility, disability, keyboard, shortcut keys. These words must also be searchable.

In the IBM DB2 Intelligent Miner Visualization documentation, the following basic accessibility tasks are covered:

- Navigating by using a keyboard
- Customizing fonts
- Customizing colors
- Providing alternative descriptions for graphics that can be read by a screen reader

Finally, all product documentation must be tested by using screen-reading tools.

Concluding remarks

Developing software that is accessible to users with disabilities requires multidisciplinary teamwork and a User-Centered Design approach. Like usability, accessibility is more than a simple add-on. For applications that are highly graphical, specialized forms of presentation and interaction may be required.

Early in the planning phase, the development team must carefully determine which skills, methods, and tools are needed to design, implement, and test accessibility features because these activities can occupy significant time and resources. For this highly graphical product, the effort for coding and testing

was approximately 20 percent greater. It must be noted, however, that this effort applied to the first-time implementation and that it included the learning curves of all team members; the effort for subsequent developments is expected to be significantly lower.

The design solutions presented in this paper were aimed at providing a basic enablement for accessibility; that is, disabled users are not excluded from using the application. However, there is still much room for improvement toward an "intelligent user interface" that is able to adapt itself to the characteristics and behavior of the user.[18] For instance, instead of users switching manually from graphical to textual modality, the system could present the most appropriate modality automatically, select and filter the content accordingly, and so forth.

Accessibility checklists and coding guidelines give essential guidance to programmers; nevertheless, prevailing development environments may not include support for all accessibility problems. The examples for high-contrast settings and keyboard handling, as described in this paper, may help solve some of these issues.

Acknowledgment

The authors would like to thank Andrea Snow-Weaver and Phillip Jenkins from the IBM Accessibility Center for their invaluable advice and help.

*Trademark or registered trademark of International Business Machines Corporation.

**Trademark or registered trademark of Sun Microsystems, Inc., Microsoft Corporation, or Linus Torvalds.

Cited references

1. H. S. Kaye, *Computer and Internet Use Among People with Disabilities*, Disability Statistics Report (13), U. S. Department of Education, National Institute on Disability and Rehabilitation Research, Washington, DC (2000).
2. C. A. Meares and J. F. Sargent, *The Digital Work Force: Building Infotech Skills at the Speed of Innovation*, U.S. Department of Commerce, Technology Administration, Office of Technology Policy, Washington, DC (1999).
3. Laws, Standards, and Regulations, IBM Corporation, http://www.ibm.com/able/laws/index.html.
4. Web Accessibility Initiative, World Wide Web Consortium, http://www.w3.org/WAI.
5. Electronic and Information Technology Accessibility Standards, U.S. Architectural and Transportation Barriers Compliance Board, http://www.access-board.gov/sec508/508standards.htm.
6. E. Bergman, "Towards Accessible Human-Computer Interaction," *Advances in Human-Computer Interaction*, Vol. 5, J. Nielsen, Editor, Ablex Publishing Corporation, Norwood, NJ, now available from Intellect Ltd., Exeter, UK (1995).
7. Developer Guidelines, Software Accessibility, IBM Corporation, at http://www.ibm.com/able/guidelines.html.
8. R. S. Schwerdtfeger, *IBM Guidelines for Writing Accessible Applications Using 100% Pure Java*, Developer Guidelines, IBM Corporation, at http://www.ibm.com/able/guidelines/java/snsjavag.html.
9. *Human-Centered Design Processes for Interactive Systems*, ISO 13407, International Organization for Standardization, Geneva (1999).
10. K. Vredenburg, S. Isensee, and C. Righi, *User-Centered Design: An Integrated Approach*, Prentice Hall, Upper Saddle River, NJ (2002).
11. C. Stephanidis, D. Akourmianakis, M. Sfyrakis, and A. Paramythis, "Universal Accessibility in HCI: Process-Oriented Design Guidelines and Tool Requirements," *Proceedings of the 4th ERCIM Workshop on User Interfaces for All*, Stockholm (1998).
12. DB2 Intelligent Miner for Data, IBM Corporation, at http: www.ibm.com/software/data/iminer/fordata/.
13. Java Accessibility Utilities, Sun Microsystems, Inc., at http: java.sun.com/products/jfc/jaccess-1.3/doc/.
14. G. W. Meyer and D. P. Greenberg, "Color Defective Vision and Computer Graphics Displays," *IEEE Computer Graphics and Applications* **8**, No. 5, 28–40 (1988).
15. H. Brettel, F. Viénot, and J. D. Mollon, "Computerized Simulation of Color Appearance for Dichromats," *Journal of the Optical Society of America A* **14**, No. 10, 2647–2655 (October 1997).
16. Simulation of Colorblind Vision, http://vischeck.com.
17. Y. Saillet, *Enhance the Accessibility of Your GUIs: Build a Customizable Cross-Platform Look and Feel for Visually Impaired Users*, developerWorks, IBM Corporation (July 9, 2003), http: //www.ibm.com/developerworks/java/library/j-customlaf/.
18. M. T. Maybury, "Intelligent User Interfaces for All," in *User Interfaces for All*, C. Stephanidis, Editor, Lawrence Erlbaum Associates, Mahwah, NJ (2001), pp. 65–80.

