The Visual Library is a NetBeans component for creating and drawing graphs and diagrams, such as UML artifacts, navigation schemes for MIDP and others. But with a little effort it can have many more creative uses.
The Visual Library has been conceived mainly for building, handling and rendering graphs, including flow and UML diagrams (it has been originally developed for the NetBeans Mobility Pack, but from NetBeans 6.0 on it’s part of the NetBeans Platform APIs). I like to think of the Visual Library more generally, however, as an API for creating interactive “whiteboards” where you can place, move, and rearrange items visually.

From this perspective, the Visual Library reveals all its power, as modern UIs are each day more focused on the concept of modeling “real-life” objects that can be moved around. Thus it found its way into blueMarine, an open-source photo management application I created which is based on the NetBeans Platform (see my article in Issue 3 of NetBeans Magazine for more about this tool).

In blueMarine, the Visual Library is used to implement a virtual version of a photographer’s “Light Table” – a place where photos can be laid out and rearranged (see Figure 1). It’s also the basis for an advanced geotagging component (Figure 2). In this article I’ll describe the Light Table, which is the simpler of the two Visual Library-based components, but complex enough that we can show in practice many features of the API.

Though all the examples illustrated in this article refer to a NetBeans Platform application, you can use the Visual Library in plain Swing apps by adding a couple of JAR files to the classpath.
Basic concepts

Let’s first introduce some key concepts of the Visual Library. **Widgets** (the `org.netbeans.api.visual.widget.Widget` class and its descendants) represent a diagram’s nodes. A widget can consist of a simple drawing, a piece of text, an icon, or a group of these basic elements. It can also wrap a Swing component. You may need to subclass `Widget` for special purposes, but in most cases you'll be fine with one of the provided widget classes such as `ImageWidget` and `LabelWidget`.

**Connection widgets**, in particular, represent arcs that connect pairs of other widgets. They are usually drawn as arrows (with some graphic variants for line caps). Also, their paths can be “routed” using different algorithms, for example to avoid clutter in diagrams.

Other essential elements are **Actions** and **ActionFactories**. You probably won’t be satisfied by just creating a diagram and staring at it, so there’s plenty of support for making diagrams dynamic and interactive. It’s possible to create, delete and select widgets, and have them change appearance when you hover over them. You can additionally drag, connect and disconnect widgets. The Visual Library provides **Actions** and corresponding factories to perform these tasks, and allows you to customize their behavior.

Finally, a **scene** (`org.netbeans.api.visual.widget.Scene`) is the container for everything (in my initial analogy it represents the actual “whiteboard”).

In this article I assume the reader is quite confident with these basic concepts. There are already various tutorials available on the library’s website where you can learn the essentials. Here I present a more advanced, let’s say “creative”, use of the library.

Getting the code

You can download the full working code described in this article by using Subversion:

```
svn checkout \https://bluemarine-incubator.dev.java.net/svn/bluemarine-incubator/trunk/src/LightTable -r 232 \ --username guest
```

Revision 232 is the one matching the code listings in this article.

The Light Table

The Light Table lives inside a **TopComponent** (in a plain Swing application you would use a **JFrame** or a **JPanel**).

If you’re not familiar with the NetBeans Platform, a **TopComponent** is sort of a hybrid between a **JFrame** and a **JPanel**. It is normally docked, thus behaving as a **JPanel**, but can be undocked and float around like a **JFrame**. It is usually the container used for a user interface in NetBeans Platform applications.

A number of required objects are initialized in the **LightTableTopComponent** (see Listing 1):

- **scene** – An instance of `ObjectScene`, a class that keeps associations between widgets and the objects that model them. This facility is quite useful for implementing the MVC pattern.
- **view** – A Swing **JComponent** that renders the objects in the **scene**. You obtain it by asking the **scene** object for its creation.
- **mainLayer** – The Visual Library

Listing 1. Basic objects for the Visual Library in LightTableTopComponent.java

```java
private final ObjectScene scene = new ObjectScene();
private final JComponent view = scene.createView();
private final JComponent mainLayer = new LayerWidget(scene);
private final JComponent satelliteView = scene.createSatelliteView();
```

Listing 2. Initializing components in LightTableTopComponent

```java
spScrollPane.setViewportView(view);
pnLayeredPane.add(spScrollPane, JLayeredPane.DEFAULT_LAYER);

// pnSatelliteView is a JPanel wrapping the satelliteView
pnLayeredPane.add(pnSatelliteView, JLayeredPane.PALETTE_LAYER);
scene.addChild(mainLayer);
scene.getActions().addAction(ActionFactory.createZoomAction());
```
usually organizes widgets in different layers. This can be handy especially for improving performance of graphs with connections. Our Light Table does not need connections so we’re fine with a single layer.

