Java Swing model architecture

Swing engineers created the Swing toolkit implementing a modified Model View Controller design pattern. This enables efficient handling of data and using pluggable look and feel at runtime.

The traditional MVC pattern divides an application into three parts: a model, a view and a controller. The model represents the data in the application. The view is the visual representation of the data. And finally the controller processes and responds to events, typically user actions, and may invoke changes on the model. The idea is to separate the data access and business logic from data presentation and user interaction, by introducing an intermediate component: the controller.

The Swing toolkit uses a modified MVC design pattern. It has a single *UI object* for both the view and the controller. This modified MVC is sometimes called a *separable model architecture*.

In the Swing toolkit, every component has its model, even the basic ones like buttons. There are two kinds of models in Swing toolkit:

- state models
- data models

The state models handle the state of the component. For example the model keeps track whether the component is selected or pressed. The data models handle data they work with. A list component keeps a list of items that it is displaying.

For Swing developers it means that they often need to get a model instance in order to manipulate the data in the component. But there are exceptions. For convenience, there are some methods that return data without the need for a programmer to access the model.

```
public int getValue() {
    return getModel().getValue();
}
```

An example is the `getValue()` method of the `JSlider` component. The developer does not need to work with the model directly. Instead, the access to the model is done behind the scenes. It would be an overkill to work with models directly in such simple situations. Because of this, Swing provides some convenience methods like the previous one.

To query the state of the model, we have two kinds of notifications:

- lightweight notifications
- stateful notifications

A lightweight notification uses a `ChangeListener` class. We have only one single event (`ChangeEvent)` for all notifications coming from the component. For more complicated components, a stateful notification is used. For such notifications, we have different kinds of events. For example the

`JList` component has `ListDataEvent` and `ListSelectionEvent`.

If we do not set a model for a component, a default one is created. For example the button component has a `DefaultButtonModel` model.

```
public JButton(String text, Icon icon) {
  // Create the model
  setModel(new DefaultButtonModel());

  // initialize
  init(text, icon);
}
```

Looking at the JButton.java source file we find out that the default model is created at the construction of the component.

ButtonModel

The model is used for various kinds of buttons like push buttons, check boxes, radio boxes and for menu items. The following example illustrates the model for a `JButton`. We can manage only the state of the button because no data can be associated with a push button.

```
package com.zetcode;

import java.awt.Container;
import java.awt.EventQueue;
import java.awt.event.ActionEvent;
import javax.swing.AbstractAction;
import javax.swing.DefaultButtonModel;
import javax.swing.GroupLayout;
import javax.swing.JButton;
import javax.swing.JCheckBox;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.event.ChangeEvent;
import javax.swing.event.ChangeListener;

public class ButtonModel extends JFrame {

    private JButton okbtn;
    private JLabel enabledLbl;
    private JLabel pressedLbl;
    private JLabel armedLbl;
    private JCheckBox cb;
    public ButtonModel() {

        initUI();
    }

    private void initUI() {

        Container pane = getContentPane();
        GroupLayout gl = new GroupLayout(pane);
        pane.setLayout(gl);

        okbtn = new JButton("OK");
        okbtn.addChangeListener(new DisabledChangeListener());
        cb = new JCheckBox();
        cb.setAction(new CheckBoxAction());

        enabledLbl = new JLabel("Enabled: true");
        pressedLbl = new JLabel("Pressed: false");
        armedLbl = new JLabel("Armed: false");

        gl.setAutoCreateContainerGaps(true);
        gl.setAutoCreateGaps(true);

        gl.setHorizontalGroup(gl.createParallelGroup()
                .addGroup(gl.createSequentialGroup()
                        .addComponent(okbtn)
                        .addGap(80)
                        .addComponent(cb))
                .addGroup(gl.createParallelGroup()
                        .addComponent(enabledLbl)
                        .addComponent(pressedLbl)
                        .addComponent(armedLbl))
        );

        gl.setVerticalGroup(gl.createSequentialGroup()
                .addGroup(gl.createParallelGroup()
                        .addComponent(okbtn)
                        .addComponent(cb))
                .addGap(40)
                .addGroup(gl.createSequentialGroup()
                        .addComponent(enabledLbl)
                        .addComponent(pressedLbl)
                        .addComponent(armedLbl))
        );
        pack();

        setTitle("ButtonModel");
        setLocationRelativeTo(null);
        setDefaultCloseOperation(EXIT_ON_CLOSE);
    }

    private class DisabledChangeListener implements ChangeListener {

        @Override
        public void stateChanged(ChangeEvent e) {

            DefaultButtonModel model = (DefaultButtonModel)
okbtn.getModel();

            if (model.isEnabled()) {
                enabledLbl.setText("Enabled: true");
            } else {
                enabledLbl.setText("Enabled: false");
            }

            if (model.isArmed()) {
                armedLbl.setText("Armed: true");
            } else {
                armedLbl.setText("Armed: false");
            }

            if (model.isPressed()) {
                pressedLbl.setText("Pressed: true");
            } else {
                pressedLbl.setText("Pressed: false");
            }
        }
    }

    private class CheckBoxAction extends AbstractAction {

        public CheckBoxAction() {
            super("Disabled");
        }

        @Override
        public void actionPerformed(ActionEvent e) {
            if (okbtn.isEnabled()) {
                okbtn.setEnabled(false);
            } else {
                okbtn.setEnabled(true);
            }
        }
    }

    public static void main(String[] args) {

        EventQueue.invokeLater(new Runnable() {
            @Override
            public void run() {
                ButtonModel ex = new ButtonModel();
                ex.setVisible(true);
            }
        });
    }
}
```

