Closable Tab 1.0 Component Specification

1. Design

The Java Swing Closable Tabs component provides a tabbed pane where each tab can be directly closed. Each tab displays an “X” image that the user can click to close the tab directly. A context menu is also provided to perform other manipulations of the open tabs, like “Close all”, and “Close other tabs”.

The primary aim of this design is to insert the close button and context menu features without interfering with the UI of the JTabbedPane class. The UI of the JTabbedPane class controls the look and feel of the tabbed pane and where and how the tabs are rendered. By using this approach, the look and feel of this component can be freely configured. To accomplish this, we subclass the JTabbedPane class and over-ride two key protected methods.

By over-riding the paintComponent method, we paint the close buttons. We allow 4 different images to be used depending on whether the button is hovered upon or not, and whether the button is on the active tab or not. We work with the existing UI, get the rectangle of the tab from it and paint the button in the top-right corner of that rectangle.

By over-riding the processMouseEvent method, we control what mouse events are passed on to the UI. We intercept the mouse-clicks that trigger the close button as well as the context menu, thus preventing the UI from changing tabs, and leading to a pleasant user experience.

We also provide the ActiveTabSelector contract to control the way in which the new active tab is chosen when the current active tab is closed. This is a very useful feature which leads to a substantial boost in user experience. For example, if the user is currently at tab 0, and then activates tab 4 and later closes it, he expects to be back at tab 0. However, the default handling of the JTabbedPane cannot do this since it does not remember the tab navigation history. We therefore provide two implementations of this contract - one that remembers tab change history and moves to the previous tab and another that goes to the nearest tab in either the increasing or decreasing direction.

The ClosableTabbedPaneEventHandler contract allows handling of the tabbed pane events. While the addition of a tab can only be listened to, the removal of a tab is split into two events - tab closing and tab closed. The handler can react to the closing event and decide whether or not the closed event should occur. This is very useful when each tab represents a modifiable file - for example an editor for multiple files. The handler can be used to ensure the user can save the file or cancel the operation if desired.

1.1 Design Patterns

Strategy Pattern - This pattern is used to abstract the logic of switching the active tab when it is being closed. The contract for the algorithm is set by the ActiveTabSelector contract and the ClosableTabbedPane acts as the context where implementations may be plugged in.

Observer Pattern - This pattern is used by the ClosableTabbedPane as well as PreviousActiveTabSelector classes in order to observe events. This is done by attaching listeners to the sources which need to be observed.

Listener Pattern - This pattern is used to allow applications to listen to events of the ClosableTabbedPane class. It is accomplished by the ClosableTabbedPaneEventHandler contract and ClosableTabbedPaneEvent class. The handlers can be added/removed from the ClosableTabbedPane class.

1.2 Industry Standards

None.

1.3 Required Algorithms

1.3.1 Close button layout and mouse processing

In this component, the close button will always be positioned in the top-right corner of the tab. This makes for a consistent user experience. You may also note that the tab is rendered the same way by the JTabbedPane UI, irrespective of whether the tab is at the top, bottom, left or right.

To get the rectangle for close button image for a given tab, we use the following pseudo-code. The algorithm simply finds the rectangle for the tab from the UI and then compensates for the image width and padding.

```
public class ClosableTabbedPane extends JTabbedPane {
  ...
    private Image myGetCloseButtonImage(CloseButtonState buttonState) {
        Image image = closeButtonImages.get(buttonState);
        if (image == null) {
            throw new IllegalStateException("the button image for " + buttonState
+ " is not set yet");
        }
        long time = System.currentTimeMillis();
        while ((image.getWidth(null) == -1 || image.getHeight(null) == -1)
            && System.currentTimeMillis() - time < 100) {
            // Does nothing
        }
        if (image.getWidth(null) == -1 || image.getHeight(null) == -1) {
            throw new IllegalStateException("the button image for " + buttonState
+ " cannot be loaded");
        }
        return image;
    }
    private Rectangle getCloseImageRectangle(int tabIndex) {
        // Get the button state
        CloseButtonState state = getButtonState(tabIndex);
        // Get the image location and hence the image rectangle
        Image image = myGetCloseButtonImage(state);

        // Get the tab's rectangle
        Rectangle tab = this.getUI().getTabBounds(this, tabIndex);
        // Calculate x by subtracting the padding and image width from the right-
end of the tab.
        int x = tab.x + tab.width - image.getWidth(null) -
closeButtonPadding.width;
        x = tab.x + tab.width - image.getWidth(null) - closeButtonPadding.width;
        // Calculate y by adding the padding to the top of the tab.
        int y = tab.y + closeButtonPadding.height;

        return new Rectangle(x, y, image.getWidth(null), image.getHeight(null));
    }
}
```

