Zoom Panel 1.0 Component Specification

1. Design

The Zoom Panel component provides a Java Swing panel that performs zoom (and other kind of transformations) for another JComponent. It will transform the graphics of the original component to a certain zoom factor and will receive the mouse events and delegate to it according to the zoom transformation. This panel will also provide scrollbars, so that the zoomed component can be scrolled nicely.

This design separates the functionalities of Zoom Panel into two parts, the zoom transform, and the decorative controls. The zoom effect of the JComponent is achieved by the ZoomPane class, and the decorations are provided by the ZoomPanel class. The details of the zoom transform are encapsulated by ZoomTransform interface, so the transformation can be changed during runtime and other kinds of transformations are easy to plug-in in the future.

The zoom transform is applied upon the both the graphics and mouse events for the zoomed component, such that the zoomed component need not know it has been zoomed. That is to say, the zoomed Component will not need modification to perform the zooming, it can paint itself as usual and the listeners of it can handle the events as usual, this class will take care of the transform and convert the position of events for it. Noticeably, the ZoomPane can work on two modes, zoomed and not zoomed. If the enclosed component is zoomed, it will handle the transform as mentioned above; if the enclosed component is not zoomed the ZoomPane will just be a dummy container around the enclosed component, the all transformations will be disabled to avoid possible slowdown due to useless calculation.

The ZoomPanel utilizes Swing JScrollPane to provide decorations such as scroll bars and control over the viewport. Therefore, the implementation would be straightforward, and much other additional functionality, such as mouse wheel support, row/column header components, is brought into this component with the reuse.

1.1 Design Patterns

Decorator

Many parts of this design follow the reuse by inheritance idiom of Java GUI design, both ZoomPane and ZoomPanel classes are decorators over their super class. ZoomPane provides the additional responsibilities of zoom transform whereas ZoomPanel provides the scroll bars and view port control. Moreover, the inner class ZoomRepaintManager is also a decorator over the default system RepaintManager.

Strategy

The ZoomTransform interface encapsulates the zoom algorithm and the classes from the transform subpackage are interchangeable implementations.

Observer

Java AWT/Swing event model implements the observer pattern to notify the listeners, this design contains an inner class, MouseEventProxy, which is an implementation of the listener of the pattern to transform the event.

1.2 Industry Standards

None

1.3 Required Algorithms

Knowledge of the painting process of AWT and Swing is necessary to understand and implement this design. The following sub section will introduce the painting system briefly then specify the algorithms.

1.3.1 Painting Guideline

AWT/Swing uses callback API for painting components. The paint is usually done in paint method for AWT component and the paintComponent method for Swing component. The system will differentiate lightweight and heavyweight component when performing the painting, and there are some slight differences between AWT and Swing components, as this component is all about JComponent of Swing, which is lightweight component, we focus on the lightweight component paint in Swing.

There are two kinds of painting operations: system-triggered painting, and application-triggered painting.

System-triggered Painting

In a system-triggered painting operation, the system requests a component to render its contents, usually for one of the following reasons:

- The component is first made visible on the screen.
- The component is resized.
- The component has damage that needs to be repaired. (For example, something that previously obscured the component has moved, and a previously obscured portion of the component has become exposed).

App-triggered Painting

In an application-triggered painting operation, the component decides it needs to update its contents because its internal state has changed.

Whatever kind of painting is requested, this component must guarantee the zoom transform is performed before the zoomed component is get paint. We must insert the transformation code in the paint procedure. For Swing components, paint() is always invoked as a result of both system-triggered and app-triggered paint requests. And Swing's implementation of paint() factors the call into 3 separate callbacks:

- paintComponent() – this paint the core content of the component
- paintBorder() – this paint the border
- paintChildren() – this asks the children of the component to paint.

We will explain later that we can add the zoomed component as the child of the ZoomPane, so we must paint the zoomed component as the ZoomPane's core content, this is done is paintComponent method. Later section will describe the transform and paint details.

