Scrolling on demand - A scrollable toolbar component

Volker Simonis
WSI für Informatik, Universität Tübingen, Germany
email: simonis@informatik.uni-tuebingen.de

May 8, 2004

Abstract

Modern GUI programs offer the possibility to easily access status informations and functionalities by means of various menus, toolbars and information panels. However, as a program becomes more complex, or in the case where users have the possibility to configure and extend these components, they often tend to get overfilled. This leads to scrambled or even truncated components.

This article introduces a new container component called `ScrollableBar`, which can be used as a wrapper for any Swing component. As long as there is enough place to layout the contained component, `ScrollableBar` is completely transparent. As soon as the available space gets too small however, `ScrollableBar` will fade in two small arrow buttons on the left and the right side (or on the top and the bottom side if in vertical mode), which can be used to scroll the underlying component, thus avoiding the above mentioned problems.

`ScrollableBar` is a lightweight container derived from `JComponent` which uses the standard Swing classes `JViewport` and `JButton` to achieve its functionality. It fills a gap in the set of the standard Swing components and offers the possibility to create more robust and intuitive user interfaces.

1 Introduction

Every professional applications comes with a fancy graphical user interface today and with Swing, the standard widget set of Java, it is quite easy to create such applications. However, the design and implementation of a robust and user friendly GUI is not a trivial task. One common problem is the fact that the programmer has no knowledge about the clients desktop size. This may vary today from the standard notebook and flat panel resolution of 1024x768 to 1900x1200 for high end displays. Even worse, Java applications can run on many other devices like for example mobile phones, which have an even more restricted resolution.

Another challenge arises from the extensibility of applications. While having the possibility to extend an application with various plugins may be a nice feature for the user, the fact that these plugins will populate the menus and toolbars in an unpredictable way imposes new problems on the programmer.

Figure 1: Stylepad with scrollable menu, tool and status bars.

One possibility to solve these problems is to limit the size of the GUI components to a certain minimal size. However, this may impose unnecessary restrictions on the user. (Think for example of somebody who by default works with such an application, which needs at least a resolution of 1024x768 but who occasionally gives demo talks with a beamer which only supports an 800x600 resolution.) Furthermore, if an application with a graphical user interface pretends to be resizable by displaying a resizable frame, than the user expects he will be able to resize it based on his needs, not the programmer ones.

The second possibility is to do nothing and wait what happens. This is the way how most of the GUI applications are written today. Just compare figure 3 with figure 4 and see how parts of the status- and toolbars are cut of if the window is shrinked beyond its optimal size. In the best case, the user could just reenlarge the application if this happens. In the worst case, if she is working on a device with a restricted resolution, it may be impossible to access the desired functionality. In any case such an application looks highly unprofessional!

2 Scrollable menus and toolbars!

The solution for all the above mentioned problems would be scrollable menus and toolbars. However Swing, as many other widget sets, does not offer such kind of components. Using the standard `JScrollPane` component as a container for menus and toolbars is not an option here, because `JScrollPane` is too heavy weight. Its scrollbars are simply too big. But there is another Swing component which can serve us as a template: since version 1.4, the `JTabbedPane` class offers the possibility to scroll its panes instead of wrapping them on several lines if they do not fit on a single line. As can be seen in figure 2, arrow buttons for moving the tabs have been added at the upper right part (for more information see [6]).

Figure 2: Example of a `JTabbedPane` with the tab layout policy set to `SCROLL_TAB_LAYOUT`.

We now want to achieve the same behavior for menus, toolbars and other status bars and information panels. To get a visual impression of how the modified components will look like compare the figures 3 and 1. They both show a screen-shot of the Stylepad demo application shipping with every JDK which has been extended by a vertical toolbar and a useful status bar (see figure 4). While the menu, status bar and the toolbars are truncated and partially inaccessible in figure 3, they can be scrolled and are fully functional in figure 1 by using the arrow buttons which have been faded in.

Figure 3: The Stylepad application from figure 4 with truncated tool and status bars.