To paint all the close-buttons execute the following pseudo-code. The algorithm simply loops over all tabs and for each tab gets the state for the close button in that tab. It then uses the previous location calculation code to get the location and then paints that image.

```
public class ClosableTabbedPane extends JTabbedPane {
  ...
  private CloseButtonState getButtonState(int index)
  {
    CloseButtonState state = null;
    //State if this tab is active
    if(index==this.getSelectedIndex())
      state = index==hoveredTabIndex?CloseButtonState.ACTIVE_TAB_HOVERED:
         ACTIVE_TAB_NOT_HOVERED;
```

```
    //State if this tab is inactive
    else
      state = index==hoveredTabIndex?CloseButtonState.INACTIVE_TAB_HOVERED:
         INACTIVE_TAB_NOT_HOVERED;
    return state;
  }
    protected void paintComponent(Graphics g) {
        ExceptionUtils.checkNull(g, null, null, "The Graphics g must not be
null");
        // Let the UI do its painting first
        super.paintComponent(g);

        // Loop over all tabs
        for (int i = 0; i < this.getTabCount(); i++) {
            CloseButtonState state = getButtonState(i);
            // Get the image location and paint
            Image image = myGetCloseButtonImage(state);

            // Get the position of the close button
            Rectangle imageRect = getCloseImageRectangle(i);

            // Loop until the image is fully loaded
            while (!g.drawImage(image, imageRect.x, imageRect.y, null)) {
                // Empty loop
            }
        }
    }
}
```

We need to process the mouse events for the tabbed pane with the following objectives.

- Find out which tab (if any) has its close button hovered upon.
- Close a tab if the close button is left-clicked.
- Open a context menu if any tab is right-clicked.
- Prevent the tabbed pane from handling any of the above click events. This ensures that the active tab does not change during these operations, leading to an improved user experience.

Pseudo-code is given below. We make use of previously defined functions again.

```
public class ClosableTabbedPane extends JTabbedPane {
...
    protected void processMouseEvent(MouseEvent e) {
        ExceptionUtils.checkNull(e, null, null, "MouseEvent e must not be null");
        // Loop over all tabs
        for (int i = 0; i < this.getTabCount(); i++) {
            // See if the mouse-event falls in this tab's bounds
            Rectangle rect = this.getUI().getTabBounds(this, i);
            if (rect.contains(e.getPoint())) {
                Rectangle imageRect = getCloseImageRectangle(i);
                // If the mouse pointer is over the button
                if (imageRect.contains(e.getPoint())) {
                    // For a left-click
                    if (e.getID() == MouseEvent.MOUSE_CLICKED && e.getButton() ==
MouseEvent.BUTTON1) {
                        // Construct the event
                        ClosableTabbedPaneEvent[] event = new
ClosableTabbedPaneEvent[] {new ClosableTabbedPaneEvent(
                            i, this)};
                        // Fire a closing event
                        boolean[] result = fireTabsClosingEvent(event);
                        // Fire the closed event
                        if (result[0]) {
                            fireTabsClosedEvent(event);
                        }
                        return;
                    }
```

```
                    // Ignore mouse-presses, to prevent the tab from switching in
case the left mouse button is
                    // depressed but the click is still incomplete.
                    if (e.getID() == MouseEvent.MOUSE_PRESSED && e.getButton() ==
MouseEvent.BUTTON1) {
                        return;
                    }
                }

                // If it's a popup trigger, pop the context menu and exit the
method
                if (e.isPopupTrigger()) {
                    closeContextMenuTabIndex = i;
                    closeContextMenu.show(this, e.getX(), e.getY());
                    return;
                }

                // If it's a right mouse button event, ignore it.
                // This is needed since the UI will otherwise activate the tab on
a right mouse down as well
                if (e.getButton() == MouseEvent.BUTTON3) {
                    return;
                }
            }
        }

        // If we reach here, let the mouse event be handled normally
        super.processMouseEvent(e);
    }
    protected void processMouseMotionEvent(MouseEvent e) {
        ExceptionUtils.checkNull(e, null, null, "MouseEvent e must not be null");

        // Ignore MOUSE_DRAGGED
        // http://forums.topcoder.com/?module=Thread&threadID=607194&start=0
        if (e.getID() == MouseEvent.MOUSE_DRAGGED) {
            // Let the mouse event be handled normally
            super.processMouseMotionEvent(e);
            return;
        }
        // Set to -1 if none of the close button is hovered
        int newIndex = -1;
        // Loop over all tabs
        for (int i = 0; i < this.getTabCount(); i++) {
            // See if the mouse-event falls in this tab's bounds
            Rectangle rect = this.getUI().getTabBounds(this, i);
            if (rect.contains(e.getPoint())) {
                Rectangle imageRect = getCloseImageRectangle(i);
                // If the mouse pointer is over the button
                if (imageRect.contains(e.getPoint())) {
                    // Set the hovered tab index to this tab
                    newIndex = i;
                }
            }
        }

        // Let the mouse event be handled normally
        super.processMouseMotionEvent(e);

        // IMPORTANT: Only repaint when hoveredTabIndex is modified.
        // This will strongly make this class efficient
        if (hoveredTabIndex != newIndex) {
            hoveredTabIndex = newIndex;
            repaint();
        }
    }
}
```