The app-triggered painting is a bit complicated. The zoomed component will receive the request, however, the default repaint process will not know that zoom transformation is needed before the painting, since it is not required to know it has been zoomed. The ZoomPane must intercept the request and perform the transformation. The default implementation of repaint will put the request as asynchronous request via RepaintManager, so we can replace the RepaintManager implementation to let the ZoomPane know when zoomed component is receiving such request. The details are in the later section.

As mentioned above, we can add the zoomed component as child of the ZoomPane if we want to perform zoom transformation. This is because if adding the zoomed component as the child, the system will "think" that the zoomed component is placed above the ZoomPane, then the zoomed component will gain mouse input and receive the mouse events. As the zoomed component has been zoomed and the event position is in the screen coordinates, the event listener of zoomed component will not have valid behavior upon the events. Thus, we have to intercept the mouse event as well, and apply inverse transform on the position then delegate to the listeners. The details are also in later section.

1.3.2 Zoom transform and component painting

This is done in paintComponent of ZoomPane.

```
1. invoke super version
2. if the transform is no zoom, then return
3. apply the transform
      Graphics2D g2 = (Graphics2D)g.create();
      transform.applyTransform(g2);
4. paint the zoomed component
      zoomComponent.paint(g2);
5. release the resource if any
      g2.dispose
```

1.3.3 Replace the default RepaintManager

The repaint request is given to the method addDirtyRegion of RepaintManager, in this method, we do the following in our custom implementation:

```
      1. delegate to original RepaintManager.
      2. if c is zoomComponent, then make the region of the enclosing
component as dirty as well
           if (zoomComponent.equals(c)) {
               rm.markCompletelyDirty(ZoomPane.this);
           }
```

Optimized painting, such as making use of the clip region, is welcomed when implementing this algorithm. Other methods of the implementation will simply delegate the original RepaintManager.

We replace the RepaintManager in the constructor of ZoomPane class:

```
  RepaintManager old = RepaintManager.currentManager(zoomComponent)
  RepaintManager rm = new ZoomRepaintManager(old);
  RepaintManager.setCurrentManager(rm);
```

1.3.4 Mouse Event Delegation

The design contains an implementation of the MouseInputListener to transform and delegate event, this implementation is added to ZoomPane as both MouseListener and MouseMotionListener if we want to intercept the mouse event. All the methods are implemented in the same manner:

```
     1. convert the MouseEvent
          MouseEvent converted = transform.applyInverseTransform(e);
     2. get the listeners of the enclosed component, the listeners'
type (MouseListerners v.s. MouseMotionListeners) is dependent on the
operation performed (e.g. MouseListeners for mouseClicked and
MouseMotionListeners for mouseMoved);
     3. invoke the name-like method of each listener retrieved in 2.
```

1.3.5 Disable Transformation when No Zoom

The zoom transformation is disabled if the zoom factor is 1.0 to avoid possible slowdown due to useless calculations. So, there will be two conceptual modes of the ZoomPane class. When working on no zoom, we can add the zoomed component as its child, these will let the zoomed component painted as normal, and as it can receive and handle the mouse event correctly, we should remove the mouse delegate. We will switch the mode if the transform is changed during runtime.

```
     a) from no zoom to zoom
          remove the zoomComponent from the containment
              remove(zoomComponent);
```

```
           register the mouseProxy instance as the MouseListener and
MouseMotionListener of this zoom pane
               addMouseListener(mouseProxy);
               addMouseMotionListener(mouseProxy);
              addFocusListener(focusProxy);
               addKeyListener(keyProxy);
      b) from zoom to no zoom
           add the zoomComponent from the containment
               add(zoomComponent);
           remove the mouseProxy instance as the MouseListener and
MouseMotionListener of this zoom pane
               removeMouseListener(mouseProxy);
               removeMouseMotionListener(mouseProxy);
             removeFocusListener(focusProxy);
               removeKeyListener(keyProxy);
```

1.4 Component Class Overview

ZoomPanel:
ZoomPanel adds scroll bars, viewport control, and other facilities to the ZoomPane, so they can work as a whole as the familiar zoom window. This implementation use Swing JScrollPane as the base to support the scroll bars and viewport.

