DeveloperManual VisualizerTutorial

Tutorial - Making a JMeter Visualizer

Of the component types, visualizers require greater depth in Swing than something like controllers, functions or samplers. You can find the full source for the distribution graph in "components/org/apache/jmeter/visualizers/". The distribution graph visualizer is divided into two classes.

- **DistributionGraphVisualizer** – visualizer which Jmeter instantiates
- **DistributionGraph** – Jcomponent which draws the actual graph

The easiest way to write a visualizer is to do the following:

1. extend org.apache.jmeter.visualizers.gui.AbstractVisualizer
2. implement any additional interfaces need for call back and event notification. For example, the DistributionGraphVisualizer implements the following interfaces:
    a. ImageVisualizer
    b. ItemListener – according to the comments in the class, ItemListener is out of date and isn't used anymore.
    c. GraphListener
    d. Clearable

AbstractVisualizer provides some common functionality, which most visualizers like "graph results" use. The common functionality provided by the abstract class includes:

- configure test elements – This means it create a new ResultCollector, sets the file and sets the error log
- create the stock menu
- update the test element when changes are made
- create a file panel for the log file
- create the title panel

In some cases, you may not want to display the menu for the file textbox. In that case, you can override the "init()" method. Here is the implementation for DistributionGraphVisualizer.

```
1    /**
2     * Initialize the GUI.
3     */
4    private void init()
5    {
6         this.setLayout(new BorderLayout());
7
8         // MAIN PANEL
9         Border margin = new EmptyBorder(10, 10, 5, 10);
10
11        this.setBorder(margin);
12
13        // Set up the graph with header, footer, Y axis and graph display
14        JPanel graphPanel = new JPanel(new BorderLayout());
15        graphPanel.add(createGraphPanel(), BorderLayout.CENTER);
16        graphPanel.add(createGraphInfoPanel(), BorderLayout.SOUTH);
17
18        // Add the main panel and the graph
19        this.add(makeTitlePanel(), BorderLayout.NORTH);
20        this.add(graphPanel, BorderLayout.CENTER);
21    }
```

The first thing the init method does is create a new BorderLayout. Depending on how you want to layout the widgets, you may want to use a different layout manager. Keep in mind using different layout managers is for experts.

The second thing the init method does is create a border. If you want to increase or decrease the border, change the four integer values. Each integer value represents pixels. If you want your visualizer to have no border, skip lines 9 and 11. Line 15 calls "createGraphPanel," which is responsible for configuring and adding the DistributionGraph to the visualizer.

```
1    private Component createGraphPanel()
2    {
3        graphPanel = new JPanel();
4        graphPanel.setBorder(BorderFactory.createBevelBorder(
5        BevelBorder.LOWERED,Color.lightGray,Color.darkGray));
6        graphPanel.add(graph);
7        graphPanel.setBackground(Color.white);
8        return graphPanel;
9    }
```

At line 6, the graph component is added to the graph panel. The constructor is where a new instance of DistributionGraph is created.

```
public DistributionGraphVisualizer()
{
    model = new SamplingStatCalculator("Distribution");
    graph = new DistributionGraph(model);
    graph.setBackground(Color.white);
    init();
}
```

The constructor of DistributionGraphVisualizer is responsible for creating the model and the graph. Every time a new result is complete, the engine passes the result to all the listeners by calling add(SampleResult res). The visualizer passes the new SampleResult to the model.

```
1    public synchronized void add(SampleResult res)
2    {
3        model.addSample(res);
4        updateGui(model.getCurrentSample());
5    }
```

In the case of the DistributionGraphVisualizer, the "add" method doesn't acutally update the graph. Instead, it calls "updateGui" in line four.

```
public synchronized void updateGui(Sample s)
{
    // We have received one more sample
    if (delay == counter) {
        updateGui();
        counter = 0;
    } else {
            counter++;
    }
}
```

Unlike GraphVisualizer, the distribution graph attempts to show how the results clump; therefore the DistributionGraphVisualizer delays the update. The default delay is 10 sampleresults.

```
1    public synchronized void updateGui()
2    {
3        if (graph.getWidth() < 10){
4            graph.setPreferredSize(new Dimension(getWidth() - 40, getHeight() - 160));
5        }
6        graphPanel.updateUI();
7        graph.repaint();
8    }
```

Lines 3 to 4 are suppose to resize the graph, if the user resizes the window or drags the divider. Line 6 updates the panel containing the graph. Line 7 triggers the update of the DistributionGraph. Before we cover writing graphcs, there are couple of important methods visualizer must implement.

```
public String getLabelResource()
{
    return "distribution_graph_title";
}
```

The label resource retrieves the name of the visualizer from the properties file. The file is located in "core/org/apache/jmeter/resources". It's best not to hardcode the name of the visualizer. Message.properties file is organized alphabetically, so adding a new entry is easy.

```
public synchronized void clear()
{
    this.graph.clear();
    model.clear();
    repaint();
}
```

Every component in Jmeter should implement logic for "clear()" method. If this isn't done, the component will not clear the UI or model when the user tries to clear the last results and run a new test. If clear is not implemented, it can result in a memory leak.

```
public JComponent getPrintableComponent()
{
    return this.graphPanel;
}
```

The last method visualizers should implement is "getPrintableComponent()". The method is responsible for returning the Jcomponent that can be saved or printed. This feature was recently added so that users can save a screen capture of any given visualizer.