1.3.2 Maintaining tab change history

The tab change history is represented using a linked list of integers. The start of the list (i.e. element at index 0) represents currently active, the element at 1 represents the tab

which was active just before the current tab, the element at index 2 was the tab active before that and so on. This history can change due to two events.

- A tab is inserted - In this case, the index numbers of existing tabs may get changed. For all tabs that lie at an index >= the index of the inserted tab, we need to increment the index number to keep the numbers relevant.
- A tab is removed - As with the previous case, we need to update the numbers in the history. Optionally, we may also need to remove those numbers which correspond to tabs that have been removed.
- The selected tab changes - We need to add an element (the current tab index) to the start of the list. Also, if the size of the history is beyond the configured maximum, we need to remove the last element of the history.

When we need to select a new active tab, we simply choose the element at the start of the list and set the selected index of the tabbed pane to that element. Note that we need that all tab closes to be done through the close button or context menu in order for this to work properly.

1.4 Component Class Overview

ClosableTabbedPane - This is the primary class of the component. It extends JTabbedPane and can be used as a drop-in replacement for it. It paints a close button in the top-right corner of each tab which can be used to close that tab. It also allows closure of tabs using a context menu. An optional border can be drawn around tab components. Event handlers can also be attached that can listen to tab addition and removal events - with the option to override the closure of tabs.

ClosableTabbedPane#ContextMenuItemActionListener - This private inner class of the closable tabbed pane listens for the selection of menu items in the context menu that allows for tab closure.

CloseButtonState - This enumeration lists the possible states in which a close button can be.

ClosableTabbedPaneEventHandler - This interface lays the contract for handling tabbed pane events (addition/removal of tabs). This is a handler - and not just a passive listener - since it can prevent the closure of tabs.

ClosableTabbedPaneEvent - Represents an event of the ClosableTabbedPane. It simply stores the index of the tab being affected and the source pane which caused the event.

ActiveTabSelector - This interface lays the contract for selecting a new active tab when a currently active tab is closed.

DirectionalActiveTabSelector - This class implements the ActiveTabSelector interface by selecting the nearest tab in one of two direction - the direction of increasing tab index or the direction of decreasing tab index.

PreviousActiveTabSelector - This class implements the ActiveTabSelector interface by selecting the previous tab that was active.

PreviousActiveTabSelector#HistoryMaintenanceClosableTabbedPaneEventHandler - This class implements the ClosableTabbedPaneEventHandler interface in order to maintain the correct tab change history despite tabs closing and opening.

PreviousActiveTabSelector#HistoryMaintenanceChangeListener - This class implements the ChangeListener interface in order to maintain the tab change history.

1.5 Component Exception Definitions

ClosableTabbedPaneConfigurationException - This exception is thrown to indicate an error during the configuration of the closable tabbed pane.

java.lang.IllegalArgumentException - This exception is thrown if the arguments passed to a method are not legal.

java.lang.IllegalStateException - This exception is thrown if a method being called is not legal for the current state of the object.

1.6 Thread Safety

This component is not thread safe. The primary class, ClosableTabbedPane is mutable and also inherits from JTabbedPane (which is unsafe) making it unsafe for multi-threaded operations. The PreviousActiveTabSelector class is also not safe since it has mutable state. All other classes are thread safe - either by being immutable or by being stateless. To use this component in a thread safe manner, there are two possible usage patterns.

- Follow standard usage of Swing widgets - instantiate and configure the widget, then route all operations on it through the Swing event handler thread. This usage pattern is followed in the demo.
- Synchronize on shared instances of ClosableTabbedPane.

