PATTERNS ON THE TABLE

James W. Cooper

The more I look into the details of the objects in the JFC (Swing) library, the more impressed I am with the care and thought that went into their design, and the power and flexibility they provide for my programs. A particular case in point is the JTable object, which provides a fairly easy way to display pretty sophisticated tables on the screen. And, as we'll see in this article, using the JTable also exemplifies several of the Design Patterns we've discussed already.

In the simplest case, you can invoke the JTable class very simply using standard default settings and it will display the contents of any rectangular array of data you pass to it. We start our example by deriving a subclass from the JxFrame class we introduced two months ago.

```
public class JxFrame extends JFrame {
    public JxFrame(String title) {
        super(title);
        setCloseClick();
        setLF();
    }
    private void setCloseClick() {
        //create window listener to respond to window close click
        addWindowListener(new WindowAdapter() {
                              public void windowClosing(WindowEvent e) {
                                  System.exit(0);
                              }
                          });
    }
    //-------------------------------------------
    private void setLF() {
        // Force SwingApp to come up in the System L&F
        String laf = UIManager.getSystemLookAndFeelClassName();
        try {
            UIManager.setLookAndFeel(laf);
        } catch (UnsupportedLookAndFeelException exc) {
            System.err.println("Warning: UnsupportedLookAndFeel: " + laf);
        } catch (Exception exc) {
            System.err.println("Error loading " + laf + ": " + exc);
        }
    }
}
```

This class sets the JFrame to close when the close box is clicked, and sets the look and feel to that of the current platform.

```
public class SimpleTable extends JxFrame {
    public SimpleTable() {
        super("Simple table");
        JPanel jp = new JPanel();
        getContentPane().add(jp);
```

To create a simple JTable to display data, you can pass it a one-dimensional array of column labels and a two dimensional array of data:

```
       Object[] [] musicData = {
            {"Tschaikovsky", "1812 Overture", new Boolean(true)},
            {"Stravinsky", "Le Sacre", new Boolean(true)},
            {"Lennon","Eleanor Rigby", new Boolean(false)},
            {"Wagner", "Gotterdammerung", new Boolean(true)}
```

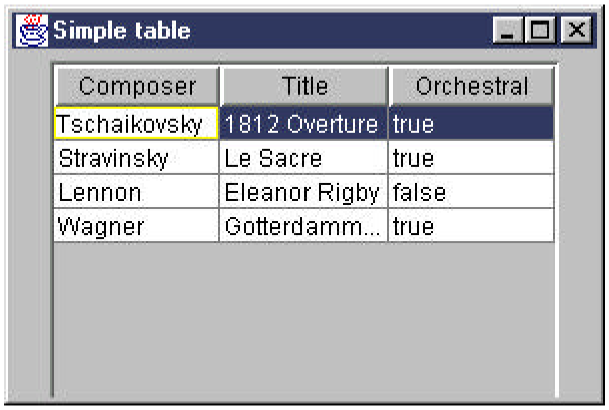
```
        };
        String[] columnNames = {"Composer", "Title", "Orchestral"};
        JTable table = new JTable(musicData, columnNames);
```

Creating a JTable this way takes these two arrays and uses them as the data model for the table display.

Like the JList, the JTable doesn't have its own scroll bars, and you add them by just putting the JTable inside a JScrollPane:

```
JScrollPane sp = new JScrollPane(table);
table.setPreferredScrollableViewportSize(new Dimension(250,170));
jp.add(sp);
```

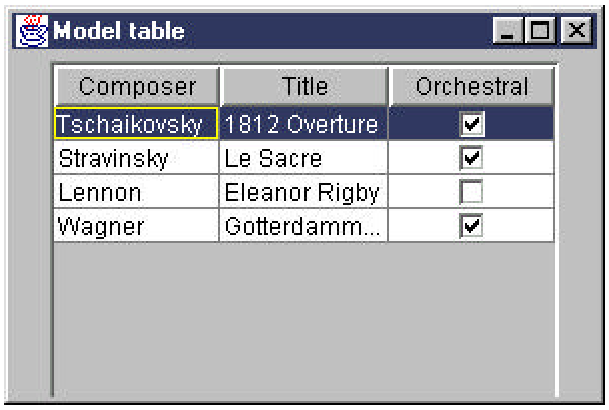
This generates the simple display shown in Figure 1.



