THE STRATEGY PATTERN

The Strategy pattern is much like the State pattern in outline, but a little different in intent. The Strategy pattern consists of a number of related algorithms encapsulated in a driver class called the Context. Your client program can select one of these differing algorithms or in some cases the Context might select the best one for you. The intent, like the State pattern, is to switch easily between algorithms without any monolithic conditional statements. The difference between State and Strategy is that the user generally chooses which of several strategies to apply and that only one strategy at a time is likely to be instantiated and active within the Context class. By contrast, as we have seen, it is likely that all of the different States will be active at once and switching may occur frequently between them. In addition, Strategy encapsulates several algorithms that do more or less the same thing, while State encapsulates related classes that each do something somewhat different. Finally, the concept of transition between different states is completely missing in the Strategy pattern.

Motivation

A program which requires a particular service or function and which has several ways of carrying out that function is a candidate for the Strategy pattern. Programs choose between these algorithms based on computational efficiency or user choice. There can be any number of strategies and more can be added and any of them can be changed at any time.

There are a number of cases in programs where we'd like to do the same thing in several different ways. Some of these are listed in the *Smalltalk Companion*:

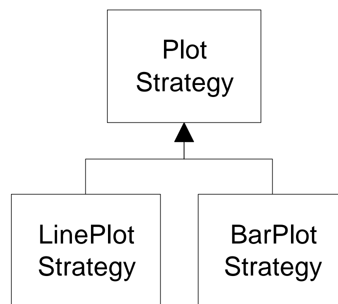
- Save files in different formats.
- Compress files using different algorithms
- Capture video data using different compression schemes
- Use different line-breaking strategies to display text data.
- Plot the same data in different formats: line graph, bar chart or pie chart.

In each case we could imagine the client program telling a driver module (Context) which of these strategies to use and then asking it to carry out the operation.

The idea behind Strategy is to encapsulate the various strategies in a single module and provide a simple interface to allow choice between these strategies. Each of them should have the same programming interface, although they need not all be members of the same class hierarchy. However, they do have to implement the same programming interface.

Sample Code

Let's consider a simplified graphing program that can present data as a line graph or a bar chart. We'll start with an abstract PlotStrategy class and derive the two plotting classes from it:



Since each plot will appear in its own frame, our base PlotStrategy class will be derived from JFrame:

```
public abstract class PlotStrategy extends JFrame
{
  protected float[] x, y;
  protected Color color;
  protected int width, height;

  public PlotStrategy(String title)   {
     super(title);
     width = 300;      height =200;
     color = Color.black;
     addWindowListener(new WindAp(this));
  }
  //--------------------------------------
  public abstract void plot(float xp[], float yp[]);
  //--------------------------------------
  public void setPenColor(Color c)   {
```

```
      color = c;
  }
```

The important part is that all of the derived classes must implement a method called *plot* with two float arrays as arguments. Each of these classes can do any kind of plot that is appropriate.

Note that we don't derive it from our special JxFrame class, because we don't want the entire program to exit if we close one of these subsidiary windows. Instead, we add a WindowAdapter class that just hides the window if it is closed.

```
class WindAp extends WindowAdapter
{
   JFrame fr;
   public WindAp(JFrame f)    {
      fr = f;             //copy Jframe instance
   }
   public void WindowClosing(WindowEvent e)    {
    fr.setVisible(false);  //hide window
   }
}
```

The Context

The Context class is the traffic cop that decides which strategy is to be called. The decision is usually based on a request from the client program, and all that the Context needs to do is to set a variable to refer to one concrete strategy or another.

```
public class Context
{
 //this object selects one of the strategies
 //to be used for plotting
 //the plotStrategy variable points to selected strategy
   private PlotStrategy plotStrategy;
   float x[], y[];        //data stored here
//---------------------------------
  public Context() {
 setLinePlot(); //make sure it is not null
}
//---------------------------------
//make current strategy the Bar Plot
  public void setBarPlot()
{ plotStrategy = new BarPlotStrategy(); }
//---------------------------------
//make current strategy the Line Plot
  public void setLinePlot()
{ plotStrategy = new LinePlotStrategy(); }
//---------------------------------
//call plot method of current strategy
```

```
  public void plot() {
     plotStrategy.plot(x, y);
  }
//----------------------------------
  public void setPenColor(Color c) {
     plotStrategy.setPenColor(c);
  }
  //----------------------------------
  public void readData(String filename)
{      //read data from datafile somehow
}
}
```

The Context class is also responsible for handling the data. Either it obtains the data from a file or database or it is passed in when the Context is created. Depending on the magnitude of the data, it can either be passed on to the plot strategies or the Context can pass an instance of itself into the plot strategies and provide a public method to fetch the data.