- **satelliteView** – This JComponent will render a “bird’s eye view” of the scene, useful if you’re going to create a large scene that extends beyond the screen size.

**Listing 2** shows how the initialization is completed. You usually need to place the scene in a JScrollPane, to allow users to navigate it. We also use a JLayeredPane to render the satellite view above the scene in a corner. In the last line we enable zooming by creating a “zoom action” through the ActionFactory class and adding it to the scene.

**Adding and removing objects**

We also need to include a couple of methods for adding and removing photos in the Light Table; see **Listing 3**. In this code, when a new DataObject is added we perform the following steps (inside the internalAdd() method):

1. Create the widget that represents the DataObject in the scene (the ThumbnailWidget, discussed below).
2. Assign its initial position (with some code needed to convert a Swing Point to a proper value in the Visual Library’s scene model coordinates).
3. Add the widget to the layer.
4. Add the widget and the DataObject to the scene (which binds them together).
5. Define the dynamic behavior of the widget by adding some actions. Many actions are created through ActionFactory, while others can be instantiated with specific methods of the scene object.
6. Finally, call scene.validate(). After one or more widgets are changed, the scene needs to be revalidated. The Visual Li-
library usually does this automatically, but as we’re writing customized Swing code that performs a change we need to do this manually.

Before we go on, let me discuss some basic concepts about the way the NetBeans Platform implements the MVC pattern. In the Platform, the `DataObject` class is used to model a domain object, and an associated class – `Node` – is used to model its representation inside a view (a list, a tree, etc.). For example, `Nodes` contain a text label, an icon, and a list of associated actions which can be activated by a popup menu.

It’s a very good thing to have two distinct classes, since you can have multiple `Nodes` for each `DataObject`. This lets you create different representations of the same domain object. You usually subclass `DataObject` and add your application-specific logic. For example, blueMarine defines a `PhotoDataObject` class which contains code for reading and writing an image. However, you won’t see this class in the code in this article because I’m following a best practice of keeping models as general as possible, by working with plain `DataObjects` and delegating everything to the related `Node` classes. Thus, I could use the same code, e.g. for rendering movies (with a `MovieDataObject`) or other visual documents.

Now let’s go back to the `LightTableTop-Component`.

**Widget behavior**

We want users to be able to select our widget, resize it by dragging its borders, and move it by dragging its contents. Also, the widget should change appearance when the mouse hovers over it, and come to the top of the stack when clicked. For movement, selection and hovering, adding...
Creative Uses of the Visual Library

explicit class casts (for example, as the method boundSuggested() is general, it deals with a Widget rather than with my ThumbnailWidget). Instead, I’ve used a Lookup, a very useful class from the NetBeans Platform (which is also available for use in plain Swing projects). It acts as a container of custom objects, which can be retrieved by specifying their class name. In Listing 4, you see that the thumbnail is retrieved by getLookup().lookup(Thumbnail.class). When we discuss the ThumbnailWidget class we’ll see how the Thumbnail object was made available.

Listing 5 shows the code for bringing the widget to the front. This is an example of how you can define new actions. I’ve extended the WidgetAction.Adapter class, which gets invoked by mouse and keyboard listeners, and overridden the relevant method.

Actions are bound to the widget by defining a “pipeline” to which mouse and keyboard events are delivered. Sometimes an event is propagated through the whole pipeline; in other cases a certain action consumes it definitely. The propagation of events is controlled by returning some specific flags such as State.CONSUMED (which stops the propagation).

The ThumbnailWidget

Now it’s time to take a look at the widget’s code. While the Visual Library already provides an ImageWidget class which renders a generic Image, we need something more complex, for the following reasons:

- First and most important from a performance perspective, reading an image from a file needs some time, and a Light Table can contain tens of images. For instance, twenty images requiring 50ms each would lead to a full second of loading time. We can’t spend all this inside the event thread, or the Light Table would be sluggish. blueMarine deals with this by means of a ThumbnailRenderer class that manages image loading on demand and renders placeholders while images are not ready.

- Also, blueMarine will soon support image manipulation, and I’ll need to update all representations in real time when such changes happen (e.g. by painting specific decorations when a thumbnail is not up to date). This is solved using the Node class’s capability of firing events that notify updates – and then ThumbnailRenderer will do all the required work.

predefined actions suffice. For resizing support though, the ActionFactory.createResizeAction() method won’t do: it lets you arbitrarily change the widget’s dimensions, but the photos need to have a fixed aspect ratio. In such cases you can customize widget actions with special strategies.