2. Environment Requirements

2.1 Environment

- Java 1.5 is required for compilation and execution.

2.2 TopCoder Software Components

- **Base Exception 2.0** - This component provides the base exception for the exceptions of our component.
- **Configuration Manager 2.1.5** - This component provides configuration support for our component.
- **Object Factory 2.0.1** - This component allows our component to create instances of ActiveTabSelector.

NOTE: The default location for TopCoder Software component jars is../lib/tcs/COMPONENT_NAME/COMPONENT_VERSION relative to the component installation. Setting the tcs_libdir property in topcoder_global.properties will overwrite this default location.

2.3 Third Party Components

None.

NOTE: The default location for 3rd party packages is ../lib relative to this component installation. Setting the ext_libdir property in topcoder_global.properties will overwrite this default location.

3. Installation and Configuration

3.1 Package Name

com.topcoder.gui.closabletabbedpane

com.topcoder.gui.closabletabbedpane.activetabselector

3.2 Configuration Parameters

Name	Description	Required/Optional
config_manager_specification_factory_namespace	This parameter gives the namespace to create the ConfigManagerSpecificationFactory class.	**Optional.** Must not be empty if present.
active_tab_selector_key	This parameter gives the key to create an instance of ActiveTabSelector using the object factory. Used only if the specification factory property is present.	**Optional.** Must not be empty if present.
active_tab_selector_identifier	This parameter gives the identifier to use with the key while creating the ActiveTabSelector. Used only if the key is present.	**Optional.** Must not be empty if present.
active_tab_hovered	This parameter gives the path to the image to be shown for the close button when it is in the ACTIVE_TAB_HOVERED state.	**Optional.** Must not be empty if present.
active_tab_not_hovered	This parameter gives the path to the image to be shown for the close button when it is in the ACTIVE_TAB_NOT_HOVERED state.	**Optional.** Must not be empty if present.
inactive_tab_hovered	This parameter gives the path to the image to be shown for the close button when it is in the INACTIVE_TAB_HOVERED state.	**Optional.** Must not be empty if present.
inactive_tab_not_hovered	This parameter gives the path to the image to be shown for the close button when it is in the INACTIVE_TAB_NOT_HOVERED state.	**Optional.** Must not be empty if present.
close_button_padding_top	This parameter gives the padding in pixels between the close button and the top of the tab.	**Optional.** Default to 0 if absent. Must be non-negative. Must not be empty if present.
close_button_padding_right	This parameter gives the padding in pixels between the close button and the right-end of the tab.	**Optional.** Default to 0 if absent. Must be non-negative. Must not be empty if present.
interior_border_color	This parameter gives the color of the interior border in the #RRGGBB format where RR, GG and BB are hex values.	**Optional.** Default to getBackground(). If interior_border_width is also absent, null

		Border will be used. Must not be empty if present.
interior_border_width	This parameter gives the width of the interior border in pixels.	**Optional.** Default to 1. If interior_border_width is also absent, null Border will be used. Must be non-negative. Must not be empty if present.

3.3 Dependencies Configuration

None.

4. Usage Notes

4.1 Required steps to test the component

- Extract the component distribution.

Follow Dependencies Configuration.

- Execute 'ant test' within the directory that the distribution was extracted to.

4.2 Required steps to use the component

While the component is fully configurable programmatically, we can also configure it using namespace properties. A sample configuration is given below.

```
<?xml version="1.0" ?>
<CMConfig>
    <!-- Namespace for ObjectFactory component -->
    <Config name="ofNamespace">
        <Property name="selector:default">
            <Property name="type">

<Value>com.topcoder.gui.closabletabbedpane.activetabselector.DirectionalActiveTabS
elector</Value>
            </Property>
            <Property name="params">
               <Property name="param1">
                      <Property name="type">
                             <Value>boolean</Value>
                      </Property>
                      <Property name="value">
                             <Value>true</Value>
                      </Property>
               </Property>
            </Property>
        </Property>
    </Config>

    <!-- Namespace for main class -->
    <Config name="myNamespace">
        <Property name="config_manager_specification_factory_namespace">
            <Value>ofNamespace</Value>
        </Property>
        <Property name="active_tab_selector_key">
            <Value>selector</Value>
        </Property>
        <Property name="active_tab_selector_identifier">
            <Value>default</Value>
        </Property>
        <Property name="active_tab_hovered">
```

```
            <Value>test_files/1.gif</Value>
        </Property>
        <Property name="active_tab_not_hovered">
            <Value>test_files/2.gif</Value>
        </Property>
        <Property name="inactive_tab_hovered">
            <Value>test_files/3.gif</Value>
        </Property>
        <Property name="inactive_tab_not_hovered">
            <Value>test_files/4.gif</Value>
        </Property>
        <Property name="close_button_padding_right">
            <Value>5</Value>
        </Property>
        <Property name="close_button_padding_top">
            <Value>4</Value>
        </Property>
        <Property name="interior_border_color">
            <Value>#4FAE52</Value>
        </Property>
        <Property name="interior_border_width">
            <Value>7</Value>
        </Property>
    </Config>

</CMConfig>
```

