Eclipse Custom Controls [1, 2]

by Shantha Ramachandran

Department of Computer Science, University of Manitoba, Winnipeg, Manitoba, Canada

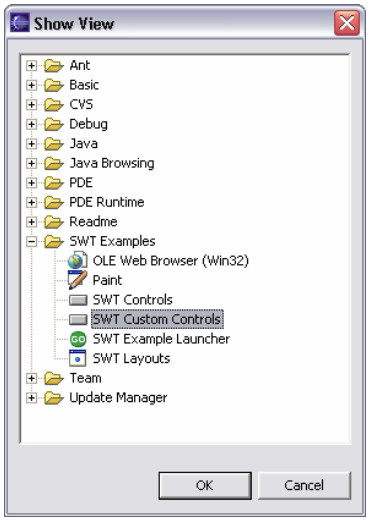
Last revised: October 22, 2003

Overview:

Tutorials 2 and 3 described the pre-defined SWT controls. Eclipse developers may also define custom controls that provide additional functionality. There are two different types of custom controls: basic widgets and compound widgets. In this tutorial, we examine a simple example of both types of widgets, and a more complex example which deals with event handling as well. We also briefly examine custom layout managers.

Custom Controls in Eclipse:

There are a number of Custom Controls that have already been implemented in Eclipse. These include CCombo, CLabel, CTabFolder, SashForm, StyledText and TableTree. To view these controls in action, first install the Examples plugin for Eclipse (see http://www.eclipse.org/downloads/index.php for how to install the Examples). After the Examples have been installed, click on Window -> Show View -> Other… From the window that appears, expand SWT Examples and click on SWT Custom Controls.



[1] This work was funded by an IBM Eclipse Innovation Grant.

All the Custom Controls identified above are illustrated in the examples plugin.

Building Your Own Custom Controls:

There are two different ways to create your own controls in Eclipse. The first is by subclassing *Canvas*, and the second is by subclassing *Composite*. *Canvas* is subclassed when you wish to draw basic widgets. This means that you do not need to use any other widgets within your control, but instead will draw the specific elements of your custom control onto a canvas. *Composite* is subclassed when you wish to group together a number of basic controls to create a larger or more functional control, called a compound widget.

Basic Widgets:

A custom control should subclass *Canvas* when the control can be drawn out by itself. The *Canvas* widget allows a blank working space on which you can draw out your widget. By creating a graphics context, you are provided with a number of methods that allow you to draw your control.

A Basic Widget Example:

Suppose we need to build a label which has a customizable border. To build this label, which is a basic widget, we first need to subclass *Canvas* and give the widget a constructor.

```
public class BorderLabel extends Canvas {
      public BorderLabel(Composite parent, int style) {
            super(parent, style);
      }
}
```

In order for the widget to be drawn, we need a PaintListener. A PaintListener detects when the widget needs to be repainted. Our PaintListener has a method, paintControl, which will be called when the widget needs redrawing. We can add the PaintListener to the control in the constructor:

```
addPaintListener(new PaintListener() {
   public void paintControl(PaintEvent e){
      BorderLabel.this.paintControl(e);
   }
});
```

To draw the widget, we need to next create the paintControl method. In this method, we create a graphics context which will do all of the drawing. In the following code, we draw

a string of text provided by the user, and then draw a rectangle around it. Note that we can set the line style and width of the line before we draw. In this example, **text** is the string set by the user, and **size** is the size of the border set by the user.

```
void paintControl(PaintEvent e) {
	GC gc = e.gc;
	Point pt = this.getSize();

	if (text != null) {
		gc.drawString(text,size,size);
	}

	if (size > 0) {
		gc.setLineWidth(size);
		gc.setLineStyle(style);
		gc.drawRectangle(0,0,pt.x-size,pt.y-size);
	}
}
```

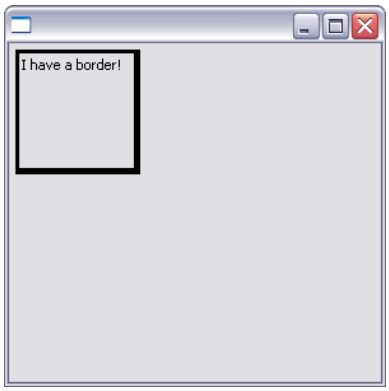
Now our widget can be drawn. All we need to do is add access methods that allow the user to set the text and the style of the border for the label.

```
public void setText(String text) {
	this.text = text;
	redraw();
}

public void setBorderSize(int size) {
	this.size = size;
	redraw();
}

public void setBorderStyle(int style) {
	this.style = style;
	redraw();
}
```

