Swing Birds Eye View Control 1.0 Component Specification

1. **Design**

The UML Tool displays diagrams used for software modeling. These diagrams tend to get very large, and usually the diagrams are too large to be displayed in their entirety on a single display. To address this issue, this component will display a thumbnail view of a diagram, allowing the user to drag a view port to various areas on the diagram, allowing for easier manipulation of the displayed areas.

Design Considerations

This design is very simple, but supports thumbnail view for any JComponent. The BirdsEyeView is a main class, which works as a parent container for this control. It is used as a public API of this component, and stores some objects, like ZoomPanel, ViewPort, input JComponent, etc.

The ViewPort class is a JComponent and provides a simple bordered rectangle with opacity, which will be drawn over the zoomed input object. The view port accepts mouse events and can be dragged by the user.

The external code (some kind of the input object manager) will be notified when view port is moved. The ViewPortListener interface defines a simple listener for that purpose, and the ViewPortEvent class provides the event data. The external code should implement that ViewPortListener and register by using ViewPort.addViewPortListener(…) method.

Please note, this component is automatically notified in any change in the input object, because the BirdsEyeView class will be registered as a child JComponent of the input object. Anyway, the user of our component has to provide the position and dimension of the input object view port. Without such information it is not possible to properly use the control. And we can not retrieve that view port information of the input object, because it can be provided in different ways (for example, with scrolling) and JComponent can be of any child type.

1. **Design Patterns**

Observer pattern is implemented by the ViewPort class, which allows registering ViewPortListener instances and will call them in the case when ViewPort has changed. And the similar **Listener pattern** is also used – the ViewPortListener defines a listener interface, ViewPortEvent is an event data, and the ViewPort is producer of events. The concrete listeners – will be in the external application.

Of course, we also re-use Listener pattern, defined in the AWT and Swing, because BirdsEyeView and ViewPort classes will be registered to the MouseListener and/or MouseMotionListener and we will be notified by those mouse events.
Composite pattern is re-used from JComponent. Each JComponent can have children of any JComponent and we use this technique to combine ZoomPanel, ViewPort and BirdsEyeView all together. All of them have the same method – paintComponent(…), which allows us the cascading update and drawing of those components.

2. Industry Standards

JFC Swing (from JSE 5).

3. Required Algorithms

Important Notes:

1. Please note, all algorithms below do not use precise rounding when converting double => int data types. The developer should check this issue in the actual control – how will it look. Most likely, all will be fine and this is the fastest way. If not, then please apply the precise rounding: Math.round(…) method can be used.
2. And the special checks are needed when deleting by 0 – it should not occur in most likely cases, but anyway should be checked and skipped (like if zoom factor == 0.0, then use zoom factor = 1.0).
3. The getSize(null) was used in algorithms below to make them fully clear. The developer could change that to getWidth() and getHeight() to slightly improve performance. And getLocation(null) was also used in algorithms below to make them fully clear. The developer could change that to getX() and getY() to slightly improve performance.

1. *Creating and Drawing the Birds Eye View*

Create the BirdsEyeView class

This algorithm relates to the BirdsEyeView(inputObject, inputObjectViewDimension, inputObjectViewPosition) constructor of the BirdsEyeView class. The next actions should be performed.

1. Set the internal fields:
   this.inputObject = inputObject;
   this.viewPort = new ViewPort();
   this.zoomPanel = new ZoomPanel(inputObject);
   this.inputObjectViewDimension = inputObjectViewDimension;
   this.inputObjectViewPosition = inputObjectViewPosition;
2. Retrieve the configuration parameters:
   Color overlayColor = UIManager.getColor(“BirdsEyeView.overlayColor”);
   Color overlayBorderColor=UIManager.getColor(“BirdsEyeView.overlayBorderColor”);

int overlayBorderThickness =
UIManager.getInt("BirdsEyeView.overlayBorderThickness");
double opacity = Double.parseDouble(
UIManager.getString("BirdsEyeView.overlayOpacity"));

3. Set the properties of the viewPort field:
   this.viewPort.setOpacity(opacity);
   this.viewPort.setBackground(overlayColor);
   this.viewPort.setForeground(overlayBorderColor);
4. Set the border:
   this.viewPort.setBorder(BorderFactory.createLineBorder(overlayBorderColor, overlayBorderThickness));
5. Register the mouse listener for the current class:
   this.addMouseListener(this);
6. Set the parent-child hierarchy of the used JComponent instances:
   this.inputObject.add(this);
   this.add(this.zoomPanel);
   this.zoomPanel.add(this.viewPort);
7. Draw the entire component (this.paintComponent(…) will be called):
   this.repaint();

Draw in the BirdsEyeView class

This algorithm relates to the paintComponent(g) method of the BirdsEyeView class. The next actions should be performed.

