Submission Title
GUI Composition: TableLayout & TablePanel

Contact Information
Richard Rasala
rasala@ccs.neu.edu
Viera K. Proulx
vkp@ccs.neu.edu
College of Computer & Information Science
Northeastern University
Boston MA 02115

Problem Statement
A central problem in GUI construction is the composition of the individual GUI widgets into some organized arrangement within a panel or frame. To avoid manual pixel positioning, Java uses layout managers to layout components. Unfortunately, the layout managers provided by Java are very problematic. The simple ones do not do enough and the advanced ones are very hard to use and often do not produce the expected results after all the work.

The Java Power Tools (JPT) introduce a powerful layout manager called TableLayout that solves the problems of one and two dimensional table layout and seems to eliminate the need for all of the other standard Java layout managers. Furthermore, the panel TablePanel which uses TableLayout as its manager allows the designer to add and arrange objects in a panel in a single step.

Solution Overview: Examples
Scenario 1: A mini-application to explore painting of circles.
In the mini-application shown below, the user can set the radius, center, and color of a circle and then paint the circle. More than one circle may be painted in the graphics window. The clear button will erase the window so the user can start fresh.

We will explain how the GUI for this mini-application is built using TablePanel. We assume that the GUI widgets have already been defined. Specifically, we assume that we have three text field views radiusTFV, xTFV, yTFV; one view colorView to select the color; two actions paint and clear to encapsulate the behavior; and one graphics panel window.

The first step is to define a panel to enclose the three text fields and their labels.

```
TablePanel inputPanel = new TablePanel(
    new Object[] {
        { "Radius:", radiusTFV },
        { "X:", xTFV },
        { "Y:", yTFV } },
    5, 5, EAST);
```

Notice that the layout of the Object[] array exactly parallels the layout of the components in the GUI. Notice also that we pass the labels as String objects. The TablePanel constructor will automatically convert these labels into GUI components. The small constants 5, 5 specify the horizontal and vertical gap between cells in the table. The constant EAST specifies that cell contents should be aligned to the EAST edge of each cell.

The next step is to define the full vertical controls panel on the left hand side.

```
TablePanel controls = new TablePanel(
    new Object[] { inputPanel, colorView, paint, clear },
    VERTICAL, 10, 10, CENTER);
```

In this case, we use a one dimensional object[] array so we must specify the layout direction (VERTICAL in this case). The constant CENTER specifies that cell contents should be centered in each cell. Notice that the action objects paint and clear are automatically used to define corresponding JButton objects in the GUI.

The third step is to join the controls and the graphics window to form the main panel.

```
TablePanel mainPanel = new TablePanel(
    new Object[] { controls, window },
    HORIZONTAL, 5, 5, NORTH);
```

The final step is to place the main panel in an enclosing frame object that is packed, centered on the screen, and made visible. This work is done via a single call in the main method.

```
public static void main(String[] args) {
    JPTFrame.createQuickJPTFrame
        ("Circle Sample", new CircleSample());
}
```

Aside: In the submission on the Java Power Framework, we have shown how this problem is solved in that context. The automatic GUI generated by JPF for the paint method contains the three text fields and the color view but no labels. The paint method makes use of the graphics window available for all methods in the JPF Methods class so no separate graphics window is needed.

Scenario 2: The Kaleidoscope case study.
To give an idea of how TablePanel and TableLayout make GUI composition easy even in more complicated situations, we will describe the GUI construction for the Kaleidoscope Case Study that is available on the JPT web site. A screen snapshot of the Kaleidoscope GUI is shown below.
The main panel is composed of the left-hand-side panel which has numerous controls and options and the right-hand-side panel which has the graphics panel for displaying the kaleidoscope animation and the run and stop buttons to start and stop the animation.

If we follow the design pattern used in the circle sample, we will define `mainPanel` as follows:

```java
TablePanel mainPanel =
    new TablePanel(
        new Object[] { lhsPanel, rhsPanel },
        HORIZONTAL, 20, 20, NORTH);
```

Of course, this would require the prior definition of `lhsPanel` and `rhsPanel`. In turn the definition of these two panels would require the prior definition of other panels or primitive widgets. We call this definition style the bottom-up style. One must start with the smallest GUI elements and build larger components in an incremental fashion.