The redraw method that is called in each set method will queue a paint event for the widget. This event causes the PaintListener to execute its paintControl method, and the widget is redrawn. Now lets look at an example that uses our custom BorderLabel widget:

```
Display display = new Display();
Shell shell = new Shell(display);
shell.setSize(300,300);

BorderLabel borderLabel = new BorderLabel(shell,SWT.NONE);
borderLabel.setText("I have a border!");
borderLabel.setBorderSize(5);
borderLabel.setBorderStyle(SWT.LINE_SOLID);
borderLabel.setSize(100,100);
borderLabel.setLocation(5,5);

shell.open();

while(!shell.isDisposed()){
if(!display.readAndDispatch())
      display.sleep();
}
display.dispose();
```

When we run this program, we get a label with a 5 pixel sold border around the edge.



There are many other properties to a custom basic widget that you are free to implement, such as finding the preferred size of the widget. For more information, see the article on the Eclipse website **Creating Your Own Widgets Using SWT**.

Compound Widgets:

Compound widgets should be used when you want to create a widget out of other widgets. When you create a compound widget, you need to subclass *Composite*. Contrary to basic widgets, where you draw the widgets themselves, compound widgets are made up of other basic widgets. No drawing is involved. The *Composite* holds the other widgets together to form one complex widget.

A Compound Widget Example:

To illustrate how to create a compound widget, we will create the same widget over again, the *BorderLabel*. This time, it will be a compound widget.

Once again, the first thing we need to do is to create a constructor. In this widget, we will use a label and a group. We can initialize the widgets in the constructor as well. To make this example simplest, we will use a FillLayout to lay the widgets out within the control.

```
public class BorderLabel2 extends Composite {
	Label label;
	Group group;

	public BorderLabel2(Composite parent, int style) {
		super(parent, style);
		group = new Group(this, SWT.BORDER);
		label = new Label(group, SWT.NONE);
		setLayout(new FillLayout());
		group.setLayout(new FillLayout());
	}
}
```

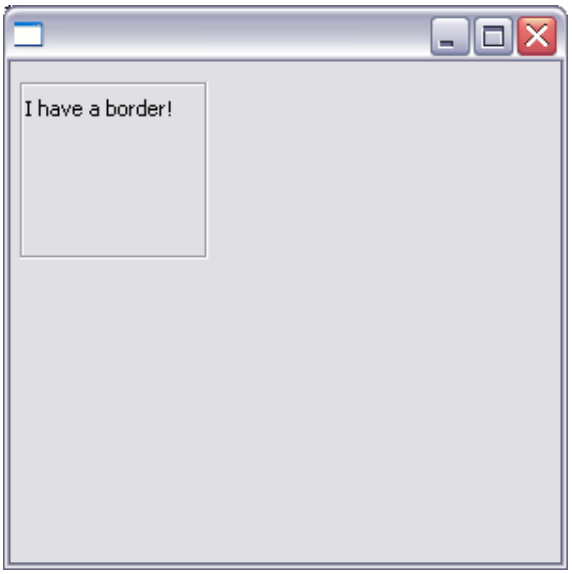
In the case of compound widgets, we do not need to draw anything. All we need to finish off the widget is to add set methods. Note that this widget is simpler than the basic widget we just created as we can only set the text, not the border size or style.

```
public void setText(String text) {
	label.setText(text);
	layout(true);
}
```

Let's look at an example that uses our *BorderLabel2*:

```
Display display = new Display();
Shell shell = new Shell(display);
shell.setSize(300,300);

BorderLabel2 borderLabel = new BorderLabel2(shell,SWT.NONE);
borderLabel.setText("I have a border!");
borderLabel.setSize(100,100);
borderLabel.setLocation(5,5);

shell.open();
while(!shell.isDisposed()){
if(!display.readAndDispatch())
	display.sleep();
}
display.dispose();
```

This results in the following window:



Once again, there are other methods that can be implemented for a compound widget. For example, if you do not wish to use a layout, you can implement a method that will position your children on a resize. For more information on compound widgets, see the article on the Eclipse website **Creating Your Own Widgets Using SWT**.

