

How Can Java Be Made Blind-Friendly

Norbert Markus¹, Zoltan Juhasz², Gabor Bognar², and Andras Arato¹

¹ KFKI Research Institute for Particle and Nuclear Physics, Laboratory of Speech Technology for Rehabilitation, Konkoly-Thege u. 29-33, 1121 Budapest, Hungary
{markus, arato}@mail.kfki.hu

² Veszprem University of Pannonia, Dept. of Information Systems,
Egyetem u. 10, 8200 Veszprem, Hungary
juhasz@irt.vein.hu

Abstract. The widely used and highly popular Java programming language is proved to be a great tool for developing platform independent applications. Everyday users mostly encounter them when using portable devices (mobile phones, PDAs, etc). However, the ordinary Java applications are inaccessible for the blind in general. Even the most used screen readers can only be enabled to handle GUI elements of a Java application by an additional adaptation package (e.g. access bridge for Jaws). Even with this, only a portion of existing Java programs that use swing classes may be made partially accessible for the blind. The solution offered eliminates the need of any screen reader.

Keywords: Accessibility for the Blind, Java programming, Education.

1 Introduction

The MOBILE Slate Talker (MOST) [1] - developed by the MOST consortium (consisting of the institutions listed in the authors section of this article above) - is a Java powered blind-friendly environment mainly used on PDA pocket computers. It provides the most essential inbuilt applications and a plug-in loading capability to load any number of Java plug-ins that may be developed by anyone. In these plug-in programs, the user interface is built of special blind-friendly elements (Java classes) provided in the "most.ui" package. These elements are near equivalents in functionality of the ordinary `awt` and `swing` classes used for graphical interfaces. The striking difference is that the design of the MOST framework deliberately refrains from using the graphical output. Instead, short audio signals and synchronized TTS announcements are used as feedback to user interaction. In addition, the PDA's touch screen is turned into a digital Braille slate allowing character entry with the user's fingertips.

The design keeps to the paradigm that all functions, applications and settings of the MOST framework must be accessible by the use of the four arrow keys. This does not exclude the availability of alternative shortcuts but there must always be a way through the menu tree to all places in the MOST system. This is an important recommendation to any programmers planning to prepare a MOST plug-in.

The MOST framework is not a screen reader and the user needs no additional screen reader installed to operate it. Instead, the stand-alone MOST system contains

integrated TTS, Braille entry and easy menu navigation capabilities that may be exploited by the plug-in programs.

Thus with a little understanding of the nature of blind computer usage, programmers may easily produce fully accessible Java programs that can be loaded as plug-ins and immediately used in the MOST system. Useful plug-in programs resulted from previous student degree projects by sighted contributors with standard Java programming expertise clearly indicate the success of this approach.

On the other hand, Java programs including MOST plug-ins may easily be developed by visually impaired persons and this has opened a way for blind programmers to develop applications for blind users. Many of the currently available MOST plug-ins come from visually impaired contributors.

2 Building Blocks of a MOST Plug-In

In the following sections, the most common predefined components of the MOST system will be presented that are essential for building a typical plug-in program.

1. A Toolkit for Creating Blind-Friendly Java Plug-Ins

The `most.ui` package offers a variety of blind-friendly user interface elements. Most times, the programmer does not have to deal with accessibility matters. Once the user interface is constructed for the plug-in program, the MOST framework will intend for well timed TTS and audio feedback and adequate menu handling and Braille/character entering modalities will be presented to the user whenever needed. In other words, all UI elements speak for themselves.

The `BFrame` class in the `most.ui` package serves as the container of each individual MOST application or plug-in. This provides a basic infrastructure to help build the application's user interface and a simplified access to system resources. It is similar to the conventional `Frame` class in `java.awt` or to `JFrame` in the `javax.swing` package.

2. Building Up the Acoustic Menu for the Application

The `MenuNode` class is the main building block for the application's menu structure. This menu tree is somewhat similar to `javax.swing.JTree` and it is constructed from elements that are instances or descendants of the `MenuNode` class. `MenuNode` is similar to `MenuComponent` or `Menu` in `java.awt` and to `JMenu` or `JMenuBar` in the `javax.swing` package (no need to differentiate in MOST)

The application's `BFrame` provides a field (`MenuNode menu`) that is the root of the program's menu whose children will be the elements of the program's main menu. Submenu (child) elements may be added to any `MenuNode` object by its `addNode(MenuNode item)` method and menu elements or entire menu branches may be removed at any time, so the menu tree of a plug-in program may be reshaped dynamically.