For more complex GUIs, we prefer a top-down or functional style that uses factory methods and getters to assemble the requisite components. Thus, in Kaleidoscope, the actual code to create the main panel is as follows:

```java
public TablePanel makeMainPanel() {
    return new TablePanel(
        new Object[] { makeLHSPanel(), makeRHSPanel() },
        HORIZONTAL, 20, 20, NORTH);
}
```

Notice that `TablePanel` is being used in precisely the same way as before but we have gained in abstraction because it is easier to change the return value of a method than to replace an explicitly named panel object. The functional approach also allows us to refactor the GUI code so that the construction of GUI elements is distributed across several classes.

Let us now sketch the rest of the details of the construction of the Kaleidoscope GUI. In the left panel, we see three panels laid out vertically using a `TablePanel`. Each panel is wrapped in a titled border which is easy to do in the functional style via a wrapper class. The most interesting of these three panels is the top panel which has 5 major components that are arranged once again using another `TablePanel`. Among these 5 components are 2 groups of radio buttons. Although the radio groups are not instances of `TablePanel`, they use `TableLayout` for their layout.

On the right side, we see a vertical `TablePanel` with a graphics window on top and a horizontal pair of buttons on bottom which are of course organized via an inner `TablePanel`.

This quick overview of the construction of the Kaleidoscope GUI illustrates why the use of `TablePanel` and `TableLayout` has made it possible for us to ignore the traditional Java layout managers.

### Solution Overview: Principles

`TableLayout` is designed for 2-dimensional table layouts but may be used for horizontal or vertical layouts as a special case. This layout manager attempts to layout components in such a way that each component is placed in a cell at least as large as its preferred size. If the cell has extra space then the component will be aligned in the center of the cell or at one of the eight compass directions as desired. The gaps between cell rows and columns may also be specified. The `TableLayout` constructor specifies the initial number of rows and columns in the table but this will be expanded if needed as components are added.

`TablePanel` is designed so that the user can both set layout and add all table components in a single constructor step. There are four families of `TablePanel` constructors:

- The simplest constructors supply structural data for the table but no content. Because the table is not finished at the time of construction, these constructors are used only rarely in practice.
- The next family of constructors takes a 1-dimensional array of `Object` to provide content and an orientation (horizontal or vertical) that determines how the content of the table will be displayed.
- The third family of constructors takes a 2-dimensional array of `Object` to provide content. The visual display of the table parallels the structural layout of the array.
- The final family of constructors takes an algorithmic object that implements `TableGenerator`. Then, given the number of rows and columns in the table, this object will algorithmically construct the contents.

In the examples, we have seen instance of the use of one and two dimensional arrays of `Object` to initialize a `TablePanel`. An important aspect the design is that we accept a general `Object` rather than require a `Component`. This enables the user to think conceptually and to allow the construction of `Components` to be encapsulated also.

The process of transforming an `Object` into a `Component` works as follows. A `Component` is inserted as is. A `String` or `Icon` is inserted into an `Annotation` object. A `Shape` is inserted into a `ShapePaintable` object and an `Image` into an `ImagePaintable`. In turn, a `Paintable` is inserted into a `JPTComponent`. An `Action` is inserted as a `JButton`. Finally, in all other cases, `null` is inserted which simply leaves a potential gap in the table.

**Aside:** The `Paintable` concepts will be discussed in a separate submission.

The last constructor family that uses a `TableGenerator` object is designed for those situations where the exact contents of the table cannot be known at compile time. This has pedagogical uses when building algorithm animations since what is being animated may depend on user choices at runtime. For example, in a bar chart animation of an array sorting algorithm, the user may change the number of items being sorted and/or the array data after each run. Using a `TableGenerator`, the bar chart that represents the array data can be regenerated algorithmically.

### Experience with the Solution

The variety of ways to build a table is quite helpful to students and faculty because it means that the right tool is available to do the desired GUI construction task. In practice, GUI frames are built using nested `TablePanels`. Once the main panel is built, there is a one statement command to create its enclosing frame, pack it, and make it visible.

We have described a rapid process for GUI composition that starts with arrays of objects, uses recursive nesting, and then creates the final frame in one step. We have found that there is a qualitative difference in how an entity is built if one can create the entity as a whole rather than have to use a sequence of `add` methods to create the entity incrementally. For complex GUIs, our approach exhibits a functional/declarative style that adds significantly to the clarity of the design process.

### API Documentation & Related Materials

The main JPT site to access documentation, code, and the jpt.jar:

http://www.ccs.neu.edu/jpt/