ZoomPane:
ZoomPane is a concrete Swing JComponent that can perform zoom and other kind of transformations for another JComponent. The transformation (normally zoom) will be applied upon the enclosed JComponent when painting(rendering) the ZoomPane, and the enclosed JComponent will receive mouse and key events (including motion events) as normal JComponent that can gain user input. The enclosed JComponent will not be noticed that it has been zoomed, that is to say, the zoomed Component will not need modification to perform the zooming, it can paint it as usual and the listeners of it can handle the events as usual, this class will take care of the transform and convert the position of events for it. It also contains convenient API to access the zooming functionality.

ZoomTransform(interface)
ZoomTransform is the transform need to be applied onto coordinate system to achieve zooming effect. The ZoomPane class will use this interface to zoom in and/or out the enclosed JComponent.

NatureZoomTransform
A nature implementation of the ZoomTransform interface. Both components (x, y) of in the zoomed coordinates system will simply be the x, y coordinates multiplied by the zoom factor.

AffineZoomTransform
Implementation of ZoomTransform interface by affine transform. It utilizes the Java system AffineTransform class to perform the zoom transform.

ZoomRepaintManager(inner class)
A custom implementation of ZoomRepaintManager. This class is necessary for the enclosing component to receive notification and repaint itself as well when an application repaint is generated to the enclosed component.

MouseEventProxy(inner class)

The implementation of MouseInputListener to convert the Mouse Events occurred in this Zoom Pane and delegate to the enclosed zoom component.

FocusEventProxy (inner class)
Implementation of KeyListener interface to delegate the KeyEvent to zoomed component. All the focus listeners of the zoomed component will receive the event directly.

KeyEventProxy(inner class)
Implementation of KeyListener interface to delegate the KeyEvent to zoomed component.

1.5 Component Exception Definitions

IllegalArgumentExceptoin
This design uses IllegalArgumentException to indicate invalid argument is passed to the provided API, generally null and nonpositive zoom factor.

1.6 Thread Safety

This design is not thread safe, since most of the classes have states and are mutable. This will not be a problem in usual GUI application. In a typical Swing application, there will only be a thread access the component's state. This is also known as "The Single-Thread Rule", "Once a Swing component has been realized, all code that might affect or depend on the state of that component should be executed in the event-dispatching thread." It is required for the user to obey this rule to use this component in a Swing application, otherwise, not only this component but also the Java Swing system will have unexpected behavior.

However, if this component is used in a multi-threaded environment, e.g., there are several worker threads to complete some kind of task; it will not cause any problems as long as the threads have no access to the state of this component, or all the request is posted via the event-dispatcher. Please refer to the javadoc of SwingUtilities class for how to do this.

If we real want to achieve thread safety of this component, despite this strongly opposed, we only need to make synchronization around the API that has access to the states to achieve thread safety, like all methods of ZoomTransform instance.

2. Environment Requirements

2.1 Environment

- Development language: Java 1.5
- Compile target: Java 1.5

2.2 TopCoder Software Components

None.

2.3 Third Party Components

None.

3. Installation and Configuration

3.1 Package Name

com.topcoder.gui.panels.zoom

3.2 Configuration Parameters

Not required. The configuration of scrollbars, viewport and transform is provided via API.

3.3 Dependencies Configuration

None.

4. Usage Notes

Please follow the Swing guild lines to use this component

4.1 Required steps to test the component

- Extract the component distribution.
- Follow Dependencies Configuration.
- Execute ‘ant test’ within the directory that the distribution was extracted to.

4.2 Required steps to use the component

This component can be working alone, it is just Swing JComponent implementation, and the application should use it in a Swing GUI application.

Please see the demo section for the general steps.