2.1 Summary of the Accessible Menu Elements

The `MenuNode` class is extended to implement all kinds of menu elements as follows:

`MenuItem` - selectable menu item for performing specific operations with a registered `MenuListener` that is fired when the item is selected. It is similar to `Menu` or `MenuItem` in `java.awt` with some inheritance differences, and to `JMenuItem` in the `javax.swing` package. Note that `java.awt.Button` or `javax.swing.JButton` may also be replaced with MOST's `MenuItem`, since the appearance considerations do not occur here.

`BList` - a single selection accessible list object with dynamically varying elements. Similar to `java.awt.List` and to `javax.swing.JList`. `javax.swing.JRadioButton` may also be replaced with `BList`.

`BTextField` - Accessible text field for presenting text data to the user and allowing character entry (in Braille). Similar to `TextArea`, `TextComponent` or `TextField` in `java.awt` and to `JTextArea`, `JTextField` in `javax.swing`. In MOST, text editing is always linear so there is no need to differentiate between single-line or multi-line modes. Plus, `BTextField` is a valid menu element while `BTextInput` that is not related to the application's menu may be "popped" under the user's fingertips whenever instant text entering is required. A `BTextField` may be made read-only, or to speak its content when it gets focus. This is especially useful for data fields with presumably short content.

`BTextProviderField` - a read-only text field for handling very large text files.

`BCheckBox` - A yes-or-no kind of menu component with programmer-defined option labels. Similar in function to `Checkbox` or `CheckboxMenuItem` in `java.awt` and to `JCheckBox` or `JCheckBoxMenuItem` in the `javax.swing` package. `javax.swing.JToggleButton` may also be substituted with `BCheckBox` or `MenuItem`.

The void `jumpTo(MenuNode node)` method of the `BFrame` class is used for jumping the user programmatically from one point to another in the menu structure.

Note: `java.awt.Dialog` or `javax.swing.JDialog` - may be realized as a submenu in MOST.

`javax.swing.JComboBox` - is not present in MOST, but it might be substituted in various ways.

2.2 Focus Handling in the Menu

Any focus change in the menu system may be captured and handled by overriding the `NavigationEventOccured` method of the application's `BFrame` that is called by the MOST system any time when the user triggers a navigation event (i.e., uses the arrow keys in the menu).

3 Other UI Components

The `most.ui` package contains further "acoustic" user interface elements that are not related with the application's menu structure. Instead, these can be opened programmatically and posed to the user when a specific intervention is needed (answering a question, choosing a file, etc.). Many of these components close automatically when the required user interaction is accomplished.

3.1 Summary of the Other UI Components

`BOptionPane` - a kind of popup "window" allowing the user's instant invention. Similar to `java.awt.Choice` and to `javax.swing.JOptionPane`.

`FileDialog`, `FileSaver` - For browsing and selecting folders and files for saving, opening, deleting ETC. Similar to `java.awt.FileDialog` and to `javax.swing.JFileChooser`.

`DateTimePicker` - for setting the date/time for a certain event or function

`BTextInput` - a popup field for presenting or entering text.

Note: `JPopupMenu` - may be realised by opening a child `BFrame` from the containing application (since the size and appearance does not matter in MOST).

`JProgressBar` - may be realised as a series of TTS announcements issued from another thread, or simply by providing a menu item that queries the current progress state.

3.2 Focus Handling in the Applications

Focus changes at the application level (i.e., when the user changes from one frame to another) are handled as follows.

When the user enters the application, the `BFrame`'s `windowOpened` method is called by the MOST framework, so this event may be captured and handled by overriding this method. Similarly, the `BFrame`'s `windowClosing` method is called when the user leaves the application, so this method may be overridden to handle that event. When a child frame is closed, the containing `BFrame`'s `childClosing` method is called that may also be overridden.

3.3 Handling Braille Dot, Key and Button Events

The individual key strokes, push button and Braille dot events may be handled by overriding the `keyPressed`, `keyReleased`, `keyTyped` and `brailleKeyPressed` methods.