4.3 Demo

This component can be used to enhance any application that uses tabbed panes, if closing individual tabs makes sense. We show how it can be used inside a text editor that can be used to edit multiple files at one time. Exception handling is omitted for the sake of clarity.

```
public class TextEditor implements ClosableTabbedPaneEventHandler,
DocumentListener {
    /** Declare all necessary widgets, the main JFrame, menu bars etc. */
    private JFrame frame = new JFrame("Hello");

    /** Declare the closable tabbed pane. */
    private ClosableTabbedPane tabbedPane;

    /** The currently open files. */
    private List<String> filenames = new ArrayList<String>();

    /** For each tab, whether the contents of the tab have been modified. */
    private List<Boolean> modified = new ArrayList<Boolean>();

    /**
     * The constructor of this editor.
     */
    public TextEditor() {
        // Initialize all widgets

        /*
         * Create and add the closable tabbed pane, add a listener to it. We
perform programmatic configuration. We
         * could also use namespace properties as shown in section 4.2
         */

        tabbedPane = new ClosableTabbedPane(JTabbedPane.BOTTOM);

        // We-use the different close images for 4 close button states
        tabbedPane.setCloseButtonImage(CloseButtonState.ACTIVE_TAB_HOVERED,
Toolkit.getDefaultToolkit()
            .createImage("test_files/1.gif"));
        tabbedPane.setCloseButtonImage(CloseButtonState.ACTIVE_TAB_NOT_HOVERED,
Toolkit.getDefaultToolkit()
            .createImage("test_files/2.gif"));
        tabbedPane.setCloseButtonImage(CloseButtonState.INACTIVE_TAB_HOVERED,
Toolkit.getDefaultToolkit()
```

```
            .createImage("test_files/3.gif"));
        tabbedPane.setCloseButtonImage(CloseButtonState.INACTIVE_TAB_NOT_HOVERED,
Toolkit.getDefaultToolkit()
            .createImage("test_files/4.gif"));

        // A rounded, blue, 5-pixel border
        tabbedPane.setInteriorBorder(new LineBorder(Color.BLUE, 5, true));

        // A 5x5 padding at the top-right
        tabbedPane.setCloseButtonPadding(new Dimension(5, 5));

        /* When an active tab closes, we want to switch to the last tab which was
active */
        // tabbedPane.setActiveTabSelector(new PreviousActiveTabSelector());
        // Add an event handler and add the pane to the frame
        tabbedPane.addClosableTabbedPaneEventHandler(this);
        frame.getContentPane().add(tabbedPane);

        // Display the frame etc.
        frame.setSize(1000, 500);
        frame.setVisible(true);

        openNewFile("test_files/a.txt");
        openNewFile("test_files/b.txt");
        openNewFile("test_files/c.txt");
        openNewFile("test_files/d.txt");
    }

    /**
     * The entrance to this demo.
     * @param args
     *            the arguments
     */
    public static void main(String[] args) {
        // Simply create the editor
        new TextEditor();
    }

    /**
     * This method should be called by the menu-item handler for the open file
menu item.
     * @param filename
     *            name of the file to open
     */
    private void openNewFile(String filename) {
        try {
            // If the file is already open, simply activate the tab
            int index = filenames.indexOf(filename);
            if (index != -1) {
                tabbedPane.setSelectedIndex(index);
                return;
            }

            // This string will hold the contents of the file
            String text = "";
            BufferedReader br = new BufferedReader(new FileReader(filename));
            while (true) {
                String str = br.readLine();
                if (str == null) {
                    break;
                }
                text += str + "\n";
            }
            br.close();

            // Create a text area with the content
            JTextArea area = new JTextArea(text, 10, 20);

            // Add a listener to check for modifications
            area.getDocument().addDocumentListener(this);
```

```
            // Add the text area as a tab
            tabbedPane.insertTab(filename, null, area, null,
tabbedPane.getTabCount());

            // Add the filename and an entry for modified
            filenames.add(filename);
            modified.add(Boolean.FALSE);
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    /**
     * The update event is caught here, and we can mark the text to be modified -
should be saved later.
     * @param e
     *            the event
     */
    public void changedUpdate(DocumentEvent e) {
        for (int i = 0; i < tabbedPane.getTabCount(); i++) {
            if (((JTextArea) tabbedPane.getComponentAt(i)).getDocument() ==
e.getDocument()) {
                modified.set(i, Boolean.TRUE);
            }
        }
    }

    /**
     * Does same thing as changedUpdate.
     * @param e
     *            the event
     */
    public void removeUpdate(DocumentEvent e) {
        changedUpdate(e);
    }

    /**
     * Does same thing as changedUpdate.
     * @param e
     *            the event
     */
    public void insertUpdate(DocumentEvent e) {
        changedUpdate(e);
    }

    /**
     * Does nothing special when tab is added.
     * @param e
     *            the event
     */
    public void tabAdded(ClosableTabbedPaneEvent e) {
    }

    /**
     * When tabs are closing we ask the user if he wishes to save any unsaved
files. The user can also cancel the
     * operation (using a standard yes/no/cancel dialog).
     * @param e
     *            the events
     * @return if each event is handled properly
     */
    public boolean[] tabsClosing(ClosableTabbedPaneEvent[] e) {
        boolean[] result = new boolean[e.length];
        for (int i = 0; i < e.length; i++) {
            // If the file has been modified
            if (modified.get(e[i].getTabIndex())) {
                // Show a confirm dialog
                String message = "Do you want to save " +
filenames.get(e[i].getTabIndex()) + "?";
                int option = JOptionPane.showConfirmDialog(tabbedPane, message);
```

