STUDIA UNIV. BABEŞ–BOLYAI, INFORMATICA, Volume **XLVII**, Number 1, 2002

A PROPERTY SHEET

SORIN MOLDOVAN

Abstract. The graphical user interfaces (GUI) are very important today. There is no application without designing a friendly, easy to understand and accessible environment, both for beginners and for expert users. The aim of this article is to analyze some aspects specific to graphical interfaces provided by CAD and CASE tools and to offer a solution for the programmers developing components for such a tool in JAVA programming language. This solution is illustrated in the design and implementation of a property sheet used to display general and specific properties for the UML entities the user works with.

1. Why such a component?

The objects with which the user operates are, most of the time, complex and vast, having lots of attributes (properties). For example, Visual Basic users, when they create a form, have to manage, for each graphical component, a large list of properties (name, position, color, font, text, etc.). It would be a good idea if these characteristic attributes of an object were grouped in several categories. At a certain moment, the user is not interested in all the properties, but only in some of them. Therefore, some categories should be summarized, rather than detailed. Through this facility, we can dynamically control the level of detail at which we study the object and also have a custom view upon an object.

Another aspect is related to editing the attribute values together with their validation. By referring again to the example about the Visual Basic forms, we notice that attributes such as name, position, can have an infinite (or at least very large) domain of values. For editing these attributes, we will use controls of type TextField (they may be specialized for alpha-numeric or only numeric values, or may have additional constraints). Some other properties can only have a finite domain of values (sometimes very small – for example, the visibility can be either True or False). In this case, instead of the user providing a value as input and then checking if the value is valid or not, the component will provide a list of values (which will cover the whole domain), and the user will choose one of these

2000 *Mathematics Subject Classification.* 68N30.

1998 *CR Categories and Descriptors.* D.2.3 [**Software**] : Software Engineering – *Coding Tools and Techniques*; D.2.7 [**Software**] : Software Engineering – *Distribution, Maintenance and Enhancements* .

possible values. For properties such as the used font, editing is more complex and we will open a dialog (probably a system dialog). Between some objects there are dependency relationships (*object1* "owned by" *object2*, for instance). If the properties window of *object1* has the "Owner" property with a value of *object2*, then in some cases we can navigate the relationship from *object1* towards *object2* through the property sheet.

In the framework of the OCL Evaluator project the need of such property sheet occurred. In addition to the model browser and diagrams which offer a view upon the whole project or a part of it, one may have the possibility to inspect a certain element and to modify its state. It acts as a view and a controller on the UML model. Figure 1 illustrates how class properties are displayed. What this example does not reveal is the possibility to edit each property using a specific control depending of the kind of information the property contains.

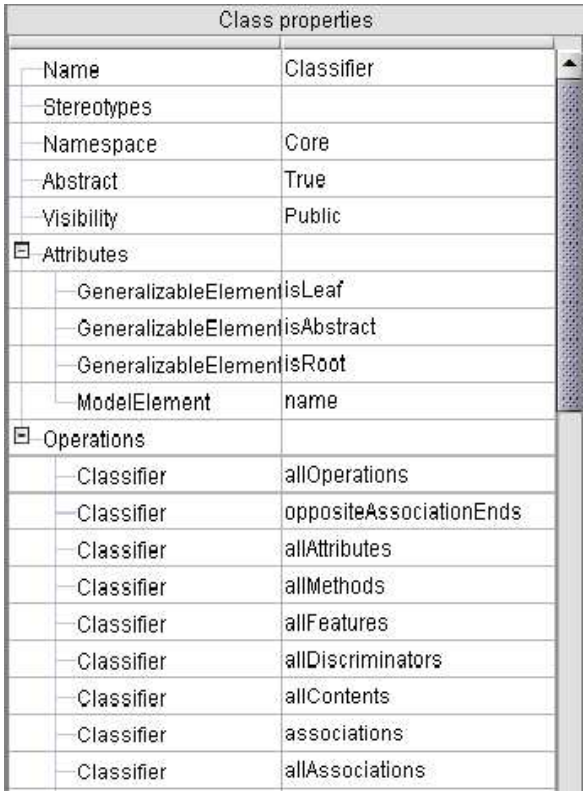

FIGURE 1. The property sheet in a CASE tool

The implementation of the property sheet was realized in JAVA programming language using the Swing package. First of all, we present the main technique we used.