4 TTS Messages

Textual information is usually sent to the TTS engine via the `most.system.Context` class, rather than printed on the screen: `Context.getTTSEngine().say("Any text")` generates a simple TTS announcement that will be heard immediately if possible. This announcement may be interrupted by a "stop" or another "say" call.

Subsequent TTS messages may be queued by `Context.getTTSEngine().join()` that suspends the execution of the current thread until the ongoing TTS announcement is completed or stopped.

An uninterruptible TTS message can be issued by `Context.getTTSEngine().sayComplete("Any text")` whereby a subsequent say request is automatically queued.

Its other variant `Context.getTTSEngine().sayComplete("Any text", false)` queues an uninterruptible message after the currently spoken TTS announcement rather than suppressing it.

A TTS announcement may be stopped by `Context.getTTSEngine().stop()` that also releases any active `join()` locks.

TTS rate, pitch, volume and exception dictionary settings may be queried and modified by further `Context` methods.

5 Internationalization of MOST Applications and Plug-Ins

The national localisation of a MOST plug-in may be accomplished through an unified locale handler of the `java.util.ResourceBundle` class. The application's menu names, warnings, etc. may be provided in language specific external files, from which the currently needed text items can easily be retrieved and used in the program instead of static text messages hard-coded in the program source.

For instance, British English text items can be provided for a clock plug-in in the 'clock_locale_res_en_UK.properties' file, which may be created simply with text editing skills. If the MOST framework's general locale property is set to British English, the system will attempt to load this property file with the above name. If it cannot be found, the default file 'clock_locale_res.properties' will be used instead.

Thus, once a MOST application or plug-in is prepared in this way, and is localised to a new language, there is no need to rewriting and recompiling the program, it is enough to copy the newly prepared locale file to the same folder that contains the program's class files and eventually pack them together in a jar file.

6 Example Program

The sample source code below presents a simple talking clock plug-in for the MOST system. For the sake of simplicity, this code is solely provided for the English speaking test users, rather than exploiting the available locale resources.

```
package hu.most.test.plugins;
import java.util.*;
import hu.vein.irt.braille.util.plugin.*;
import most.system.Context;
import most.ui.*;
public class Clock extends AbstractPlugin implements MenuListener {
    private MenuItem sayCurrentTime, sayCurrentDate;
    private MenuNode settings;
    private BTextField lastReport;
    private BCheckBox slowSpeed;
    private static final String MONTH[] = {"January", "February",
        "March", "April", "May", "June", "July", "August", "September",
        "October", "November", "December"};

    public Clock(BFrame parent) throws PluginException {
        //The first String parameter below will appear as the program's
        // name in the plug-ins menu
        super(parent, "Talking clock.", "Clock plugin");
    } //plug-in constructor
```

```

//This is called to generate the programs menu when the plug in is
// opened the first time
protected void createMenu() {
    //Parameters: String label, String description (spoken item name)
    sayCurrentTime = new MenuItem("SAY_TIME", "Current time.");
    //Register 'this' as MenuListener
    sayCurrentTime.addMenuListener(this);
    //Add the item to the plug-in's main menu
    menu.addNode(sayCurrentTime);
    sayCurrentDate = new MenuItem("SAY_DATE", "Today's date.");
    sayCurrentDate.addMenuListener(this);
    menu.addNode(sayCurrentDate);
    //Menu node for a submenu
    settings = new MenuNode("SETTINGS", "Settings.");
    //Braille text field for spelling the last reported date/time with
    // cursor
    lastReport = new BTextField("LAST_REPORT", "Last announcement.",
        "", "", context);
    lastReport.setEditable(false); //make it a read-only field
    //When entering the text field, its current content is spoken
    lastReport.setSayContent(true);
    //Add a text field as menu item to the submenu
    settings.addNode(lastReport);
    slowSpeed = new BCheckBox("SLOW_SPEED", "Speech rate.",
        new String[]
            {"Normal.", "Slow."}, true); //False would be the default
    //So it can acknowledge the change of mode
    slowSpeed.addMenuListener(this);
    settings.addNode(slowSpeed); //Add another item to the submenu
    //Add the submenu to the plug-in's main menu
    menu.addNode(settings);
}

//This is called when an item is selected for which 'this' is
// registered as MenuListener
public void itemSelected(Object item) {
    if (item.equals(sayCurrentTime)) {
        Calendar currentTime = Calendar.getInstance();
        say("It's " + currentTime.get(Calendar.HOUR_OF_DAY) + " "
            +currentTime.get(Calendar.MINUTE) + ".");
        return;
    }
    if (((MenuNode)item).getDescription().equals("Today's date. ")) {
        Calendar currentDate = Calendar.getInstance();
        say("Today's " + MONTH[currentDate.get(Calendar.MONTH)] + " "
            +currentDate.get(Calendar.DAY_OF_MONTH) + " "
            +currentDate.get(Calendar.YEAR) + ".");
        return;
    }
}
//All types of menu elements are descendants of MenuNode
if (((MenuNode)item).getLabel().equalsIgnoreCase("slow_speed")) {
    say((slowSpeed.getState() ? "Slow" : "Normal")
        + " mode enabled.");
}
}

private void say(String tosay) {
    //Retrieve current speech rate
    int rate = Context.getTTSEngine().getSpeed();
    //1 = a very slow rate
    if (slowSpeed.getState()) Context.getTTSEngine().setSpeed(1);
    try {
        Context.getTTSEngine().say(tosay);
    }
    //Wait until String tosay is spoken
}