3 The implementation

I will now describe how to implement a class called `ScrollableBar`, which can serve as a container for a `java.awt.Container` object or any other object derived from it. Most of the time, `ScrollableBar` objects are completely transparent. Only if the place required by the wrapped component for layout becomes too small, the `ScrollableBar` object will fade in two arrow buttons at the left and right side of the component (or on the top and the bottom side if in vertical mode) which can be used to scroll the wrapped component. As soon as there will be again enough place for the layout of the enclosed component, these arrow buttons will disappear immediately.

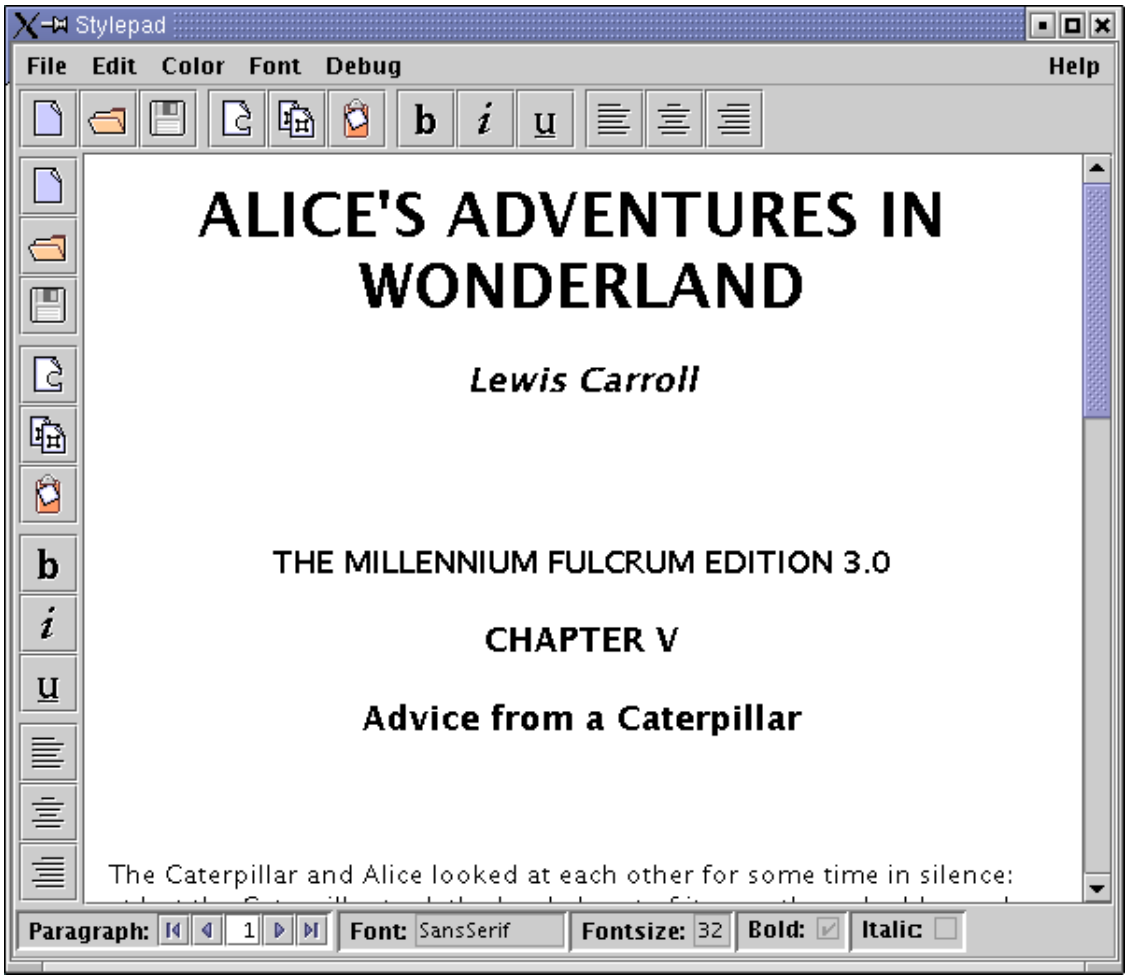

Figure 4: The Stylepad application at preferred size.

3.1 The Swing architecture

For a better understanding of the `ScrollableBar` implementation, it is helpful to take a closer look at the architecture of Swing. The Swing library is a modern widget set based on the Model-View-Controller (MVC) pattern [2]. But while the classical MVC pattern consists of three independent parts, namely the model, the view and the controller, Swing uses a simplified version of this pattern where the view and the controller part are combined in a so called Delegate [7, 1] (see figures 5 and 6).