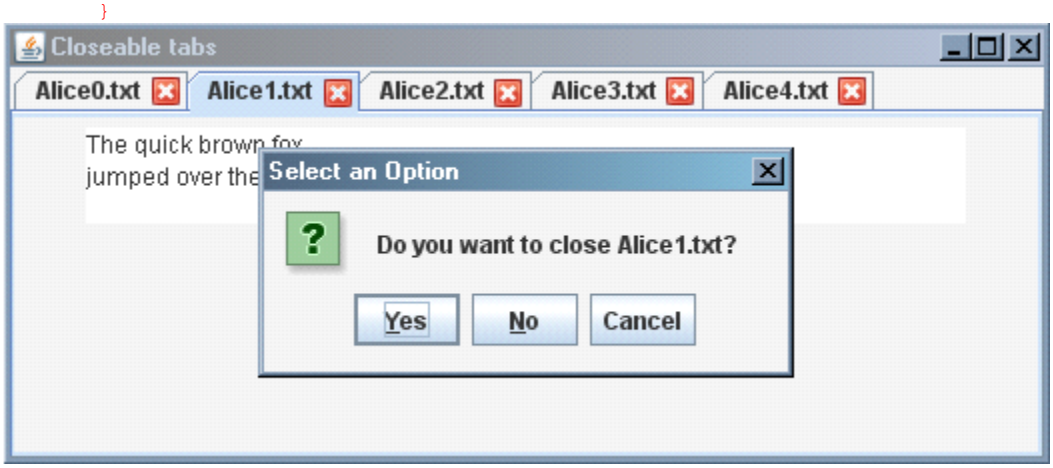
```
                    if (option == JOptionPane.YES_OPTION) {
                        // TODO Save the file

                        result[i] = true;
                    } else if (option == JOptionPane.NO_OPTION) {
                        result[i] = true;
                    }
                } else {
                    // Comes here if the file is not modified
                    result[i] = true;
                }
            }
            return result;
        }

        /**
         * When the tabs are closed, we remove the entries from the filenames and
    modified lists.
         * @param e
         *            the events
         */
        public void tabsClosed(ClosableTabbedPaneEvent[] e) {
            // First store the indices
            int[] index = new int[e.length];
            for (int i = 0; i < e.length; i++) {
                index[i] = e[i].getTabIndex();
            }

            // Sort the array, so we can remove the entries from last to first
            Arrays.sort(index);

            // Remove the entries
            for (int i = e.length - 1; i >= 0; i--) {
                filenames.remove(index[i]);
                modified.remove(index[i]);
            }
        }
    }
```



5. Future Enhancements

None presently.