Accepted for publication July 10, 2003.

Dirk Willuhn *IBM Germany Laboratory, Schoenaicherstrasse 220, 71032 Boeblingen, Germany (dwilluhn@de.ibm.com).* Mr. Willuhn, a senior usability engineer, joined IBM Germany as an HCI and user research specialist in 1988. He has been a UCD team lead and user interface designer for a number of IBM products, including workflow management, database performance monitoring, data mining, and financial messaging. As a member of the IBM Corporate UCD Advisory Council and German DATech organization, he contributes to the development of procedures and methods for usability testing and User Engineering. He received his diploma in psychology from the University of Tuebingen, Germany.

Carsten Schulz *IBM Germany Laboratory, Schoenaicherstrasse 220, 71032 Boeblingen, Germany (schulzc@de.ibm.com).* Mr. Schulz, a software developer, joined IBM Germany as a software engineer in 2000. He has been working in data-mining product development as a developer and team lead for visualization of data-mining results. He received his diploma in computer science from the University of Braunschweig, Germany.

Lieselotte Knoth-Weber *IBM Germany Laboratory, Schoenaicherstrasse 220, 71032 Boeblingen, Germany (lkw@de.ibm.com).* Mrs. Knoth-Weber is currently a senior information developer, having joined IBM Germany as a management assistant in 1970. She was a management assistant in various IBM departments such as quality assurance, controlling, and plant management of the semiconductor manufacturing plants in Sindelfingen and Hannover, Germany. From 1988 until 1990 she was manager in the Central Administration department. In 1990 she joined the IBM Software Laboratory as an information developer. She has been team lead for various IBM projects, including data mining and content management.

Stephan Feger *IBM Germany Laboratory, Schoenaicherstrasse 220, 71032 Boeblingen, Germany (sfeger@de.ibm.com).* Mr. Feger currently is an advisory visual designer and has worked as a visual designer in IBM Software Solutions Development, Boeblingen since 1995, driving the visual design direction for products such as MQSeries® Workflow, Intelligent Miner for Data, and DB2 Performance Monitor. He joined the IBM Boeblingen Development Laboratory in 1984 and served as an industrial designer for a wide range of IBM hardware products until 1994. He received his diploma in industrial design from the State Academy of Fine Arts, Stuttgart and is a PMI- (Project Management Institute®) certified Project Management Professional (PMP).

Yannick Saillet *IBM Germany Laboratory, Schoenaicherstrasse 220, 71032 Boeblingen, Germany (ysaillet@de.ibm.com).* Mr. Saillet is a software engineer and joined IBM Germany as a software developer in 1998. He first worked for IBM Learning Services as a software engineer in several distributed learning projects. He joined the IBM Boeblingen Laboratory in 2000 and has since been active in the development of DB2 Intelligent Miner products. He received his masters degree from the ESSTIN (Ecole Supérieure des Sciences et Technologies de l'Ingénieur de Nancy) in Nancy, France.