In our example, we have a button, a check box, and three labels. The labels represent three properties of the button: pressed, disabled, or armed state.

```
okbtn.addChangeListener(new DisabledChangeListener());
```

We use a `ChangeListener` to listen for button state changes.

```
DefaultButtonModel model = (DefaultButtonModel) okbtn.getModel();
```

Here we get the default button model.

```
if (model.isEnabled()) {
    enabledLbl.setText("Enabled: true");
} else {
    enabledLbl.setText("Enabled: false");
}
```

We query the model whether the button is enabled. The label is updated accordingly.

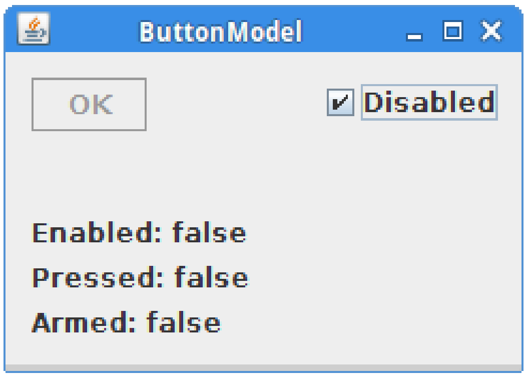
```
if (okbtn.isEnabled()) {
    okbtn.setEnabled(false);
} else {
    okbtn.setEnabled(true);
```

The check box enables or disables the button. To enable the OK button, we call the `setEnabled()` method. So we change the state of the button. Where is the model? The answer lies in the

`AbstractButton.java` file.

```
public void setEnabled(boolean b) {
    if (!b && model.isRollover()) {
        model.setRollover(false);
    }
    super.setEnabled(b);
    model.setEnabled(b);
}
```

The answer is that internally the Swing toolkit works with a model. The `setEnabled()` is another convenience method for programmers.



```
Enabled: false
Pressed: false
Armed: false
```