See an example of a strategy in Listing 4. The widget’s boundSuggested() method is called by the Visual Library while we are dragging the widget; it’s passed both the original and current bounds. By returning a freshly computed Rectangle we can override the default settings. The code first gets the image’s aspect ratio and then computes the height from the new width or vice versa. This calculation takes into account the widget’s borders: if we draw a fixed-size border around the photo its thickness must not affect the aspect ratio calculation.

In blueMarine, preview images are wrapped by a Thumbnail class. Similarly, in your applications you’ll usually have a specific class containing the data you want to render in the widget. The problem is how to bind a widget to a data model. While the simplest solution appears to be to create specific getter/setter methods in ThumbnailWidget, this would introduce specific dependencies and require

In Listing 5. Customized action for bringing a widget to the front with a mouse click

private static final WidgetAction.Adapter bringToFrontAction = new WidgetAction.Adapter()
{
    @Override
    public State mouseClicked (final Widget widget, final WidgetMouseEvent event)
    {
        if (event.getButton() == MouseEvent.BUTTON1) {
            widget.bringToFront();
            return State.CONSUMED;
        }
        return State.REJECTED;
    }
};
```
public class ThumbnailWidget extends Widget {
    private final Node node;
    private final Thumbnail thumbnail;
    private ThumbnailRenderer thumbnailRenderer = DEFAULT_THUMBNNAIL_RENDERER;
    private final Lookup lookup;

    private final PopupMenuProvider popupMenuProvider = new PopupMenuProvider() {
        public JPopupMenu getPopupMenu (final Widget widget, final Point location) {
            return node.getContextMenu();
        }
    };

    public ThumbnailWidget (final Scene scene, final Node node, Dimension size) {
        super(scene);
        this.node = node;
        thumbnail = thumbnailManager.findThumbnail(node.getLookup().lookup((DataObject.class)));
        size = new Dimension(size);
        lookup = new ProxyLookup(node.getLookup(), Lookups.fixed(node, thumbnail));
        final BufferedImage image = thumbnail.getImage();
        if (image != null) {
            final int width = image.getWidth();
            final int height = image.getHeight();
            final double hScale = size.getWidth() / (float)width;
            final double vScale = size.getHeight() / (float)height;
            final double scale = Math.min(hScale, vScale);
            size.setSize(Math.round(scale * width), Math.round(scale * height));
        }
        setUnselectedBorder(DEFAULT_UNSELECTED_BORDER);
        setSelectedBorder(DEFAULT_SELECTED_BORDER);
        setBorder(unselectedBorder);
        final Insets insets = getBorder().getInsets();
        size.width += insets.left + insets.right;
        size.height += insets.bottom + insets.top;
        setPreferredSize(size);
        getActions().addAction(ActionFactory.createPopupMenuAction(popupMenuProvider));
    }

    @Override
    public Lookup getLookup() {
        return lookup;
    }
}
```

• Lastly, it's necessary to implement a context menu for the widgets, which must be coherent with the rest of the application. **Nodes** again provide support for this. (See the result in **Figure 3**).

Considering all this, it seems obvious that we need to implement a special widget class that delegates the implementation of context menus to the **Node** class and the rendering operations to **ThumbnailRenderer**. Let's first concentrate on the widget's creation.

In the code shown in **Listing 6**, I set some fields to refer to the **Node** and **Thumbnail**; then I adjust the user-specified size to comply with the photo's aspect ratio. I've previously mentioned the role of the **Lookup** class in linking a widget to its model, and shown how to extract the model from a properly prepared **Lookup** instance. Now, in **Listing 6**, you can see how the **Lookup** instance is prepared. A **ProxyLookup** is a NetBeans Platform class that “merges” two existing instances of **Lookup** – the one coming from the **Node** (required for the context menus to work) and a new one that contains both the **Node** and the **Thumbnail**.

To paint a custom widget, we add code to the **paintWidget()** method (see **Listing 7**). The obvious part here is that the image rendering is delegated to my **thumbnailRenderer.paint()** method. Less trivial is managing the scaling (remember, a scene can be zoomed in and out).
This is done by controlling the scale of Graphics2D.

Listening for node changes

Changes in the representation of photos are handled by firing events on the relevant Node. So that our ThumbnailWidget updates correctly, we need to setup a NodeListener, which you can attach and detach in the notifyAdded() and notifyRemoved() methods (see Listing 8). These are called when the widget is added to or removed from a scene (you can think of them as a kind of life-cycle control).