Composer	Title	Orchestral
Tschaikovsky	1812 Overture	true
Stravinsky	Le Sacre	true
Lennon	Eleanor Rigby	false
Wagner	Gotterdamm...	true

Figure 1 – A simple table, loaded from two arrays.

This approach is very handy for quick displays of tables of data. And the cells in this table are *editable*: you can type in changes in any cell, and the underlying data array is changed automatically. But the JTable offers lots more flexibility to customize displays than that. If you can tell the JTable the class of the data in each column, it can display it in a suitable format for that class. For example, in Figure 2, we see the exact same data displayed, but with the boolean data represented as check boxes:



Composer	Title	Orchestral
Tschaikovsky	1812 Overture	☑
Stravinsky	Le Sacre	☑
Lennon	Eleanor Rigby	☐
Wagner	Gotterdamm...	☑

Figure 2 – The orchestral column is boolean and is shown as check boxes using a simple TableModel.

Using TableModels

You can accomplish this simple format change by introducing a TableModel class to hold the data and return it to the JTable display object. The TableModel is actually an interface which describes eight methods you must implement. However, it is easier to start by subclassing the AbstractTableModel class which contains default implementations for all of these methods. Then you need only subclass the methods you need to change. You should also note that the TableModel interface and the AbstractTableModel class are in the javax.swing.table package, rather than the javax.swing package, so you'll have to import both of them.

We'll create a subclass of AbstractTableModel called MusicModel.

```
public class MusicModel extends AbstractTableModel {
    String[] columnNames = {"Composer", "Title", "Orchestral"};
    Object[] [] musicData = {
        {"Tschaikovsky", "1812 Overture", new Boolean(true)},
        {"Stravinsky", "Le Sacre", new Boolean(true)},
        {"Lennon","Eleanor Rigby", new Boolean(false)},
        {"Wagner", "Gotterdammerung", new Boolean(true)}
    };

    private int rowCount, columnCount;
    //-------------------------------------------
    public MusicModel(int rowCnt, int colCnt) {
        rowCount = 4;
        columnCount = 3;
    }
    //-------------------------------------------
    public String getColumnName(int col) {
        return columnNames[col];
    }
    //-------------------------------------------
    public int getRowCount() {
        return rowCount;
    }
    //-------------------------------------------
    public int getColumnCount() {
        return columnCount;
    }
    //-------------------------------------------
    public Class getColumnClass( int col) {
        return getValueAt(0, col).getClass();
    }
    //-------------------------------------------
    public boolean isCellEditable(int row, int col) {
        return(col > 1);
    }
    //-------------------------------------------
    public void setValueAt(Object obj, int row, int col) {
        musicData[row][col] = obj;
        fireTableCellUpdated(row, col);
    }
    //-------------------------------------------
    public Object getValueAt(int row, int col) {
        return musicData[row][col];
    }
}
```

Note that we are still using the same two arrays of data we used before. The difference is that they are now created inside a data model class rather than in the calling program, and that we implement the *getValueAt* and *setValueAt* methods to get and set the data values. Therefore, you aren't limited to arrays and could return values from any type of underlying object or data structure. You may recognize this approach, since it is exactly the same as we used for the JList object. It also is an example of the Observer pattern: the data are observed by the display table object.

The other important difference is the *getColumnClass* method that returns the class of the data in that column. It is this information that allows the underlying rendering system to choose the right renderer for each class. The basic JTable rendering system will render Boolean as a checkbox, Number as a right-aligned label, ImageIcon as a centered label and Object as its String value.