1. Calculate the zoom factor:
   Dimension thisSize = this.getSize(null);
   Dimension inputSize = this.getInputObject().getSize(null);
   double scaleWidth = thisSize.getWidth() / inputSize.getWidth();
   double scaleHeight = thisSize.getHeight() / inputSize.getHeight();
   double zoomFactor = the minimum value from scaleWidth and scaleHeight;
2. Set the zoomFactor to the zoomPanel:
   this.getZoomPanel().setZoomFactor(zoomFactor);
3. Calculate the size of the view port:
   Dimension viewPortSize = new Dimension(
   this.getInputObjectViewDimension().getWidth() * zoomFactor,
   this.getInputObjectViewDimension().getHeight() * zoomFactor);
4. Set the view port size:
   this.getViewPort().setSize(viewPortSize);
5. Calculate the location of the view port:
   Point viewPortLocation = new Point(this.getInputObjectViewPosition().getX() * zoomFactor, this.getInputObjectViewPosition().getY() * zoomFactor);
6. Set the location of the view port:
   this.getViewPort().setLocation(viewPortLocation);

7. Set the zoom factor to the viewPort:
   this.getViewPort().setZoomFactor(zoomFactor);
8. That's all – return here, and the children components will be automatically repainted (including zoomPanel and viewPort) – they will draw the actual image of the control.

2. *Creating and Drawing of the View Port*

Create the ViewPort class

This algorithm relates to the ViewPort( ) constructor of the ViewPort class. The next actions should be performed.

1. Register the mouse listener for the current class:
   this.addMouseListener(this);
   this.addMouseMotionListener(this);

Draw in the ViewPort class

This algorithm relates to the paintComponent(g) method of the ViewPort class. The next actions should be performed.

1. Draw the background:
   g.setColor(this.getBackground());
   g.fillRect(0, 0, this.getWidth(), this.getHeight());
2. Apply the opacity:
   Graphics2D g2 = (Graphics2D) g;
   g2.setComposite(AlphaComposite.getInstance(AlphaComposite.SRC_OVER, (float) this.getOpacity()));
3. Calculate the location of the input object view port:
   Point currentLocation = this.getLocation(null);
   int x = currentLocation.getX() / this.getZoomFactor();
   int y = currentLocation.getY() / this.getZoomFactor();
4. Make the event data for the input object view port:
   ViewPortEvent event = new ViewPortEvent(x, y);
5. Fire the listeners to update the current position of the input object view port (the manager of the input object is most likely the consumer of this event):
   this.fireViewPortListeners(event);

3. *Processing Events in the BirdsEyeView Class*

This algorithm relates to the mousePressed(event) method of the BirdsEyeView class. The next actions should be performed.

1. Get the mouse coordinates:
   int x = event.getClientX();
   int y = event.getClientY();
2. Please note, the zoomPanel will fully cover the BirdsEyeView control, and that zoomPanel has a special pre-processing of all the mouse events, which go through it.

That zoomPanel convert their coordinates to the inputObject coordinates, which are not applicable for us. So, the conversion back is needed:
x = x / this.getZoomPanel().getZoomFactor();
y = y / this.getZoomPanel().getZoomFactor();

3. Check if the x and y are inside the "this" control bounds (just to be sure) – and return if any of them is outside "this" control.
4. Retrieve the size of the view port:
   Dimension viewPortSize = this.getViewPort().getSize(null);
   int viewPortWidth = viewPortSize.getWidth();
   int viewPortHeight = viewPortSize.getHeight();
5. Check the limits of the view port borders:
    1. Check the left limit:
       if (x - (viewPortWidth/2)) < 0, set x = viewPortWidth/2;
    2. Check the right limit:
       if (x + (viewPortWidth/2)) >= this.getSize(null).getWidth(), set x = this.getSize(null).getWidth() - viewPortWidth/2 - 1;
    3. Check the top limit:
       if (y - (viewPortHeight/2)) < 0, set y = viewPortHeight/2;
    4. Check the bottom limit:
       if (y + (viewPortHeight/2)) >= this.getSize(null).getHeight(), set y = this.getSize(null).getHeight() - viewPortHeight/2 - 1;
6. Calculate the new location of the view port:
   Point newViewPortLocation = new Point(x - viewPortWidth/2, y - viewPortHeight/2);
7. Set the new position of the view port:
   this.getViewPort().setLocation(newViewPortLocation);
8. Repaint the entire component (please note, it will also fire all the view port listeners):
   this.repaint();

4. *Processing Events in the ViewPort Class*

Mouse Pressed Event Processing

This algorithm relates to mousePressed(event) algorithm of the ViewPort class. The next actions should be performed.