Figure: ButtonModel

Custom ButtonModel

In the previous example, we used a default button model. In the following code example we will use our own button model.

```
package com.zetcode;

import java.awt.Container;
import java.awt.EventQueue;
import java.awt.event.ActionEvent;
import javax.swing.AbstractAction;
import javax.swing.ButtonModel;
import javax.swing.DefaultButtonModel;
import javax.swing.GroupLayout;
import javax.swing.JButton;
import javax.swing.JCheckBox;
import javax.swing.JFrame;
import static javax.swing.JFrame.EXIT_ON_CLOSE;
import javax.swing.JLabel;


public class ButtonModel2 extends JFrame {
    private JButton okbtn;
    private JLabel enabledLbl;
    private JLabel pressedLbl;
    private JLabel armedLbl;
    private JCheckBox cb;

    public ButtonModel2() {

        initUI();
    }

    private void initUI() {

        Container pane = getContentPane();
        GroupLayout gl = new GroupLayout(pane);
        pane.setLayout(gl);

        okbtn = new JButton("OK");
        cb = new JCheckBox();
        cb.setAction(new CheckBoxAction());

        enabledLbl = new JLabel("Enabled: true");
        pressedLbl = new JLabel("Pressed: false");
        armedLbl  = new JLabel("Armed: false");

        ButtonModel model = new OkButtonModel();
        okbtn.setModel(model);

        gl.setAutoCreateContainerGaps(true);
        gl.setAutoCreateGaps(true);

        gl.setHorizontalGroup(gl.createParallelGroup()
                .addGroup(gl.createSequentialGroup()
                        .addComponent(okbtn)
                        .addGap(80)
                        .addComponent(cb))
                .addGroup(gl.createParallelGroup()
                        .addComponent(enabledLbl)
                        .addComponent(pressedLbl)
                        .addComponent(armedLbl))
        );

        gl.setVerticalGroup(gl.createSequentialGroup()
                .addGroup(gl.createParallelGroup()
                        .addComponent(okbtn)
                        .addComponent(cb))
                .addGap(40)
                .addGroup(gl.createSequentialGroup()
                        .addComponent(enabledLbl)
                        .addComponent(pressedLbl)
                        .addComponent(armedLbl))
        );

        pack();

        setTitle("Custom button model");
        setLocationRelativeTo(null);
        setDefaultCloseOperation(EXIT_ON_CLOSE);
    }

    private class OkButtonModel extends DefaultButtonModel {

        @Override
        public void setEnabled(boolean b) {
            if (b) {
                enabledLbl.setText("Enabled: true");
            } else {
                enabledLbl.setText("Enabled: false");
            }

            super.setEnabled(b);
        }

        @Override
        public void setArmed(boolean b) {
            if (b) {
                armedLbl.setText("Armed: true");
            } else {
                armedLbl.setText("Armed: false");
            }

            super.setArmed(b);
        }

        @Override
        public void setPressed(boolean b) {
            if (b) {
                pressedLbl.setText("Pressed: true");
            } else {
                pressedLbl.setText("Pressed: false");
            }

            super.setPressed(b);
        }
    }

    private class CheckBoxAction extends AbstractAction {

        public CheckBoxAction() {
            super("Disabled");
        }

        @Override
        public void actionPerformed(ActionEvent e) {
            if (okbtn.isEnabled()) {
                okbtn.setEnabled(false);
            } else {
                okbtn.setEnabled(true);
            }
        }
    }

    public static void main(String[] args) {

        EventQueue.invokeLater(new Runnable() {
            @Override
            public void run() {
                ButtonModel2 ex = new ButtonModel2();
                ex.setVisible(true);
            }
        });
    }
}
```

This example does the same thing as the previous one. The difference is that we do not use a change listener and we use a custom button model.

```
ButtonModel model = new OkButtonModel();
okbtn.setModel(model);
```

We set the custom model for the button.

```
private class OkButtonModel extends DefaultButtonModel {
...
}
```

We create a custom button model and override the necessary methods.

```
@Override
public void setEnabled(boolean b) {
    if (b) {
        enabledLbl.setText("Enabled: true");
    } else {
        enabledLbl.setText("Enabled: false");
    }

    super.setEnabled(b);
}
```

We override the `setEnabled()` method and add some functionality there. We must not forget to call the parent method as well to proceed with the processing.