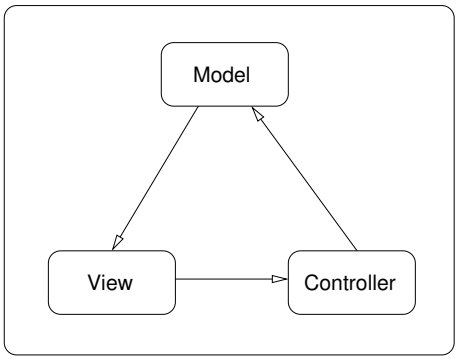

Figure 5: The classical MVC pattern.

As an example, figure 6 shows how this Model-Delegate pattern applies to the `JButton` class. In Swing, all visible components are descendants of the `JComponent` class. They usually capsule a component specific model with a delegate object, which is a descendant of `ComponentUI`. These delegates are called user interface (UI) classes in Swing. They are Look and Feel specific, i.e. they are used to implement the different Look and

Feel dependent properties of a component, but they can also be used for other kinds of customization, like for example localization [4].

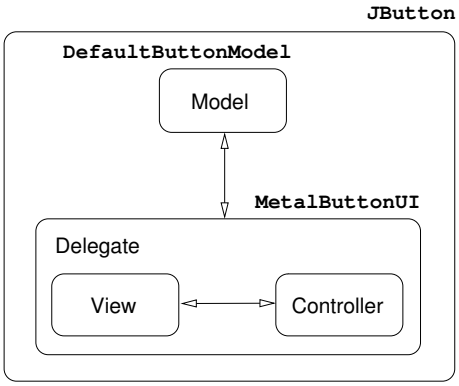

Figure 6: The Model-Delegate pattern used in Swing.

One of the main responsibilities of the UI delegate is to paint the component it is tied to. In contrast to the AWT library, in Swing it is not the `paint()` method of every component which does the work of painting itself. Instead, the component's `paint()` method just calls the `paint()` method of its delegate with a reference to itself.

3.2 The `ScrollableBar` class

Figure 7 shows the class diagram of the `ScrollableBar` class. As already mentioned, it is derived from `JComponent`. It also implements the `SwingConstants` interface in order to easily access the constants `HORIZONTAL` and `VERTICAL` which are defined there.

`ScrollableBar` has 4 properties. The two boolean properties `horizontal` and `small` store the orientation of the component and the size of the arrows on the scroll buttons. The integer property `inc` stores the amount of pixels by which the enclosed component will be scrolled if one of the arrow buttons is being pressed. Smaller values lead to a smoother but slower scrolling. Finally, the wrapped component is stored in the `comp` property. While `horizontal` is a read-only property which can only be set in the constructor, the other three properties are read/write bound properties in the sense described in the Java Beans specification [3].

The following listing shows the two-argument constructor of the `ScrollableBar` class:

Listing 1: ScrollableBar.java [Line 30 to 41]

```
public ScrollableBar(Component comp, int orientation) {
  this.comp = comp;
  if (orientation == HORIZONTAL) {
    horizontal = true;
  }
  else {
    horizontal = false;
  }
  small = true; // Arrow size on scroll button.
  inc = 4;      // Scroll width in pixels.
  updateUI();
}
```

Notice the call to `updateUI()` in the last line of the constructor. As can be seen in listing 2, `updateUI()` calls the static method `getUI()` from the class `UIManager` to query the right UI delegate and associates it with the current `ScrollableBar` object.

Listing 2: ScrollableBar.java [Line 45 to 52]

```
public String getUIClassID() {
  return "ScrollableBarUI";
}

public void updateUI() {
  setUI(UIManager.getUI(this));
  invalidate();
}
```

`UIManager.getUI()` calls `getUIClassID()` (see listing 2) to get the key which is used to query the actual UI delegate from a Look and Feel dependent internal table. Usually, the association of the standard Swing components to the appropriate UI classes is done by the different Look and Feels while they are initialized. However, as we are writing a new component, we have to establish this link manually, as shown in the following listing:

Listing 3: ScrollableBar.java [Line 19 to 22]

```
static {
  UIManager.put("ScrollableBarUI",
                "com.languageExplorer.widgets.↩
                                    ScrollableBarUI");
}
```

Notice that linking a component to its UI delegate in this way results in one and the same UI class being used independently of the actual Look and Feel.