1. Start the dragging:
   this.setDragging(true);
2. Set the current dragging position:
   this.getDragPosition().setLocation(event.getClientX(), event.getClientY());
3. It is suggested to finish processing of this event here and do not propagate the event to the parent controls, which will center the view port of mouse press (the event.consume() method can help on this). But the developer can experiment with this feature – how will it look in the actual GUI? May be centering of the view port even then pressing on it will be fine.

Mouse Dragged Event Processing

This algorithm relates to mouseDragged(event) algorithm of the ViewPort class. The next actions should be performed.

1. If this.isDragging() == true, then do the next actions:
    1. Get the new dragging position:
       Point newDragPosition = new Point(event.getClientX(), event.getClientY());
    2. If this.getDragPosition().equals(newDragPosition) == true, then simply return – no moving of the view port is needed:
       return();
    3. Calculate the displacements:
       int xDelta =newDragPosition.getX() – this.getDragPosition().getX();
       int yDelta =newDragPosition.getY() – this.getDragPosition().getY();
    4. Retrieve the size of the view port:
       Dimension viewPortSize = this.getSize(null);
       int viewPortWidth = viewPortSize.getWidth();
       int viewPortHeight = viewPortSize.getHeight();
    5. Get the current location of the view port:
       Point viewPortLocation = this.getLocation(null);
    6. Calculate the possible new location of the view port:
       int x = viewPortLocation.getX() + xDelta;
       int y = viewPortLocation.getY() + yDelta;
    7. Get the size of the parent control:
       Dimension parentSize = this.getParent().getSize(null);
       int parentWidth = parentSize.getWidth();
       int parentHeight = parentSize.getHeight();
    8. Check the limits of the view port borders for the new location:
        1. Check the left limit (do not move the view port outside the parent control):
           if (x < 0) return;
        2. Check the right limit (do not move the view port outside the parent control):
           if ((x+viewPortWidth) > parentWidth) return;
        3. Check the top limit (do not move the view port outside the parent control):
           if (y < 0) return;
        4. Check the bottom limit (do not move the view port outside the parent control):
           if ((y+viewPortHeight) > parentHeight) return;
    9. All checks were passed, therefore – set the new location of the view port:
       this.setLocation(x, y);
    10. Set the new value for the drag position:
        this.getDragPosition().setLocation(newDragPosition);
    11. Repaint the view port:
        this.repaint();

Mouse Released Event Processing

This algorithm relates to mouseReleased(event) algorithm of the ViewPort class. The next actions should be performed.

1. Stop the dragging:
   this.setDragging(false);

4. **Component Class Overview**

1. *com.topcoder.swing.birdseyeview Package*

BirdsEyeView:

The main class of the component. This is a JComponent child. It is just a container for the ZoomPanel and there is some business logic: the input object (JComponent), view port, dimension and position of the input object view port are stored in this class. They are accessible through setters/getters. This class processes mousePressed event - will center this component view port at the mouse cursor.

Please note, the BirdsEyeView will be added as a child JComponent to the inputObject. So, the BirdsEyeView will be automatically updated when the parent InputObject changes. The paintComponent(...) method of this class will be called on any update of the input object. All calculations will be performed in that method.

Please note, setting the properties of this class do not perform repainting of the control, because the efficiency is very important and several properties will be usually set (not just once).

ViewPort:

The view port of the birds eye view. This class represents the JComponent (just a simple rectangle), which will be shown over the birds eye view zoom panel and define the current view port. This is a zoomed view port related to the view port of the input object. It supports dragging by mouse and mouse press event is also processed - will center this component view port at the mouse cursor. The view port has an opacity from 0.0 (0%) to 1.0 (100%). This class will be automatically updated on any change in the ZoomPanel or BirdsEyeView JComponent, because it is a child of them. The paintComponent(...) method of this class will be called on any updates. All calculations will be performed in that method. This class support registration of the view port listeners, which will be called in the case when the input object view port position should be changed.

Please note, setting the properties of this class do not perform repainting of the control, because the efficiency is very important and several properties will be usually set (not just once).

2. *com.topcoder.swing.birdseyeview.event Package*

ViewPortListener (interface):

The listener interface for view port actions. This interface defines a public contract for ViewPort listeners. It simply defines one method - viewPortMoved - to be consumed by external code. The manager of the input object should register for this viewPortMoved event to be notified when the view port has moved over the zoom panel.