4.3 Demo

This demo use a custom JComponent as the zoomed component, it is from the Sun Tutorial http://java.sun.com/docs/books/tutorial/uiswing/painting/concepts2.html.

This will paint 20 x 20 grids, and monitor mouse click and mouse motion.

```
public class CoordinateArea extends JComponent
        implements MouseInputListener {
    // last clicked point
    Point point = null;
    Demo controller;
    Dimension preferredSize = new Dimension(400,75);
    Color gridColor;

    public CoordinateArea(Demo controller) {
        this.controller = controller;

        //Add a border of 5 pixels at the left and bottom,
        //and 1 pixel at the top and right.
        setBorder(BorderFactory.createMatteBorder(1,5,5,1,
                                                  Color.RED));

        addMouseListener(this);
        addMouseMotionListener(this);
        setBackground(Color.WHITE);
        setOpaque(true);
    }

    public Dimension getPreferredSize() {
        return preferredSize;
    }

    protected void paintComponent(Graphics g) {
        //Paint background if we're opaque.
        if (isOpaque()) {
            g.setColor(getBackground());
            g.fillRect(0, 0, getWidth(), getHeight());
        }

        //Paint 20x20 grid.
        g.setColor(Color.GRAY);
        drawGrid(g, 20);
        //If user has chosen a point, paint a small dot on top.
        if (point != null) {
            g.setColor(getForeground());
            g.fillRect(point.x - 3, point.y - 3, 7, 7);
        }
    }

    //Draws a 20x20 grid using the current color.
    private void drawGrid(Graphics g, int gridSpace) {
        Insets insets = getInsets();
        int firstX = insets.left;
        int firstY = insets.top;
        int lastX = getWidth() - insets.right;
        int lastY = getHeight() - insets.bottom;

        //Draw vertical lines.
        int x = firstX;
        while (x < lastX) {
            g.drawLine(x, firstY, x, lastY);
            x += gridSpace;
        }

        //Draw horizontal lines.
        int y = firstY;
        while (y < lastY) {
            g.drawLine(firstX, y, lastX, y);
            y += gridSpace;
        }
    }

    //Methods required by the MouseInputListener interface.
    public void mouseClicked(MouseEvent e) {
        System.out.println(1);
        int x = e.getX();
        int y = e.getY();
        if (point == null) {
            point = new Point(x, y);
        } else {
            point.x = x;
            point.y = y;
        }
        controller.updateClickPoint(point);
        repaint();
    }

    public void mouseMoved(MouseEvent e) {
        controller.updateCursorLocation(e.getX(), e.getY());
    }

    public void mouseExited(MouseEvent e) {
        controller.resetLabel();
    }

    // We are not interested of the following events.
    public void mouseReleased(MouseEvent e) { }
    public void mouseEntered(MouseEvent e) { }
    public void mousePressed(MouseEvent e) { }
    public void mouseDragged(MouseEvent e) { }
}
```

The main method to create the UI:

```
    // Create and set up the window.
    // The JFrame instance.
    JFrame frame = new JFrame("CoordinatesDemo");
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

    // We add the zoom component to the root pane
    Container container = frame.getRootPane();
    container.setLayout(new BoxLayout(container,
                                  BoxLayout.PAGE_AXIS));

    // coordinateArea is the zoomed component
    CoordinateArea coordinateArea = new CoordinateArea(this);
    // create a ZoomPanel to wrap around it.
    ZoomPanel zoomPanel = new ZoomPanel(coordinateArea);
    container.add(zoomPanel);

    //Display the window.
    frame.pack();
    frame.setVisible(true);
```

Access the zoom factor during runtime

```
    // get the old factor
    double factor = zoomPanel.getZoomFactor();
    // zoom to 3.0
    zoomPanel.setZoomFactor(3.0);
```

Access/Change the zoom transform

```
    // get the old factor
    ZoomTransform tranform = zoomPanel.getTranform();
    // zoom to 2.5
    zoomPanel.setTranform(new AffineZoomTransform(
        AffineTransform.getScaleInstance(2.5, 2.5)));
```