GraphListener

Visualizers should implement GraphListener. This is done to make it simpler to add new Sample instances to listeners. As a general rule, if the a custom graph does not plot every single sample, it does not need to implement the interface.

```
public interface GraphListener
{
    public void updateGui(Sample s);
    public void updateGui();
}
```

The important method in the interface is "updateGui(Sample s)". From DistributionGraphVisualizer, we see it calls graph.repaint() to refresh the graph. In most cases, the implementation of updateGui(Sample s) should do just that.

ItemListener

Visualizers generally do not need to implement this interface. The interface is used with combo boxes, checkbox and lists. If your visualizer uses one of these and needs to know when it has been updated, the visualizer will need to implement the interface. For an example of how to implement the interface, please look at GraphVisualizer.

Writing Custom Graphs

For those new to Swing and haven't written custom Jcomponents yet, I would suggest getting a book on Swing and get a good feel for how Swing widgets work. This tutorial will not attempt to explain basic Swing concepts and assumes the reader is already familiar with the Swing API and MVC (Model View Controller) design pattern. From the constructor of DistributionGraphVisualizer, we see a new instance of DistributionGraph is created with an instance of the model.

```
public DistributionGraph(SamplingStatCalculator model)
{
    this();
    setModel(model);
}
```

The implementation of "setModel" method is straight forward.

```
private void setModel(Object model)
{
    this.model = (SamplingStatCalculator) model;
    repaint();
}
```

Notice the method calls "repaint" after it sets the model. If "repaint" isn't called, it can cause the GUI to not draw the graph. Once the test starts, the graph would redraw, so calling "repaint" isn't critical.

```
public void paintComponent(Graphics g)
{
    super.paintComponent(g);
    final SamplingStatCalculator m = this.model;
    synchronized (m)
    {
            drawSample(m, g);
    }
}
```

The other important aspect of updating the widget is placing the call to drawSample within a synchronized block. If drawSample wasn't synchronized, Jmeter would throw a ConcurrentModificationException at runtime. Depending on the test plan, there may be a dozen or more threads adding results to the model. The synchronized block does not affect the accuracy of each individual request and time measurement, but it does affect Jmeter's ability to generate large loads. As the number of threads in a test plan increases, the likelihood a thread will have to wait until the graph is done redrawing before starting a new request increases. Here is the implementation of drawSample.

```java
private void drawSample(SamplingStatCalculator model, Graphics g)
{
    width = getWidth();
    double height = (double)getHeight() - 1.0;

    // first lets draw the grid
    for (int y=0; y < 4; y++) {
        int q1 = (int)(height - (height * 0.25 * y));
        g.setColor(Color.lightGray);
        g.drawLine(xborder,q1,width,q1);
        g.setColor(Color.black);
        g.drawString(String.valueOf((25 * y) + "%"),0,q1);
    }
    g.setColor(Color.black);
    // draw the X axis
    g.drawLine(xborder,(int)height,width,(int)height);
    // draw the Y axis
    g.drawLine(xborder,0,xborder,(int)height);
    // the test plan has to have more than 200 samples
    // for it to generate half way decent distribution
    // graph. the larger the sample, the better the
    // results.
    if (model != null && model.getCount() > 50) {
        // now draw the bar chart
        Number ninety = model.getPercentPoint(0.90);
        Number fifty = model.getPercentPoint(0.50);
        total = model.getCount();
        Collection values = model.getDistribution().values();
        Object[] objval = new Object[values.size()];
        objval = values.toArray(objval);
        // we sort the objects
        Arrays.sort(objval,new NumberComparator());
        int len = objval.length;
        for (int count=0; count < len; count++) {
            // calculate the height
            Number[] num = (Number[])objval[count];
            double iper = (double)num[1].intValue()/(double)total;
            double iheight = height * iper;
            // if the height is less than one, we set it
            // to one pixel
            if (iheight < 1){
                iheight = 1.0;
            }
            int ix = (count * 4) + xborder + 5;
            int dheight = (int)(height - iheight);
            g.setColor(Color.blue);
            g.drawLine(ix -1,(int)height,ix -1,dheight);
            g.drawLine(ix,(int)height,ix,dheight);
            g.setColor(Color.black);
            // draw a red line for 90% point
            if (num[0].longValue() == ninety.longValue()){
                g.setColor(Color.red);
                g.drawLine(ix,(int)height,ix,55);
                g.drawLine(ix,(int)35,ix,0);
                g.drawString("90%",ix - 30,20);
                g.drawString(String.valueOf(num[0].longValue()),ix + 8, 20);
            }
            // draw an orange line for 50% point
            if (num[0].longValue() == fifty.longValue()){
                g.setColor(Color.orange);
                g.drawLine(ix,(int)height,ix,30);
                g.drawString("50%",ix - 30,50);
                g.drawString(String.valueOf(num[0].longValue()),ix + 8, 50);
            }
        }
    }
}
```

In general, the rendering of the graph should be fairly quick and shouldn't be a bottleneck. As a general rule, it is a good idea to profile custom plugins. The only way to make sure a visualizer isn't a bottleneck is to run it with a tool like Borland OptimizeIt. A good way to test a plugin is to create a simple test plan and run it. The heap and garbage collection behavior should be regular and predictable.