A Complex Custom Widget:

By looking at the previous two examples, you can hopefully differentiate between a basic and a compound widget. You should also be able to create a simple custom control. Next, we will go over an example of a more complex widget that uses event handlers as well.

A Zooming Scrolling Widget Example:

Suppose that we want to display images on a screen. However, we want these images to be zoomed from small to large. Also, when the images get bigger, we want to be able to scroll the image to see all of it. To do this, we will create a zoomed scrolled canvas widget that subclasses composite. This widget will contain a canvas, scrollbars, and a scale all grouped together in one composite.

If you looked at the Eclipse custom controls that have already been created, you will notice that a *ScrolledComposite* is one of the custom controls available for use. For the next example, we will cheat a little and use the *ScrolledComposite* custom control within our own custom control, instead of building it from scratch.

So, the different components that we will need in our custom control are a scrolled composite, a canvas, a scale and an image. We will also have a label to indicate the level of zoom that is applied to our picture, as well as its width and height.

```
public class ZoomedScrolledCanvas extends Composite {
	Label zoom;
	ScrolledComposite comp;
	Canvas canvas;
	Scale scale;
	Image image;
	int width, height;

	public ZoomedScrolledCanvas(Composite parent, int style) {
		super(parent, style);
		comp = new ScrolledComposite(this, SWT.BORDER |
			SWT.H_SCROLL | SWT.V_SCROLL);
		canvas = new Canvas(comp, 0);
		scale = new Scale(this, 0);
		zoom = new Label(this, 0);
		setLayout(new GridLayout());
	}
}
```

In our constructor, we create all our widgets, and give our control a Grid Layout. This will allow us to place the controls in specified positions. We need to do one more thing in our constructor though. We have to initialize and place the widgets. This can be done by calling setup methods from the constructor. We will have a setup method for the composite, the scale and the label.

To setup the composite, we simply give it layout data to expand horizontally and vertically within the widget. We also attach the canvas to the composite so that we can draw the image within the scrollbars.

```
public void setupComp() {
	comp.setAlwaysShowScrollBars(true);
	comp.setLayoutData(new GridData(GridData.FILL_BOTH));
	comp.setContent(canvas);
	layout(true);
}
```

To setup the scale, we need to specify the maximum, minimum, the increment, and the current selection. We also need to give it layout data.

```
public void setupScale() {
	scale.setLayoutData(new
		GridData(GridData.FILL_HORIZONTAL));
	scale.setMinimum(0);
	scale.setMaximum(100);
	scale.setIncrement(10);
	scale.setSelection(50);
	layout(true);
}
```

To setup the label, we need to give it layout data and some text.

```
public void setupZoom() {
	zoom.setText("Zoom Factor = 0");
	zoom.setLayoutData(new GridData(GridData.FILL_HORIZONTAL));
	layout(true);
}
```

The next thing we have to do is draw the image onto the canvas. Since the user will be specifying the image to draw, we need a method for setting the image. In this method, we will set the canvas size to the size of the scrolled composite, so the image will appear full size. We need to add a paint listener to draw the image. We will scale it from its full size to the size of the canvas. This is useful, because when we are zooming in and out, we will simply need to change the size of the canvas. When this happens, a paint event will be triggered and the image will be redrawn. Each time it is redrawn, it will take the size of the canvas, and so we are essentially changing the size of the image by zooming in and out. Note that we also have a dispose listener to dispose of the image when the canvas is disposed.

```
public void setImage(String imageString) {
	image = new Image(this.getDisplay(), imageString);

	canvas.setSize(comp.getSize().x, comp.getSize().y);
	canvas.setLocation(0,0);
	canvas.addPaintListener(new PaintListener() {
		public void paintControl(PaintEvent e) {
			GC gc = e.gc;
			width = comp.getSize().x;
			height = comp.getSize().y;
			gc.drawImage(image, 0, 0,
				image.getBounds().width,
				image.getBounds().height,
				0, 0, canvas.getSize().x,
				canvas.getSize().y);
		}
	});
	canvas.addDisposeListener(new DisposeListener() {
		public void widgetDisposed(DisposeEvent e) {
			image.dispose();
		}
	});
	redraw();
}
```

Now that we have set up the appearance of our control, we need to add a selection listener to our scale. When the user changes the value of the scale, the image should zoom in or out. In this listener, we calculate a new size based on the zoom factor and resize the canvas.