2. Classic Model-View-Controller (MVC) Architecture

The MVC Architecture is designed for applications that need to provide multiple vies of the same data. MVC separates applications into three types of objects:

Model: Maintain data and provide data accessor methods; is the application object.

Views: Paint a visual representation of some or all of a model's data

Controller: Defines the way the user interface reacts to user input.

Models are responsible for maintaining data; for example a notepad application would store the current document's text in a model. Models typically provide methods to access and modify their data. Model also fires events to registered views when a model is changed, and the views respond by updating themselves based on the model change.

Views are responsible for providing a visual representation of some portion of a model's data. For example, a notepad application would provide a view of the current document by displaying some or all of the text stored in the model.

Controllers handle events for views. Swing listeners (such as mouse and action listeners) are MVC controllers. The notepad application mentioned previously would have mouse and key listeners that made changes to the model or view as appropriate.

Before MVC, user interface designs tended to consider these objects together. MVC decouples them to increase flexibility and reuse. MVC is a powerful design for a number of reasons. First, multiple views and controllers can be plugged into a single model, which is the basis for Swing's pluggable look and feel.

Second, a model's views are automatically notified when the model is changed, changing a model property in one view results in subsequent updates of the model's other views.

Third, because model is not dependent upon views, models do not have to be modified to accommodate new types of views and controllers.

We will refer to MVC architecture when we'll describe the interaction between the property sheet and the UML model.

Swing MVC is a specialized version of classic MVC meant to support pluggable look and feel instead of applications in general. Swing lightweight components consist of the following objects:

- a model that maintains a component's data;
- a UI delegate that is a view with listeners for handling events;
- a component that extends JComponent class.

Swing models translate directly to classic MVC models. The components delegate their look and feel to a UI delegate. UI delegates correspond to a view-controller combination in classic MVC. Controllers are referred to as listeners from here on.

Taking into account the Swing MVC architecture we describe how we have create the JTreeTable class used in our view component upon the UML model.

3. JTreeTable

A *TreeTable* is a combination of a Tree and a Table – a component capable of both expanding and contracting rows, as well as showing multiple columns of data. The Swing package does not contain a JTreeTable component, but it is fairly easy to create one by installing a JTree as a renderer for the cells in a JTable.

In Swing, the JTree, JTable, JList, and JComboBox components use a single delegate object called a *cell renderer* to draw their contents. In fact it is the **view** from the MVC pattern. A cell renderer is a component whose *paint()* method is used to draw each item in a list, each node in a tree, or each cell in a table. A cell renderer component can be viewed as a "rubber stamp": it's moved into each cell location using *setBounds()*, and is then drawn with the component's *paint()* method.

By using a component to render cells, you can achieve the effect of displaying a large number of components for the cost of creating just one. By default, the Swing components that employ cell renderers simply use a JLabel, which supports the drawing of simple combinations of text and an icon. To use any Swing component as a cell renderer, all you have to do is create a subclass that implements the appropriate cell renderer interface: TableCellRenderer for JTable, ListCellRenderer for JList, and so on.