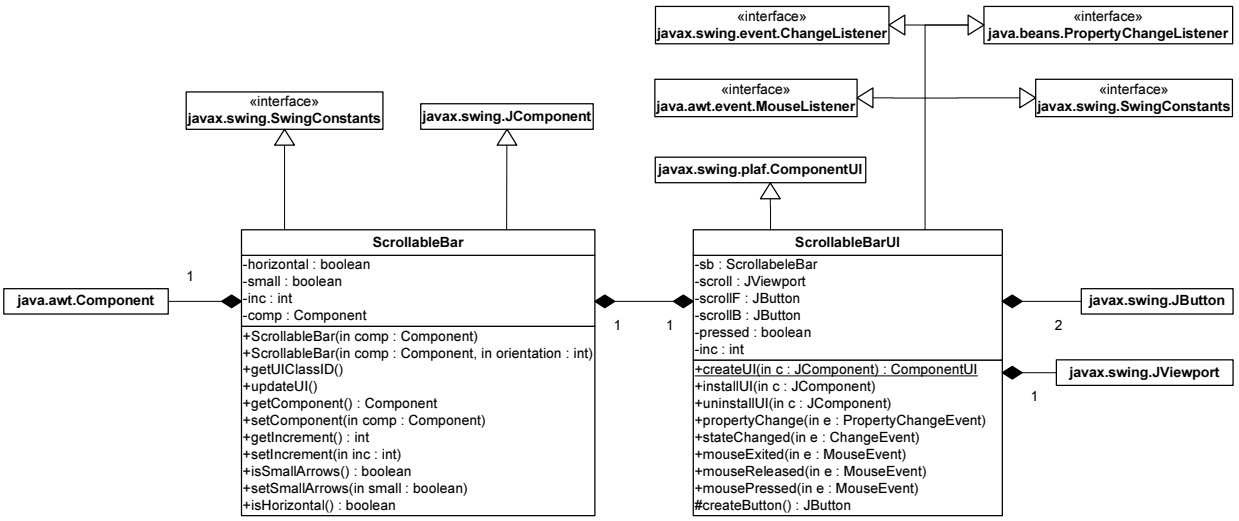

Figure 7: The UML class diagram of `ScrollableBar` and `ScrollableBarUI`.

Besides the getter and setter methods for the corresponding properties, there is no more functionality in the `ScrollableBar` class. All the painting and user interaction is handled by the UI delegate `ScrollableBarUI`.

3.3 The `ScrollableBarUI` class

One of the most important methods of the UI classes is `installUI()` which is called every time when a component is being associated with its UI delegate. This gives the UI delegate a chance to properly initialize itself and the component it is responsible for.

Listing 4: ScrollableBarUI.java [Line 51 to 106]

```
public void installUI(JComponent c) {

  sb = (ScrollableBar)c;

  inc = sb.getIncrement();
  boolean small = sb.isSmallArrows();

  // Create the Buttons
  int sbSize = ((Integer)(UIManager.get( ↩
                   "ScrollBar.width" ))).intValue();
  scrollB = createButton(sb.isHorizontal()?WEST:NORTH, ↩
                                       sbSize, small);
  scrollB.setVisible(false);
  scrollB.addMouseListener(this);

  scrollF = createButton(sb.isHorizontal()?EAST:SOUTH, ↩
```

Listing 4: ScrollableBarUI.java [Line 51 to 106] (continued)

```
                                       sbSize, small);
  scrollF.setVisible(false);
  scrollF.addMouseListener(this);

  int axis = sb.isHorizontal()?BoxLayout.X_AXIS:↩
                                    BoxLayout.Y_AXIS;
  sb.setLayout(new BoxLayout(sb, axis));

  scroll = new JViewport() {
      ... see source code ...
    };

  Component box = sb.getComponent();
  scroll.setView(box);

  sb.add(scrollB);
  sb.add(scroll);
  sb.add(scrollF);

  // Install the change listeners
  scroll.addChangeListener(this);
  sb.addPropertyChangeListener(this);
}
```

In our case, the UI delegate queries and stores the components properties along with a reference to the component itself as private instance variables. Further on, it creates two arrow buttons and an object of type `JViewport` which is used to wrap the scrollable component.