Manipulate Scrollbars:

```
    // use the API of JScrollPane
    zoomPanel.setHorizontalScrollBarPolicy(
        ScrollPaneConstants.HORIZONTAL_SCROLLBAR_AS_NEEDED);
    zoomPanel.setVerticalScrollBarPolicy(
        ScrollPaneConstants.VERTICAL_SCROLLBAR_ALWAYS);
    // Access the JScrollBar indirectly
    zoomPanel.getVerticalScrollBar().setUnitIncrement(3);
    zoomPanel.getVerticalScrollBar().setBlockIncrement(25);
```

Manipulate the viewport

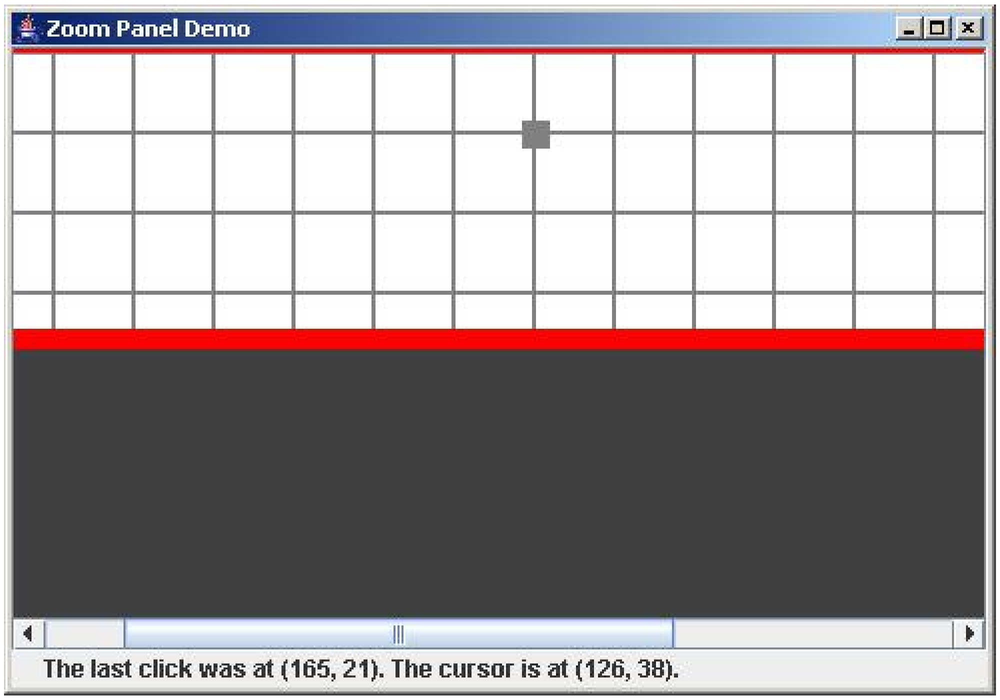
```
    // get access to the JViewport
    JViewport viewport = zoomPanel.getViewport();
    // manipulate via JViewport's API
        viewport.scrollRectToVisible(toShow);
        viewport.setViewSize(size);
        viewport.setScrollMode(JViewport.BACKINGSTORE_SCROLL_MODE);
```

Configure the background

```
    zoomPanel.setZoomBackground(Color.BLUE);
```

Screenshot

The following is a possible screenshot of the demo. The zoom factor is 2.0 and the background is set to dark gray.



The last click was at (165, 21). The cursor is at (126, 38).

5. Future Enhancements

More sophisticated transformations. Such as the rotation transform can be easily plugged in.

Feasible layout management. It is achievable via providing a custom LayoutManager implementation and using it to manage the layout of the ZoomPanel

Configuration management. Read the configuration parameter from anywhere and use them to specify the UI. It's best to have a universal configuration manager for the whole UML project.