JList models

Several components have two models; `JList` is one of them. It has the following models: the `ListModel` and the `ListSelectionModel`. The `ListModel` handles data and the `ListSelectionModel` works with the selection state of the list. The following example uses both models.

```
package com.zetcode;

import java.awt.Container;
import java.awt.EventQueue;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.awt.event.MouseAdapter;
import java.awt.event.MouseEvent;
import javax.swing.DefaultListModel;
import javax.swing.GroupLayout;
import static javax.swing.GroupLayout.Alignment.CENTER;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JList;
import javax.swing.JOptionPane;
import javax.swing.JScrollPane;
import javax.swing.ListSelectionModel;

public class ListModels extends JFrame {

    private DefaultListModel model;
    private JList list;
    private JButton remallbtn;
    private JButton addbtn;
    private JButton renbtn;
    private JButton delbtn;

    public ListModels() {

        initUI();
    }

    private void createList() {

        model = new DefaultListModel();
        model.addElement("Amelie");
        model.addElement("Aguirre, der Zorn Gottes");
        model.addElement("Fargo");
        model.addElement("Exorcist");
        model.addElement("Schindler's list");

        list = new JList(model);
        list.setSelectionMode(ListSelectionModel.SINGLE_SELECTION);

        list.addMouseListener(new MouseAdapter() {

            @Override
            public void mouseClicked(MouseEvent e) {

                if (e.getClickCount() == 2) {
                    int index = list.locationToIndex(e.getPoint());
                    Object item = model.getElementAt(index);
                    String text = JOptionPane.showInputDialog("Rename item",
item);
                    String newitem = null;
                    if (text != null) {
                        newitem = text.trim();
                    } else {
                        return;
                    }

                    if (!newitem.isEmpty()) {
                        model.remove(index);
                        model.add(index, newitem);
                        ListSelectionModel selmodel =
list.getSelectionModel();
                        selmodel.setLeadSelectionIndex(index);
                    }
                }
            }
        });
    }

    private void createButtons() {

        remallbtn = new JButton("Remove All");
        addbtn = new JButton("Add");
        renbtn = new JButton("Rename");
        delbtn = new JButton("Delete");

        addbtn.addActionListener(new ActionListener() {

            @Override
            public void actionPerformed(ActionEvent e) {

                String text = JOptionPane.showInputDialog("Add a new item");
                String item = null;

                if (text != null) {
                    item = text.trim();
                } else {
                    return;
                }

                if (!item.isEmpty()) {
                    model.addElement(item);
                }
            }
        });

        delbtn.addActionListener(new ActionListener() {

            @Override
            public void actionPerformed(ActionEvent event) {

                ListSelectionModel selmodel = list.getSelectionModel();
                int index = selmodel.getMinSelectionIndex();
                if (index >= 0) {
                    model.remove(index);
                }
            }

        });

        renbtn.addActionListener(new ActionListener() {

            @Override
            public void actionPerformed(ActionEvent e) {

                ListSelectionModel selmodel = list.getSelectionModel();
                int index = selmodel.getMinSelectionIndex();
                if (index == -1) {
                    return;
                }

                Object item = model.getElementAt(index);
                String text = JOptionPane.showInputDialog("Rename item",
item);
                String newitem = null;

                if (text != null) {
                    newitem = text.trim();
                } else {
                    return;
                }

                if (!newitem.isEmpty()) {
                    model.remove(index);
                    model.add(index, newitem);
                }
            }
        });

        remallbtn.addActionListener(new ActionListener() {
            @Override
            public void actionPerformed(ActionEvent e) {
                model.clear();
            }
        });

    }

    private void initUI() {
        createList();
        createButtons();
        JScrollPane scrollpane = new JScrollPane(list);

        Container pane = getContentPane();
        GroupLayout gl = new GroupLayout(pane);
        pane.setLayout(gl);

        gl.setAutoCreateContainerGaps(true);
        gl.setAutoCreateGaps(true);

        gl.setHorizontalGroup(gl.createSequentialGroup()
                .addComponent(scrollpane)
                .addGroup(gl.createParallelGroup()
                        .addComponent(addbtn)
                        .addComponent(renbtn)
                        .addComponent(delbtn)
                        .addComponent(remallbtn))
        );

        gl.setVerticalGroup(gl.createParallelGroup(CENTER)
                .addComponent(scrollpane)
                .addGroup(gl.createSequentialGroup()
                        .addComponent(addbtn)
                        .addComponent(renbtn)
                        .addComponent(delbtn)
                        .addComponent(remallbtn))
        );

        gl.linkSize(addbtn, renbtn, delbtn, remallbtn);

        pack();

        setTitle("JList models");
        setLocationRelativeTo(null);
        setDefaultCloseOperation(EXIT_ON_CLOSE);
    }

    public static void main(String[] args) {

        EventQueue.invokeLater(new Runnable() {
            @Override
            public void run() {
                ListModels ex = new ListModels();
                ex.setVisible(true);
            }
        });
    }
}
```

The example shows a list component and four buttons. The buttons control the data in the list component. The example is a bit larger because we did some additional checks there. For instance, we do not allow to input empty spaces into the list component.

```
model = new DefaultListModel();
model.addElement("Amelie");
model.addElement("Aguirre, der Zorn Gottes");
...
```

We create a default list model and add elements into it.

```
list = new JList(model);
list.setSelectionMode(ListSelectionModel.SINGLE_SELECTION);
```

We create a list component. The parameter of the constructor is the model we have created. We make the list into the single selection mode.

```
if (text != null) {
    item = text.trim();
} else {
    return;
}

if (!item.isEmpty()) {
    model.addElement(item);
}
```

We add only items that are not equal to null and are not empty, e.g. items that contain at least one character other than white space. It makes no sense to add white spaces or null values into the list.

```
ListSelectionModel selmodel = list.getSelectionModel();
int index = selmodel.getMinSelectionIndex();
if (index >= 0) {
    model.remove(index);
}
```

This is the code that runs when we press the delete button. In order to delete an item from the list, it must be selected—we must figure out the currently selected item. For this, we call the `getSelectionModel()` method. We get the selected index with the `getMinSelectionIndex()` and remove the item with the `remove()` method.