Based on the orientation of the associated `ScrollableBar` object, the newly created elements are then being added to it by using a vertical or horizontal box layout. Notice that the scroll buttons are initially set to be invisible. Finally, the UI object registers itself as property change listener on the associated component, as a change listener on the viewport and as a mouse listener on the arrow buttons.

The UI delegate gets informed about every size change of the `ScrollableBar` object and the wrapped component, by a receiving a `ChangeEvent` from the viewport object. Depending on the new sizes, it can change the visibility state of the arrow buttons and relayout the component. Property changes in the `ScrollableBar` object are signaled to the UI delegate by a `PropertyChangeEvent`. Based on these events, it can update the internally cached values of these properties.

Finally, the events resulting from the user interactions on the scroll buttons are handled by the different mouse listener methods. The UI delegate keeps a private boolean instance variable `pressed` which is set to true if a button was pressed and which is reset to false as soon as the button is released or the mouse pointer leaves the button. As can be seen in listing 5, pressing one of the buttons also starts a new thread which scrolls the underlying component by `inc` pixels in the corresponding direction and than sleeps for a short amount of time. These two actions are subsequently repeated in the thread as long as the value of the instance variable `pressed` is true, while the amount of sleeping time is reduced in every iteration step. This results in a continuously accelerating scrolling speed, as longer the user keeps on pressing the arrow button.

Listing 5: ScrollableBarUI.java [Line 174 to 238]

```
public void mousePressed(MouseEvent e) {
  pressed = true;
  final Object o = e.getSource();
  Thread scroller = new Thread(new Runnable() {
    public void run() {
      int accl = 500;
      while (pressed) {
        Point p = scroll.getViewPosition();
        ... Compute new view position ...
        scroll.setViewPosition(p);
        try {
          Thread.sleep(accl);
          if (accl <= 10) accl = 10;
          else accl /= 2;
```

Listing 5: ScrollableBarUI.java [Line 174 to 238] (continued)

```
        } catch (InterruptedException ie) {}
      }
    }
  });
  scroller.start();
}
```

It should be noticed that we need no special paint method for the `ScrollableBarUI` class, because painting occurs naturally from the standard Swing button and viewport components which we used.

After we have discussed the main parts of the implementation, it should be evident why the advantages of dividing the functionality of the `ScrollableBar` class into two classes outweigh the coding overhead. First of all we cleanly separated the properties of the component from the way how it is displayed and how it interacts with the user. Secondly, it is very easy now to define a new UI delegate which renders the component in a different way or to just derive a new UI delegate from the existing one which slightly adopts appearance or user interaction properties to a specific look and feel.

4 Using the `ScrollableBar` class

Using the `ScrollableBar` class is very easy and straight forward. In fact we can wrap every arbitrary Swing component inside a `ScrollableBar` object by passing it as argument to the constructor when creating the object. For the example application shown in figure 1 it was only necessary to change a single line:

```
JToolBar toolbar = new JToolBar();
...
panel.add("North", toolbar);
```

from the original Stylepad application into:

```
JToolBar toolbar = new JToolBar();
...
panel.add("North", new ScrollableBar(toolbar));
```

in order to make the horizontal toolbar scrollable if the space becomes too small to render it as a whole.

In general, the `ScrollableBar` class is more recommended for wide and not very high components in horizontal mode and narrow and high components in vertical mode. If used for other components the scroll buttons would get too big and take up too much space to be really useful.

4.1 Menu bars in JFrame objects

As shown in the last section it is very easy to use the ScrollableBar class in your own applications. Even upgrading existing applications is not very hard. The only problem which may arise is in the case where a ScrollableBar should be used as a wrapper for a menu bar which will be added directly to a JFrame object. (Notice that in our example application, the menu bar has been added to a JPanel object before the whole panel has been added to the JFrame object.)