```

```

        Context.getTTSEngine().join();
    } catch (InterruptedException intexc) {
        Thread.currentThread().interrupt();
    }
    if (slowSpeed.getState())
        Context.getTTSEngine().setSpeed(rate); //Reset original rate
    //Put the currently spoken info into the Braille text field
    lastReport.setText(tosay);
}
} //end of class Clock

```

7 Steps of the Plug-In Preparation

Following the steps below, the talking clock program may be compiled, packed, loaded and tested with the MOST system.

1. Save this program text to the file 'Clock.java'.

2. Compile the source by entering the following command line:

```
javac -source 1.3 -target 1.3 -classpath hu.jar;MOST.jar -d . Clock.java
```

The "-source" and "-target" options are only required if the plug-in is to be used on the PDA version of the MOST system. As a result, a file 'hu\most\test\plugins\Clock.class' will appear with all the necessary subfolders created on-the-fly.

3. Create the manifest file 'Clock.MF' consisting of the following line:

```
Plugin-class: hu.most.test.plugins.Clock
```

Terminate the line with a carriage return! This file will be packed with the Java class(es) into the .jar archive. This line tells the MOST system which file must be loaded from the archive as plug-in class.

4. Package the files by entering the following command line:

```
jar cvfm Clock.jar Clock.MF -C . hu
```

As a result, the file 'Clock.jar' will be created.

5. If tested on the PC version of MOST, the 'Clock.jar' file must be copied to the 'pc-most\plugins' directory, and to run the MOST program, one of the available 'start....bat' files in the 'pc-most' directory may be run.

If tested on a PDA, the file 'Clock.jar' must be copied to the folder 'storage\most2\lib\plugins' via ActiveSync and the MOST program may be launched by clicking the link file 'most2\mj9.lnk' or the MOST system will be loaded after the PDA is restarted.

6. When in MOST's main menu using the arrow keys according to the MOST conventions, find the "Plug-ins" menu, enter the "Talking clock" program. Explore the program's menu and any of its functions. Note: in the PC version, a text field may be left by pressing F2, on the PDA, the so-called "Leave edit field" (escape) button does the same.

8 Conclusion

This paper describes a Java-based GUI-free man-machine interface that has been implemented in the MOBILE SlateTalker and provided first for the Hungarian blind users. Programmers may take the opportunity to follow this approach and write immediately blind-friendly applications running on virtually any hardware and operating system. The applications, plug-ins and the MOST system itself can be localized for other languages, provided that there are TTS engines available for those languages.

Reference

1. Arató, A., Juhasz, Z., Blenkhorn, P., Evans, D.G., Evreinov, G.E.: Java-Powered Braille Slate Talker. In: Miesenberger, K., Klaus, J., Zagler, W., Burger, D. (eds.) ICCHP 2004. LNCS, vol. 3118, pp. 506–513. Springer, Heidelberg (2004)