Now take a look at the doRepaint() method in Listing 9. Notice that it must cope with the usual Swing threading issues, since Node events can be fired by an arbitrary thread. Also, after a widget has been changed (in this case by calling its repaint() method), the scene must be validated. Otherwise you won’t see any updates.

Handling borders

The cream on the cake is adding visual cues to the widgets. We want to render different borders around the photos according to their selection state: no border for unselected widgets, a white border if selected, and a special “resize border” when you hover over the photo (see Figure 4).

First, we need to override the notifyStateChanged() method, which is called whenever the widget changes state (see Listing 10). It receives two parameters representing the old and the new state. We use the isSelected() and isHovered() methods to choose the proper border.

There’s a subtle problem here: borders can vary in thickness. By default the Visual Library preserves the overall size of a widget,

```java
Listing 7. Rendering the custom widget

@Override
protected void paintWidget()
{
    final Graphics2D g = getGraphics;
    final AffineTransform transformSave = g.getTransform();
    final Rectangle bounds = getClientArea();
    thumbnailRenderer.setThumbnail(thumbnail);
    g.translate(bounds.x + 1, bounds.y + 1); 
    bounds.width -= 1;
    bounds.height -= 1;
    thumbnailRenderer.setBounds(bounds);
    final double zoomFactor = getScene().getZoomFactor(); 
    g.scale(1 / zoomFactor, 1 / zoomFactor);
    thumbnailRenderer.paint(g);
    g.setTransform(transformSave); 
}

Listing 8. Listening for changes

private final NodeListener iconChangeListerner =
    new NodeAdapter()
    {
        @Override
        public void propertyChange(final PropertyChangeEvent event) {
            if (Node.PROP_ICON.equals(event.getPropertyName()))
            {
                doRepaint();
            }
        }
    };

    @Override
    protected void notifyAdded()
    {
        super.notifyAdded(); 
        node.addNodeListener(iconChangeListerner);
    }

    @Override
    protected void notifyRemoved()
    {
        super.notifyRemoved();
        node.removeNodeListener(iconChangeListerner);
    }
```

Figure 3.
Context menu for a widget
so setting a different border thickness would change the space reserved for the photo. To preserve the size of the photos, we just need to compute the change in the border and adjust the widget size (also in Listing 10).

Some final words about borders. The Border class for Widget is different from the usual Swing Border classes (see Listing 11). Widget borders are more complex. Also, there’s a similar BorderFactory which provides some preset borders useful in most cases. In the Light Table, the default borders are just rectangles with rounded corners that can be created with BorderFactory.createRoundedBorder(). You can create several common borders similarly: for instance, BorderFactory.createResizeBorder() gives you a standard “resize border” that is painted as a dashed line with “control handles”. In some special cases, we can write code to define customized borders. For instance, Listing 12 shows how to implement a “compound border” which sticks two borders together.

### Conclusions

In this article, we’ve seen many features of the Visual Library and a “creative use” for it that goes a little outside its most common scope. This provides us some examples for understanding how the library can be extended to comply with your needs. We’ve only scratched the surface, however. For instance, we didn’t explore connection widgets, which are another powerful feature. But that would be material for another article!
import org.netbeans.api.visual.border.Border;

public class CompoundBorder implements Border {
    private final Border border1;
    private final Border border2;

    public CompoundBorder (final Border border1, final Border border2) {
        this.border1 = border1;
        this.border2 = border2;
    }

    public Insets getInsets() {
        final Insets i1 = border1.getInsets();
        final Insets i2 = border2.getInsets();
        return new Insets(Math.max(i1.top, i2.top), Math.max(i1.left, i2.left),
                           Math.max(i1.bottom, i2.bottom), Math.max(i1.right, i2.right));
    }

    public void paint (final Graphics2D g, final Rectangle bounds) {
        // You should actually check if the insets are different...
        border1.paint(g, bounds);
        border2.paint(g, bounds);
    }

    public boolean isOpaque() {
        return border1.isOpaque() || border2.isOpaque();
    }
}

Listing 12. Creating a custom, compound Border

Fabrizio Giudici (fabrizio.giudici@tidalwave.it) has a Ph.D. in Computer Engineering from the University of Genoa (1998), and begun his career as a freelance technical writer and speaker. He started up a consultancy company with two friends, and since 2005 is running his own company. Fabrizio has been architect, designer and developer in many industrial projects, including a Jini-based real-time telemetry system for Formula One racing cars. He’s a member of the NetBeans Dream Team, the IEEE and of JUG Milano.

Figure 4. Visual cues for photos in the Light Table (normal, selected and resizing)