In this example we used both list models. We called the add(), remove() and clear() methods of the list data model to work with our data. And we used a list selection model in order to find out the selected item.

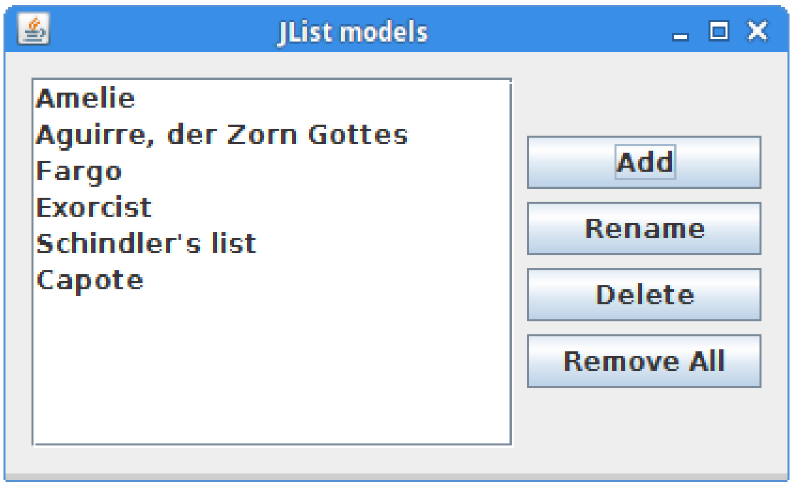


Figure: List Models

A document model

A document model is a good example of a separation of a data from the visual representation. In a `JTextPane` component, we have a `StyledDocument` for setting the style of the text data.

```
package com.zetcode;

import java.awt.BorderLayout;
import java.awt.EventQueue;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import javax.swing.BorderFactory;
import javax.swing.ImageIcon;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JPanel;
import javax.swing.JScrollPane;
import javax.swing.JTextPane;
import javax.swing.JToolBar;
import javax.swing.text.Style;
import javax.swing.text.StyleConstants;
import javax.swing.text.StyledDocument;

public class DocumentModel extends JFrame {

    private StyledDocument sdoc;
    private JTextPane textpane;
    public DocumentModel() {

        initUI();
    }

    private void initUI() {

        createToolbar();

        JPanel panel =  new JPanel(new BorderLayout());
        panel.setBorder(BorderFactory.createEmptyBorder(8, 8, 8, 8));

        textpane = new JTextPane();
        sdoc = textpane.getStyledDocument();
        initStyles(textpane);

        panel.add(new JScrollPane(textpane));
        add(panel);
        pack();

        setTitle("Document Model");
        setLocationRelativeTo(null);
        setDefaultCloseOperation(EXIT_ON_CLOSE);
    }

    private void createToolbar() {

        JToolBar toolbar = new JToolBar();

        ImageIcon bold = new ImageIcon("bold.png");
        ImageIcon italic = new ImageIcon("italic.png");
        ImageIcon strike = new ImageIcon("strike.png");
        ImageIcon underline = new ImageIcon("underline.png");

        JButton boldb = new JButton(bold);
        JButton italb = new JButton(italic);
        JButton strib = new JButton(strike);
        JButton undeb = new JButton(underline);

        toolbar.add(boldb);
        toolbar.add(italb);
        toolbar.add(strib);
        toolbar.add(undeb);

        add(toolbar, BorderLayout.NORTH);
        boldb.addActionListener(new ActionListener() {

            @Override
            public void actionPerformed(ActionEvent e) {
                sdoc.setCharacterAttributes(textpane.getSelectionStart(),
                        textpane.getSelectionEnd() -
textpane.getSelectionStart(),
                        textpane.getStyle("Bold"), false);
            }
        });

        italb.addActionListener(new ActionListener() {

            @Override
            public void actionPerformed(ActionEvent e) {
                sdoc.setCharacterAttributes(textpane.getSelectionStart(),
                        textpane.getSelectionEnd() -
textpane.getSelectionStart(),
                        textpane.getStyle("Italic"), false);
            }

        });

        strib.addActionListener(new ActionListener() {

            @Override
            public void actionPerformed(ActionEvent e) {
                sdoc.setCharacterAttributes(textpane.getSelectionStart(),
                        textpane.getSelectionEnd() -
textpane.getSelectionStart(),
                        textpane.getStyle("Strike"), false);
            }

        });

        undeb.addActionListener(new ActionListener() {

            @Override
            public void actionPerformed(ActionEvent e) {
                sdoc.setCharacterAttributes(textpane.getSelectionStart(),
                        textpane.getSelectionEnd() -
textpane.getSelectionStart(),
                        textpane.getStyle("Underline"), false);
            }
        });

    }

    private void initStyles(JTextPane textpane) {

        Style style = textpane.addStyle("Bold", null);
        StyleConstants.setBold(style, true);

        style = textpane.addStyle("Italic", null);
        StyleConstants.setItalic(style, true);

        style = textpane.addStyle("Underline", null);
        StyleConstants.setUnderline(style, true);

        style = textpane.addStyle("Strike", null);
        StyleConstants.setStrikeThrough(style, true);
    }

    public static void main(String[] args) {

        EventQueue.invokeLater(new Runnable() {
            @Override
            public void run() {
                DocumentModel ex = new DocumentModel();
                ex.setVisible(true);
            }
        });
    }
}
```