4. Rendering in Swing

Here's an example of how you can extend a JCheckBox to act as a renderer in a JTable:

```
public class CheckBoxRenderer extends JCheckBox
      implements TableCellRenderer {
   public Component getTableCellRendererComponent(JTable table,
      Object value, boolean isSelected,
      boolean hasFocus, int row, int column) {
         setSelected(((Boolean)value).booleanValue());
         return this;
      }
}
```

5. How the example program works

The code showed above shows how to use a JTree as a renderer inside a JTable. This is a slightly unusual case because it uses the JTree to paint a single node in each cell of the table rather than painting a complete copy of the tree in each of the cells. We start in the usual way: expanding the JTree into a cell render by extending it to implement the TableCellRenderer interface. To implement the required behavior or a cell renderer, we must arrange for our renderer to paint just the node of the tree that is visible in a particular cell. One simple way to achieve this is to override the *setBounds()* and *paint()* methods, as follows:

```
public class TreeTableCellRenderer extends JTree
      implements TableCellRenderer {
   protected int visibleRow;
   public void setBounds(int x, int y, int w, int h) {
      super.setBounds(x, 0, w, table.getHeight());
   }
   public void paint(Graphics g) {
      g.translate(0, -visibleRow * getRowHeight());
      super.paint(g);
   }
   public Component getTableCellRendererComponent(JTable table,
      object value,
      boolean isSelected,
      boolean hasFocus,
      int row, int column) {
         visibleRow = row;
         return this;
      }
}
```

As each cell is painted, the JTable goes through the usual process of getting the renderer, setting its bounds, and asking it to paint. In this case, though, we record the row number of the cell being painted in an instance variable named *visibleRow*. We also override *setBounds()*, so that the JTree remains the same height as the JTable, despite the JTable's attempts to set its bounds to fit the dimensions of the cell being painted.

To complete this technique we override *paint()*, making use of the stored variable *visibleRow*, an operation that effectively moves the clipping rectangle over the appropriate part of the tree. The result is that the JTree draws just one of its nodes each time the table requests it to paint.

In addition to installing the JTree as a renderer for the cells in the first column, we install the JTree as the editor for these cells also. The effect of this strategy

is the JTable then passes all mouse and keyboard events to this "editor" – thus allowing the tree to expand and contract its nodes as a result of user input.