The problem arises because JFrame provides a specialized setJMenuBar() method for adding menu bars and this method expects an argument of Type JMenuBar. At a first glance, we could just use one of the generic add() methods defined in JFrame's ancestor classes instead. However, if we take a closer look, we will see that the problem is a little bit more complex.

First of all, in the case of JFrame, children are not being added to the component directly, but to the so called "root pane", which is a special child component of every JFrame. However, we also can not add the menu bar directly to the root pane, because the root pane itself also has a special method called setJMenuBar() which expects a JMenuBar object as argument. Using this method for adding menu bars is essential, because only if it is used the RootLayout layout manager used by the JRootPane class will honor the presence of the menu bar. RootLayout, which is a protected inner class of JRootPane, uses the protected JRootPane property menuBar which has been set by JRootPane.setJMenuBar() for layout calculations.

To cut a long story short, we have to create a new SMJFrame class (which stands for Scrollable Menu JFrame) which overrides the createRootPane() method to return a new, customized root pane class. For this purpose we just derive an anonymous class from JRootPane which overrides the two methods setJMenuBar() and createRootLayout().

setJMenuBar(), the first one of this two methods wraps the menu bar into our ScrollableBar class, before storing it as a protected instance variable and adding it to the layered pane which is a part of the root pane.

The second method createRootLayout() returns an anonymous class which inherits from the JRootPane protected inner class RootLayout. It overrides the layout methods in that class in such a way, that they use the ScrollableBar instance variable for layout calculations instead of using the bare menu bar, as it was done by the original version of the methods.

These modifications finally give the desired result. A call to setJMenuBar() on a SMJFrame object will be forwarded to the customized root pane. There, the menu bar will be wrapped into a ScrollabelBar object before it will be actually added to the frame. Because the customized root pane uses a customized layout manager, it will handle the scrollable menu bar in the same way in which a JFrame object handles an ordinary menu bar. With respect to all other concerns, SMJFrame behaves exactly like its ancestor JFrame.

4.2 Limitations

The only limitation for the use of the ScrollableBar class so far is that it can not handle floating tool bars. This is because JToolBar objects have to be laid out into a container whose layout manager is of type BorderLayout if they want to be floatable. Additionally, no other children can be added to any of the other four "sides". This is obviously not the case, if the toolbar is wrapped inside a ScrollableBar object.

Fixing this problem would require extensive changes in BasicToolBarUI, the UI delegate of JToolBar. Unfortunately, because not all the methods which need to be customized are declared public or protected, in fact a complete rewrite of the delegate would be necessary.

5 Conclusion

This paper presented a quite small and simple, yet very powerful container class which fills a gap in the set of standard Swing components. Using it involves no overhead, neither at development time nor at run time but yields a lot of benefits. The most important ones are: better usability and user friendliness and more robust and intuitive GUI applications.

The source code presented in this paper is available from http://www.progdoc.org/ScrollableBar.jar.

6 Colophon

This paper has been typeset with PDFLaTeX using the twocolumn mode. The screen shots have been prepared with XV and Ghostscript. The figures have been painted with XFig while the UML class diagrams have been done with Visio. The source code presented in this paper has been included and highlighted with the program documentation system ProgDOC[5].

References

[1] R. Eckstein, M. Loy and D. Wood *"Java Swing"*, O'Reilly, 1998

[2] E. Gamma, R.Helm, R. Johnson and J. Vlissides *"Design Patterns: Elements of Reusable Object-Oriented Software"*, Reading, MA, Addison-Wesley, 1995

[3] Graham Hamilton (Ed.) *JavaBeans* Sun Microsystems, Version 1.01-A, August 1997 available at: http://java.sun.com/beans

[4] Volker Simonis *International Swinging: Making Swing Components Locale-Sensitive* C/C++ Users Journal, Java Supplement, August 2002 available at: http://www.cuj.com/java/jsup2008/

[5] Volker Simonis PROGDOC - *The Program Documentation System* available at: http://www.progdoc.org

[6] John Zukowski *"Magic with Merlin: Scrolling tabbed panes"*, available at: http://www-106.ibm.com/developerworks/java/library/j-mer0905/

[7] John Zukowski and Scott Stanchfield *"Fundamentals of JFC/Swing, Part II"*, MageLang Institute, available at: http://developer.java.sun.com/developer/onlineTraining/GUI/Swing2