The example has a text pane and a toolbar. In the toolbar, we have four buttons that change attributes of the text.

```
sdoc = textpane.getStyledDocument();
```

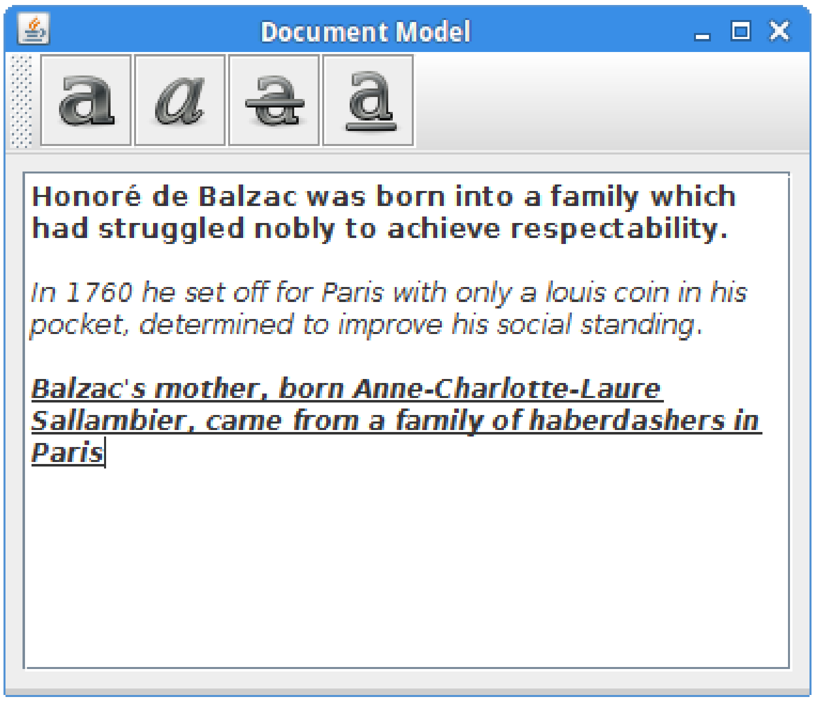
Here we get the styled document which is a model for the text pane component.

```
Style style = textpane.addStyle("Bold", null);
StyleConstants.setBold(style, true);
```

A style is a set of text attributes, such as colour and size. Here we register a bold style for the text pane component. The registered styles can be retrieved at any time.

```
doc.setCharacterAttributes(textpane.getSelectionStart(),
    textpane.getSelectionEnd() - textpane.getSelectionStart(),
    textpane.getStyle("Bold"), false);
```

Here we change the attributes of the text. The parameters are the offset and length of the selection, the style and the boolean value replace. The offset is the beginning of the text where we apply the bold text. We get the length value by substracting the selection end and selection start values. Boolean value false means that we are not replacing an old style with a new one, but we merge them. This means that if the text is underlined and we make it bold, the result is an underlined bold text.



Honoré de Balzac was born into a family which had struggled nobly to achieve respectability.

In 1760 he set off for Paris with only a louis coin in his pocket, determined to improve his social standing.

Balzac's mother, born Anne-Charlotte-Laure Sallambier, came from a family of haberdashers in Paris

Figure: Document model

In this chapter we have mentioned Swing models.