Cell Renderers

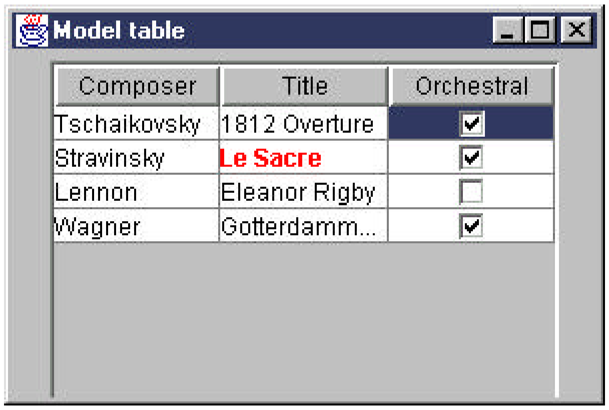
Of course, just as you can handle the data yourself, you can handle the rendering of the data yourself using your own renderer for each class of data. For example, if you want to render Strings in some unusual way, you need only tell the table the class that will handle String rendering:

```
table.setDefaultRenderer(String.class, new ourRenderer());
```

A cell renderer must implement the TableCellRenderer interface, which has only one method: *getTableCellRendererComponent*. In the simple example below, we create our component out of a JLabel component with this new method added:

```
public class ourRenderer extends JLabel
  implements TableCellRenderer {
    Font bold, plain;
    public ourRenderer() {
        super();
        setOpaque(true);
        setBackground(Color.white);
        bold = new Font("SansSerif", Font.BOLD, 12);
        plain = new Font("SansSerif", Font.PLAIN, 12);
        setFont(plain);
    }
    public Component getTableCellRendererComponent(
             JTable jt, Object value, boolean isSelected,
             boolean hasFocus, int row, int column) {
        setText((String)value);
        //render column 1, row 1 in red, bold
        //all others in black, plain
        if (row ==1 && column==1) {
            setFont(bold);
            setForeground(Color.red);
        } else {
            setFont(plain);
            setForeground(Color.black);
        }
        return this;
    }
}
```

Note that all this simple renderer does that is different is to return the JLabel with a bold, red font for row 1, column 1, and a plain, black font for all other cells, as illustrated in Figure 3. Note also that this renderer is only called for Strings. The Boolean cell renderer remains unchanged. The idea of the renderer can be confusing on first reading, because there is only one renderer for all the cells containing strings, and it changes its properties depending on specific data in the cells (in this case, that data in position (1,1)). Further, the cell renderer class is almost always the class that does the actual render*ing* and it usually returns *this* : its current instance.



Composer	Title	Orchestral
Tschaikovsky	1812 Overture	✓
Stravinsky	**Le Sacre**	✓
Lennon	Eleanor Rigby	☐
Wagner	Gotterdamm...	✓

Figure 3- The same data using a custom cell renderer to make cell (1,1) bold and red.

Rendering Other Kinds of Classes

Now lets suppose that we have more complicated classes that we want to render. Suppose we have written a document mail system and want to display each document according to its type. However, the documents don't have easily located titles and we can only display them by author and type. Since we intend that each document could be mailed, we'll start by creating an interface Mail to describe the properties the various document types have in common:

```
public interface Mail {
    public ImageIcon getIcon();
    public String getLabel();
    public String getText();
}
```

So each type of document will have a simple label (the author) and a method for getting the full text (which we will not use here). Most important, each document type will have its own icon, which you can obtain with the *getIcon* method.

For example, the NewMail class would look like this:

```
public class NewMail implements Mail {
    private String label;

    public NewMail(String mlabel) {
        label = mlabel;
    }
    public ImageIcon getIcon () {
        return new ImageIcon("images/mail.gif");
    }
    public String getText() {
        return "";
    }
    public String getLabel() {
        return label;
    }
}
```

The renderer for these types of mail documents is just one which gets the icon and label text and uses the DefaultCellRenderer (derived from Jlabel) to render them:

```
public class myRenderer extends DefaultTableCellRenderer {
    private Mediator md;

    public myRenderer(Mediator med) {
        setHorizontalAlignment(JLabel.LEADING);
        md = med;
    }
 //-------------------------------
    public Component getTableCellRendererComponent(
             JTable table, Object value, boolean isSelected,
             boolean hasFocus, int row, int col) {
        if (hasFocus) {
            setBackground(Color.lightGray );
            md.tableClicked ();
        } else
            setBackground(new Color(0xffffce));
        if (value != null) {
            Mail ml = (Mail) value;
            String title = ml.getLabel ();
            setText(title);           //set the text
            setIcon(ml.getIcon ());   //and the icon
        }
        return this;

    }
}
```