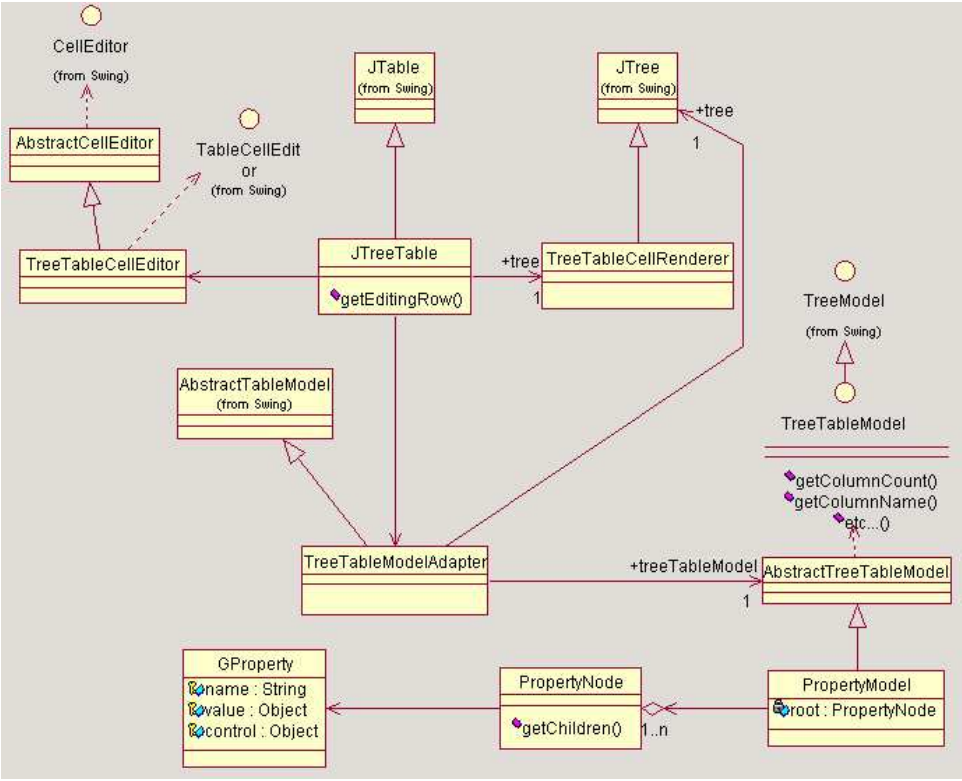

FIGURE 2. TreeTable Architecture

6. A different editor for each row

We mentioned above that this component has a specific editor depending on the values domain range. So we subclass DefaultCellEditor and name it TableCellEditor. This class knows what editor is responsible for editing of a row. A private member *editors* contains in a HashSet all the editors used. It is not necessary for each row to provide an editor. If a row has no editor associatd a default editor is provided.

We add a method to this class: *selectEditor(MouseEvent e)*. This method (invoked by *isCellEditable(EventObject anEvent)* and *shouldSelectCell(EventObject anEvent)*) calculate the row being edited and sets the *editor* member to the coresponding editor (from the hash set).

Invocation of a method of the TableCellEditor (which is set as the CellEditor for the second column of the view) leads to invocation of the same method upon the selected editor.

7. Interaction between UML model and the property sheet

To acquire the information from the UML model to be displayed a hierarchy of adapter classes is created. This is similar with the metamodel class hierarchy. These classes have a method *getProperties(Element element)* for gathering properties and a method *propertyChanged(GProperty prop)* responsible for modification of properties of the current object (in our tool this responsibility is delegated a class who performs all operations upon the model). This method also made some check upon the new value of the attribute. It returns true if the update was successfully and false otherwise.

The steps performed when the properties of an element need to be shown are (notice the use, once more, of the MVC pattern):

- setting up the target element;
- invocation of the *updateView* method;
- creation of a new *PropertyModel* used by the JTreeTable to display the information;
- gathering information (after an adapter class of the UML element is created "P..."); interogation of the **model**;
- creation of nodes using the collected properties (root node is created with the target element as argument and is never displayed); creating the **view**;
- editors for each property are set up; setting the **controllers**.

8. Conclusions and further developments

The goal of this article was to present a solution for developing graphical components used to inspect the treats of objects which the user operates. Even if the example above is from the CASE tools world we tried to provide a solution which can be very easy adapted to any kind of application. In terms of MVC, only the model has to be updated to correspond to the needs of the application. (One has to subclass and full implement the *TreeTableModel* interface – see **Figure 2**).

A very important point about the property sheet is the graphical aspect. In this version this is quite simple. Its improvement will increase also the quality. These are some aspects that can be improved:

- Adding colours. It is useful to color attributes listed with different colors to easily differentiate between them (e.g. In the class properties, the color of inherited features may differ from the color of the ones defined by the class). Also, the attributes that cannot be modified may have a specific color to suggest they are read-only.

- Attaching icons to the leafs of the tree to provide additional information of the property (e.g. The features may have attached icons to illustrate their visibility).
- A pop-up menu context dependent and keyboard shortcuts to improve the editing proces of the model.

References

[1] Sun Java Tutorial, `http://java.sun.com/docs/books/tutorial/index.html`
[2] David M. Geary, *Graphic JAVA. Mastering the JFC*, vol II, Palo Alto, CA, 1999.
[3] E. Gamma, R. Helm, R. Johnson, J. Vlissides, *Design Patterns*, Addison Wesley, Boston, MA, 1994.
[4] OMG Unified Modeling Language Specification, `http://www.omg.org`

Babeş-Bolyai University, Computer Science Research Laboratory, RO 3400 Cluj-Napoca, Str. Kogălniceanu 1, Romania
E-mail address: `sorin@lci.cs.ubbcluj.ro`