ViewPortEvent:

An event which indicates that a view port action occurred in a component. This is a simple data container class for storing information about view port. It has information about new coordinates of the input object view port.

5. **Component Exception Definitions**

IllegalArgumentException:

This exception is for reporting wrong arguments data of method parameters. It can be thrown is argument is null, or has incorrect value.

No custom exceptions were defined, because this is a GUI control component and it is very simple.

6. **Thread Safety**

This component is not thread-safe, because most of the classes in this component are mutable except the event class. That event class is thread-safe because it is immutable. Thread-safety is not required. Like many other standard Swing components and controls, thread-safety should be cared by users. And there is one issue we want to discuss further. Event listeners list is used by both the main application thread (adding or removing listeners), and the event dispatching thread (using the listeners). This problem can be solved by listenerList field in JComponent, which is thread-safe. We add the listeners to that list, so thread safety is not a problem here.

2. **Environment Requirements**

1. **Environment**

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

1. **TopCoder Software Components**

- **Zoom Panel 1.0** – 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

NOTE: The default location for TopCoder Software component jars is../lib/tcs/ COMPONENT_NAME/COMPONENT_VERSION relative to the component installation. Setting the tcs_libdir property in topcoder_global.properties will overwrite this default location.

1. **Third Party Components**

- None.

NOTE: The default location for 3rd party packages is ../lib relative to this component installation. Setting the ext_libdir property in topcoder_global.properties will overwrite this default location.

3. **Installation and Configuration**
    1. **Package Name**

com.topcoder.swing.birdseyeview – the package with the main classes of this component.
com.topcoder.swing.birdseyeview.event – contains listener interface and the event data class.

2. **Configuration Parameters**

All the configuration properties of this component will be provided through the UIManager. They are shown on the next table.

Parameter	Description	Values
BirdsEyeView.overlayColor	The fill color of the view port.	Color.***Optional*** It can be absent/null – means use the default color (gray).
BirdsEyeView.overlayBorderColor	The border color of the view port.	Color.***Optional*** It can be absent/null – means use the default color (black).

Parameter	Description	Values
BirdsEyeView.overlayBorderThickness	The thickness (in pixels) of the view port border.	int.***Optional*** It should be 0 or greater. It can be absent – means use the default value (1).
BirdsEyeView.overlayOpacity	The opacity value of the view port.	String.***Optional*** Actually, this is double value to be parsed from the string. It should be from 0.0 to 1.0. It can be absent/null/empty string – means use the default opacity (0.5).

The configuration options can be retrieved like this:
Color overlayColor = UIManager.getColor(“BirdsEyeView.overlayColor”);

3. **Dependencies Configuration**

None.

4. **Usage Notes**
    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.

2. **Required steps to use the component**

None special.

3. **Demo**

The graphical example of the birds eye view control is shown on the "docs\BirdsEyeMockup.png" file. It most likely to be used with the TC UML Tool. So, the user scenario is as follows.

First, let's create our birds eye view control:

```
// Assume we have the big class diagram, let's name it an input object
JComponent inputObject = some instance of the big control;

// The currently shown part of the input object has a size. We should
use
// that size with our birds eye view control
Dimension inputObjectViewDimension = new Dimension( 800, 600 );

// The user can scroll the input object (for example, the big class
// diagram). The current location of the left-top origin of the
currently
// shown part of the input object should be used with our birds eye
view
// control
Point inputObjectViewPosition = new Point( 300, 200 );

// Finally - create the birds eye view control
BirdsEyeView control = new BirdsEyeView( inputObject,
inputObjectViewDimension, inputObjectViewPosition );
// From now on, the birds view control is created, registered as a
child
// of the inputObject, registered to mouse events, and displayed.
// Any changes in the inputObject (like change the entire size of the
// inputObject, change of the graphical data - add/remove new class) -
// will be automatically displayed by the current control.
// The user can manually change the size of our control and modify any
// GUI related properties as for other JComponent controls
```

The user can interact with the inputObject in his/her application. The most changes are automatically processed by our birds view control, but there are other special cases, where the external code should manually notify our control:

- Change the size of the input object view port (like the user can re-size the window of the input object view port). So, we should adjust the size of our control view port in this case.
- Change the position of the input object view port (like the user can scroll the input object). So, we should adjust the location of our control view port in this case.

```
// Set the new dimension of the inputObject view port. The birds eye
view
// control will update its own view port size to have proper value
control.setInputObjectViewDimension( new Dimension(640, 480) );
// Or another way:
control.getInputObjectViewDimension().setSize( 500, 400 );


// Set the new position of the inputObejct view port. The birds eye
view
// control will update its own view port position to have proper value
control.setInputObjectViewPosition( new Point(100, 150) );
// Or another way
control.getInputObjectViewPosition().setLocation( 90, 80 );

// If those updates were performed in the paintComponent(...) method of
// the input object, then no other actions are needed – our birds eye
// view control will be updated automatically.
//
// In other case (if dimension and position were changed outside the
// paintComponent(...) method of the parent JComponent), then we should
// manually redraw our control like this:
control.repaint();
```

Please note, the user can manipulate with our birds eye view control by mouse. For example, the user can drag the birds eye view port rectangle. Or any click outside the birds view port rectangle will center the view port at the mouse cursor. Of course, the input object view port should reflect those changes. In the case of TC UML Tool, the currently shown part of the class diagram should be moved accordingly. The birds eye view provides a simple and flexible mechanism of notifications – when the input object view port position should be changed (like scrolled). First of all, the external application should provide the concrete listener:

```
// Make the listener class
public class ChangeViewPortListener implements ViewPortListener
{
// Some internal fields, constructor(s), initialization, etc...

// The event handler
public void viewPortMoved( ViewPortEvent event )
{
// Retrieve the new coordinates of the view port
```

```
int x = event.getInputObjectViewX();
int y = event.getInputObjectViewY();

// Modify the input object view port by new x and y values.
// Some scrolling actions can be performed here
}

}
```

Please note, that listener action can be implemented inside some other class (like the manager of the input object).
Second, the concrete listener should be registered in our birds eye view control:

```
// Make an instance of the listener
ViewPortListener listener = new ChangeViewPortListener( );

// Register the listener to the birds eye view control
control.getViewPort().addViewPortListener( listener );
```

Now, the listener object will automatically get all the notifications of the required position change for the input object. Please note, the event:ViewPortEvent object, provided to that listener, contains the proper new position for the input object view port. This is not the position of the birds eye view control view port, but the coordinates for the input object view port.
Of course, the listener can be de-registered like this:

```
// De-register the view port listener
control.getViewPort().removeViewPortListener( listener );

// Or all the ViewPortListener instances can be easily removed
control.getViewPort().clearViewPortListeners( );
```

Please note, the user has an ability to change the input object on-the-fly. For example, the user can switch from class diagram to the sequence diagram. The next actions are needed.

```
// Change to another input object
control.setInputObject(another instance of the JComponent input
object);
// The next actions were performed in the previous call:
// - the birds eye view was totally de-registered from the previous
// inputObject.
// - all the ViewPortListener instances were de-registered from the
birds
```

```
// eye view control view port.
// - the zoomPanel was updated with a new inputObject JComponent
instance
// - the new inputObject was set to the internals of the bird eye view
// control
// - the birds eye view control was registered as a child to the new
// inputObject

// The input object view port position is most likely should be new,
not
// from the previous inputObject (for example, the sequence diagram can
// be scrolled to the different position than class diagram).
// Set the new position of the input object view port
control.setInputObjectViewPosition( new Point(20, 40) );

// The size of the input object view port is most likely should NOT be
// new, but the same as from previous input object (for example, the
// shown part of the sequence diagram will be the same as of the
previous
// class diagram. So, no updating of the input object view part
dimension
// is needed.
//
// But in the general cases, the user can set the new dimensions of the
// input object view port like this:
control.setInputObjectViewDimension( new Dimension(1024, 768) );

// Please note, the previous listener was removed, because in general
// case the new inputObject will be a totally different and we can not
// assume in our component that the user will ALWAYS use just one
// listener for all the changed input objects.
//
// So, the user have to manually register a new listener like this:
control.getViewPort().addViewPortListener( some another listener that
implements the ViewPortListener interface );
```

There is a special “opacity” property in the viewPort part of our birds eye view control. It relates to the transparency of the birds eye view control view port. The transparency of the view port can be manually adjusted like this (of course, other opacity values (like 0.2, 0.123, and so on) can be applied):

```
// Set the semi-transparent view port
client.getViewPort().setOpacity( 0.5 );
```

```
// Set the fully transparent view port
client.getViewPort().setOpacity( 0.0 );

// Set the solid (non-transparent) view port
client.getViewPort().setOpacity( 1.0 );
```

5. **Future Enhancements**

   At some point in the future, the zoom level of the diagram may be tied to a resizable viewport overlay, so if the user resizes the viewport overlay to be bigger, the zoom panel being displayed will have its zoom level set smaller, etc...