Since one of the arguments to the getTableCellRendererComponent method is whether the cell is selected, we have an easy way to return a somewhat different display when the cell is selected. In this case, we return a gray background instead of a white one. However, we could set a different Border as well if we wanted to. A display of the program is shown in Figure 4.

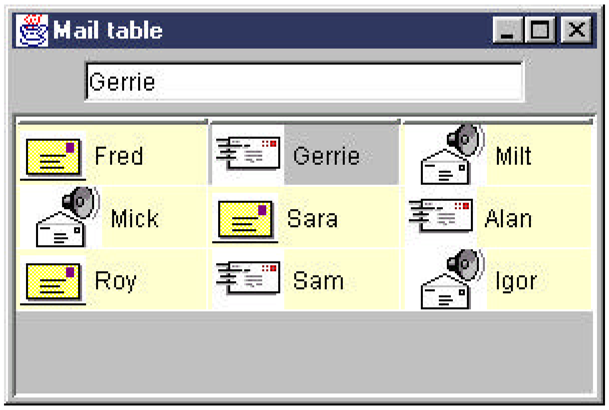


Figure 4 - Rendering objects which implement the Mail interface, using their icon and text properties. Note the selected cell is shown with a gray background.

Selecting Cells in a Table

You can select a row, separated rows, a contiguous set of rows or single cells of a table, depending on the list selection mode you choose. In this example, we want to be able to select single cells, so we choose the single selection mode:

```
setSelectionMode(ListSelectionModel.SINGLE_SELECTION);
```

If we want to take a certain action when table cell is selected we get the ListSelectionModel from the table and add a list selection listener to it.

```
ListSelectionModel lsm =  getSelectionModel();
lsm.addListSelectionListener (new TableSelectionListener(med));
```

The TableSelectionListener class we create just implements the *valueChanged* method and passes this information to a Mediator class:

```
public class TableSelectionListener implements ListSelectionListener {

   private Mediator md;

   public TableSelectionListener(Mediator med) {
      md= med;
   }
   public void valueChanged(ListSelectionEvent e) {
      ListSelectionModel lsm =
         (ListSelectionModel)e.getSource();
      if( ! lsm.isSelectionEmpty ())
        md.tableClicked();
   }
}
```

The Mediator class, as we have discussed before, is the only one that deals with interactions between separate visual objects. In this case, it just fetches the correct label for this class and displays it in the text field at the top of the window, as shown in Figure 4. Here is the entire Mediator class:

```
public class Mediator {
    Ftable ftable;
    JTextField txt;
    int tableRow, tableColumn;
    FolderModel fmodel;

    public Mediator ( JTextField tx) {
        txt = tx;
    }
    public void setTable(Ftable tbl) {
        ftable = tbl;
    }
    //-------------------------------
    public void tableClicked() {
        int row = ftable.getSelectedRow ();
        int col = ftable.getSelectedColumn ();
//don't refresh if not changed
        if ((row != tableRow) || (col != tableColumn)) {
            tableRow = row;
            tableColumn = col;
            fmodel = ftable.getTableModel ();
            Mail ml =  fmodel.getDoc(row, col);
            txt.setText (ml.getLabel ());

       }
    }
}
```

Patterns in this Image Table

If you look at the UML diagram in Figure 5, you can easily see a number of common Design Patterns in the program:

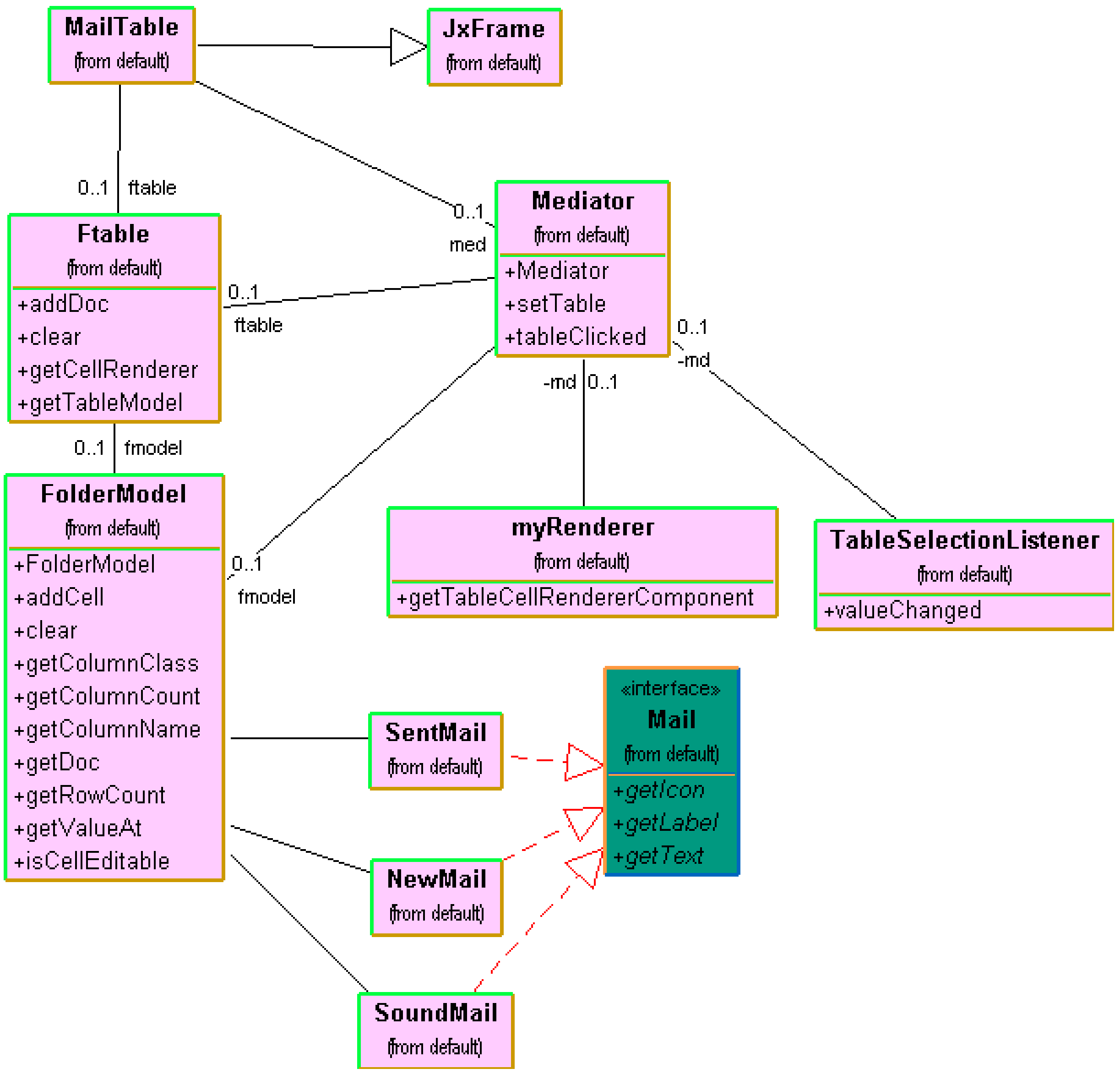


Figure 5 – A UML Diagram of the classes in the Image table.

As we noted earlier, the separation of the data from the viewer in the Swing JTable amounts to an Observer pattern. Further, by catching the table selection in a listener and then sending the result to a Mediator class, we are implementing the Mediator pattern. Finally, having the 3 mail classes which each produce a different icon is a Factory Method pattern. These patterns occur all through Java programming, especially when you use the Swing classes, and it is extremely helpful to recognize how often you can refer to patterns in your every day programming to simplify communication between you and your colleagues.