Remember we placed a paint listener on our canvas, so when the canvas is resized in the selection listener, the paint listener will generate a PaintEvent and the image will be drawn to its new size.

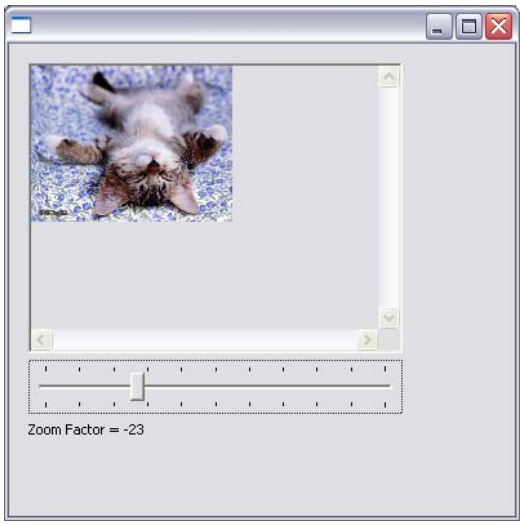
```
scale.addSelectionListener(new SelectionAdapter() {
      public void widgetSelected(SelectionEvent arg0) {
            zoom.setText("Zoom Factor = " +
                  (scale.getSelection()-50));
            int newWidth = width * (scale.getSelection())/50;
            int newHeight = height * (scale.getSelection())/50;
            canvas.setSize(newWidth, newHeight);
      }
});
```

This is the last step to creating our custom zoomed scrolled image widget. To see the widget in action, we use it in exactly the same way as we would use any widget.

```
Display display = new Display();
Shell shell = new Shell(display);
shell.setSize(400,400);

ZoomedScrolledCanvas zsc = new ZoomedScrolledCanvas(shell,
      SWT.NONE);
zsc.setSize(300,300);
zsc.setLocation(10,10);
zsc.setImage("cats.JPG");

shell.open();

while(!shell.isDisposed()){
if(!display.readAndDispatch())
      display.sleep();
}
display.dispose();
```

We set the size and position of our widget just as we would with a basic widget. The only custom parameter we must set is the image. We set the image to be cats.JPG. The following screen is generated from this code:

When we move the zoom scale down, the image gets smaller:



When we move the zoom scale up, the image gets larger and we can scroll through it:

So far, we have described basic custom widgets, compound custom widgets, and event handling associated with custom widgets. There are many things you can do with a custom widget, it all depends on what you need it to do! For more information on how to build custom widgets, download the Eclipse custom widget source code (org.eclipse.swt.custom) or see the online article **Creating Your Own Widgets Using SWT**.

Custom Layouts

In the same fashion that you can create custom widgets, you can also create custom layouts to arrange your widgets in a specific manner. Before building a custom layout manager, make sure that the layouts provided by SWT are not sufficient and that a custom layout really is necessary.

Building a Simple Layout

The following is an example to show how to build a simple layout. This example creates a layout that arranges widgets in a square, so we'll call it the *SquareLayout*. It will take the widgets on a composite and arrange them row by row in a square format. The widgets will fill the space of the composite.

The first thing we need to do is subclass the *Layout* class. We will add fixed values for the margin and the spacing between the widgets on the composite.

```
public class SquareLayout extends Layout {
	public static final int MARGIN = 2;
	public static final int SPACING = 2;

	// constructor
	public SquareLayout() {
		super();
	}
}
```

The next thing we will do is override the *layout* method. This is the method that will actually lay out the widgets on the composite. But before we can know where to lay the child widgets, we need to know the size of the widgets, and the size that the composite is going to be.

