Submission Title
Graphics: Paintable & MutatablePaintable

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
One of the fundamental patterns in Design Patterns by Gamma, Helm, Johnson, and Vlissides is the Composite pattern. This pattern lets clients treat individual objects and compositions of objects uniformly. The motivating example for this pattern in the Design Patterns book is the case of a graphic entity that can either be a particular graphic or a composite graphic that renders itself by rendering each of its constituents in turn.

Unfortunately, Java does not directly support the Composite pattern for graphic entities. This submission concerns a family of classes that overcomes this limitation.

Solution Overview
Let us begin with an overview of the key interfaces and classes.

- **Paintable**. This interface is fundamental to the support for the Composite pattern. The most important method required by this interface is:
  ```java
  public void paint(Graphics g)
  ```
  This interface also requires methods to find the bounds and center of the painted region, to determine if a point falls within the painted region, and to control opacity and visibility.

- **MutatablePaintable**. This interface extends Paintable and requires that the object maintain an invertible affine transform called the mutator that will be applied during the rendering process to transform what is rendered. This interface also requires helper methods to set or modify the mutator and to perform simple movement directly.

- **MutatableWrapper**. The class wraps a Paintable to make a MutatablePaintable object. The implication of this class is that one can focus on Paintable and get MutatablePaintable for free.

- **PaintableSequence**. The class implements Composite for Paintable. In addition, as each Paintable is added, it is converted if necessary to MutatablePaintable by using a MutatableWrapper. Thus a PaintableSequence is MutatablePaintable not just Paintable. It is possible to mutate a sequence as a whole or to mutate its individual objects separately.

With the above infrastructure, we have the tools for building Paintables of arbitrary complexity but we need some initial structures to seed the process. This is provided by the following three classes.

- **ShapePaintable**. The class builds a Paintable using an object that satisfies the Java Shape interface together with a PaintMode that determines whether to fill or draw or both. Optionally, one can also set paints and strokes.

- **ImagePaintable**. The class builds a Paintable using an image specified by an Image object, an image specified by an ImageIcon object, an image specified by a filename, or an image specified by a URL. Optionally, one can set the top-left corner of the rendered image.

- **TextPaintable**. The class builds a Paintable using a String for the text, a Font & Paint to fill the text, and information on bounds and location. In particular, one can anchor text in any of 12 positions.

One other class is worth mentioning up front.

- **ClippingWrapper**. The class wraps a Paintable and a clipping Shape to obtain a Paintable that clips the original Paintable to the Shape.

Now let us show some examples.

**Example 1: Two Figure Eights**

This example was one of the earliest tests.

![Two Figure Eights](image.png)

Its code uses some tools from the submission on Shape Generation and also the window object in the Java Power Framework:

```java
float[][] figure8data = new float[][] {
{ 100, 150 }, { 300, 250 },
{ 300, 150 }, { 100, 250 });

AutomaticShape figure8 = new AutomaticShape
(figure8data, null, null, Path.BEZIER_CUBIC);

ShapePaintable paintable1 = new ShapePaintable
(figure8, PaintMode.FILL_DRAW, Color.red);
ShapePaintable paintable2 = new ShapePaintable
(figure8, PaintMode.FILL_DRAW, Color.blue);

MutatableWrapper wrapper2 = new MutatableWrapper
(paintable2, TransformFactory.rotate(200, 200, 90));
PaintableSequence sequence = new PaintableSequence
(new Paintable[] { paintable1, wrapper2 });

window.clearPanel();
sequence.paint(window.getBufferGraphics());
window.repaint();
```

Notice that we use the same shape figure8 to make two paintables. One is red and one is blue. The blue paintable is placed in a mutable wrapper and rotated by 90 degrees. The red paintable and the rotated blue paintable are placed in a paintable sequence which is then rendered onto the graphics window.

**Example 2: Shapes, Images, and Text in a Panel**

The example paints a square with all defaults (therefore black), an image of roses, the text Hello World rotated -90 degrees, and a circle in orange with a thick black boundary. The 4 objects are placed in a TablePanel which is shown in a frame.
The code uses some tools from the submission on GUI Composition. Note that in this case the layout manager for the panel determines the final pixel positioning of the objects.

```java
Object a = new Rectangle2D.Double(0, 0, 100, 100);
Object b = new ImagePaintable("./Images/Roses.jpg");
Font font = new Font("serif", Font.BOLD, 48);
TextPaintable text = new TextPaintable
("Hello World", font, Color.blue, null, TextAnchor.CENTER_BASELINE, 0, 0);
Object c = new MutatableWrapper
(text, TransformFactory.rotate(0, 0, -90));
Shape circle = new Ellipse2D.Double(0, 0, 100, 100);
Object d = new ShapePaintable
(circle, PaintMode.FILL_DRAW, Color.orange, Color.black, new BasicStroke(5));
TablePanel panel = new TablePanel(
    new Object[][]{
        {a, b},
        {c, d}
    }, 5, 5, CENTER);
JPTFrame.createQuickJPTFrame("Table Panel Test", panel);
```

Example 4: Kaleidoscope

The Kaleidoscope case study (which is available on the web site) was the motivating example for the development of the tools in this submission. The graphics window in the Kaleidoscope frame can display a sequence of shapes or images or both. The shapes can be rectangles, ellipses, polygons, or smooth cubic curves as constructed by the Shape Generation tools. The images in this case came from some digital photos of plants and vegetables but may be taken from any image files placed in the Images directory. As you can see, some images are clipped to a circular region using a clipping wrapper. Many of the shapes are rendered with some transparency to imitate colored glass. Transparency would also be possible with images but we decided not to do this after some experiments.

When the Kaleidoscope animation is running, a mirror wedge is formed at an angle 180/N for some small integer N. The graphics contents within the wedge region is reflected into 2N cells as in a physical kaleidoscope. The mirror wedge rotates by 1 degree in each animation cycle. The user can choose to have the individual objects in the window be modified by a random transform that is either restricted to being rigid (translation or rotation) or permitted to distort objects (scale or some more general affine transform). This facility is based on the fact that a mutable paintable can be mutated using a strategy object that adapts to the individual entity being mutated. Thus, for example, each object in the paintable sequence can be subject to its own particular random rotation during each animation cycle.

Experience with the Solution

The paintable and mutable paintable tools have only recently been added to JPT so we do not have extensive classroom testing. However, the Kaleidoscope case study and the many tests that support it have proven to us the effectiveness of the concepts and the implementation.

We believe that the paintable and mutable paintable tools in combination with the shape generation tools will provide students with a rich domain for computational exploration. Students will be able to design, render, and animate graphics for many different purposes. Furthermore, when they are ready to look at the JPT source code, they will be rewarded with excellent examples of object-oriented design.

API Documentation & Related Materials

The main JPT site to access documentation, code, and the jpt.jar:
http://www.ccs.neu.edu/jpt/