We will create a method called *initialize*. In this method, we will figure out how many widgets will be in a row or a column. Since we are creating a square, these numbers are the same. If we find the square root of the total number of children, and round this number up, we will get the number of rows or columns.

```
num = (int) Math.ceil(Math.sqrt(children.length));
```

Next, we need to calculate the width and height of the composite when all the widgets are laid out on it. Since all the widgets are the same size, we need to find the largest size and the largest height of all the widgets. Then we can just multiply this number by the number of rows and columns to get the total width and height of the composite. All the widgets will grow to be the same size as the largest widget, similar to the *FillLayout*.

```
sizes = new Point [children.length];
int tempWidth = 0, tempHeight = 0;
for (int i = 0; i < children.length; i+=num) {
      sizes[i] = children[i].computeSize(SWT.DEFAULT,
            SWT.DEFAULT,true);
      partialWidth = Math.max(sizes[i].x, tempWidth);
      partialHeight = Math.max(sizes[i].y, tempHeight);
}
// find the height and width for the composite
totalWidth = partialWidth*num + (children.length - 1)*SPACING;
totalHeight = partialHeight*num + (children.length - 1)*SPACING;
```

After completing the *initialize* method, we can now create our *layout* method. Here, all we need to do is place the widgets, taking margins and spacing into account. Since we know the size that the widgets must be from our initialization, we can simply specify the coordinates of the widgets.

One thing to note is that if the composite size is set to be larger than our calculated size, we will choose the larger size and grow the widgets to fit.

```
Rectangle rect = composite.getClientArea();
int x, y = MARGIN;
int width = Math.max(partialWidth, rect.width/num-2*MARGIN);
int height = Math.max(partialHeight, rect.height/num-2*MARGIN);
for (int i = 0; i < children.length; i+=num) {
      x = MARGIN;
      for (int j=i; j<children.length && j<i+num; j++) {
            children[j].setBounds(x, y, width, height);
            x += (int) width + SPACING;
      }
      y += height + SPACING; // add 2 for spacing
}
```

After we have completed this, the only thing left to do is override the *ComputeSize* method. This will return the preferred size of the composite using your layout.

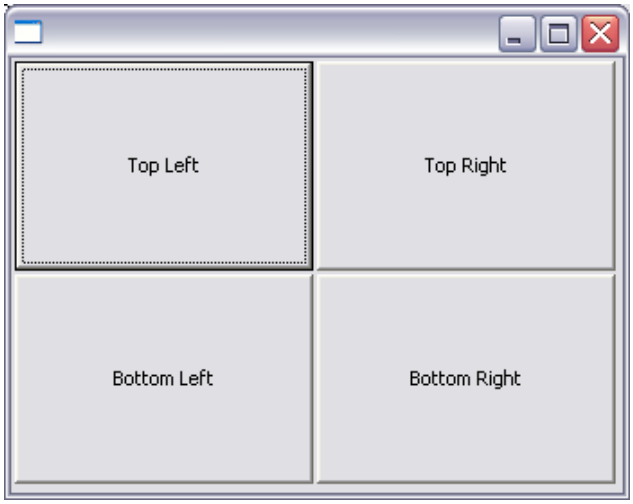
```
Point computeSize(Composite composite, int wHint, int hHint, Boolean
            flushCache) {
      Control children[] = composite.getChildren();
      if (flushCache || sizes==null || sizes.length!=children.length) {
            initialize(children);
      }
      // if they have given a width or height, use it
      // otherwise, use the calculated width and height
      int width = wHint, height = hHint;
      if (wHint == SWT.DEFAULT) width = totalWidth;
      if (hHint == SWT.DEFAULT) height = totalHeight;
      // account for margin
      return new Point(width + 2*MARGIN, height + 2*MARGIN);
}
```

We will create a sample program to test our layout. In this program, we will create four buttons. The layout should arrange the four buttons in a square:

```
      Display display = new Display();
      Shell shell = new Shell(display);
      shell.setLayout(new SquareLayout());

      Button b1 = new Button(shell, SWT.PUSH);
      b1.setText("Top Left");
      Button b2 = new Button(shell, SWT.PUSH);
      b2.setText("Top Right");
      Button b3 = new Button(shell, SWT.PUSH);
      b3.setText("Bottom Left");
      Button b4 = new Button(shell, SWT.PUSH);
      b4.setText("Bottom Right");

      shell.pack();
      shell.open();

      while(!shell.isDisposed()){
      if(!display.readAndDispatch())
            display.sleep();
      }
      display.dispose();
```

From this we get the following window:

When we make this window larger, you can see that the buttons all grow and remain in the same proportions to each other.



Note that if the number of widgets is not a perfect square, there will not be an even number of widgets in each row and column:

There are many other things you can do with your custom layout. For more information, see the online articles **Creating Your Own Widgets Using SWT** and **Understanding Layouts in SWT**.