The Program Commands

This simple program is just a panel with two buttons that call the two plots:

Each of the buttons is a command object that sets the correct strategy and then calls the Context's plot routine. For example, here is the complete Line graph button class:

```
public class JGraphButton extends JButton
       implements Command
{
   Context context;
   public JGraphButton(ActionListener act, Context ctx)
   {
   super("Line graph");          //button label
   addActionListener(act);       //add listener
   context  = ctx;               //copy context
   }
   //--------------------------------
   public void Execute()    {
      context.setPenColor(Color.red); //set color of plot
      context.setLinePlot();          //set kind of plot
      context.readData("data.txt");   //read the data
      context.plot();                 //plot the data
```

```
    }
}
```

The Line and Bar Graph Strategies

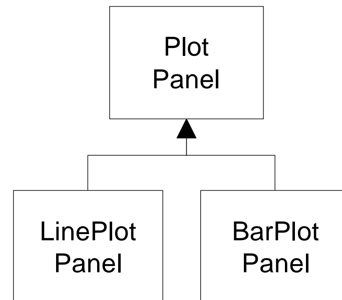
The two strategy classes are pretty much the same: they set up the window size for plotting and call a plot method specific for that display panel. Here is the Line graph Strategy:

```
public class LinePlotStrategy extends PlotStrategy
{
   LinePlotPanel lp;
         public LinePlotStrategy()
  {
     super("Line plot");
     lp = new LinePlotPanel();
     getContentPane().add(lp);
  }
//--------------------------------------
   public void plot(float[] xp, float[] yp)
   {
      x = xp;  y = yp;         //copy in data
      findBounds();            //sets maxes and mins
      setSize(width, height);
      setVisible(true);
      setBackground(Color.white);
      lp.setBounds(minX, minY, maxX, maxY);
      lp.plot(xp, yp, color); //set up plot data
      repaint();              //call paint to plot
   }
}
```

Drawing Plots in Java

Since Java GUI is event-driven, you don't actually write a routine that draws lines on the screen in direct response to the plot command event. Instead you provide a panel whose *paint* event carries out the plotting when that event is called. The *repaint()* method shown above ensures that it will be called right away.

We create a PlotPanel class based on JPanel and derive two classes from it for the actual line and bar plots:



The base PlotPanel class contains the common code for scaling the data to the window.

```
public class PlotPanel extends JPanel
{
   float xfactor, yfactor;
   int xpmin, ypmin, xpmax, ypmax;
   float minX, maxX, minY, maxY;
   float x[], y[];
   Color color;
//--------------------------------------------
 public void setBounds(float minx, float miny,
              float maxx, float maxy)   {
     minX=minx;      maxX= maxx;
     minY=miny;      maxY = maxy;
  }
//--------------------------------------------
 public void plot(float[] xp, float[] yp, Color c)  {
  x = xp;      //copy in the arrays
  y = yp;
  color = c;   //and color

  //compute bounds and sclaing factors
  int w = getWidth() - getInsets().left -
                          getInsets().right;
  int h = getHeight() - getInsets().top -
                                 getInsets().bottom;

  xfactor = (0.9f * w) / (maxX - minX);
  yfactor = (0.9f * h)/ (maxY - minY);

  xpmin = (int)(0.05f * w);   ypmin = (int)(0.05f * h);
  xpmax = w - xpmin;   ypmax = h - ypmin;
  repaint();      //this causes the actual plot
 }
//--------------------------------------
protected int calcx(float xp) {
   return (int)((xp-minX) * xfactor + xpmin);
```

```
}
 protected int calcy(float yp) {
   int ypnt = (int)((yp-minY) * yfactor);
   return ypmax - ypnt;
 }
}
```

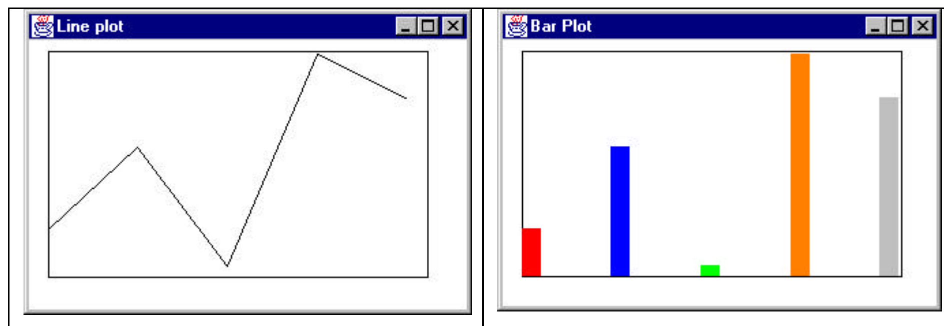
The two derived classes simply implement the *paint* method for the two kinds of graphs. Here is the one for the Line plot.

```
public class LinePlotPanel extends PlotPanel
{
 public void paint(Graphics g)
 {
   int xp = calcx(x[0]);       //get first point
   int yp = calcy(y[0]);
   g.setColor(Color.white);    //flood background
   g.fillRect(0,0,getWidth(), getHeight());
   g.setColor(Color.black);

   //draw bounding rectangle
   g.drawRect(xpmin, ypmin, xpmax, ypmax);
   g.setColor(color);

   //draw line graph
   for(int i=1; i< x.length; i++)
   {
      int xp1 = calcx(x[i]);         //get n+1st point
      int yp1 = calcy(y[i]);
      g.drawLine(xp, yp, xp1, yp1); //draw line
      xp = xp1;                      //copy for next loop
      yp = yp1;
   }
 }
}
```

The final two plots are shown below:



Consequences of the Strategy Pattern

Strategy allows you to select one of several algorithms dynamically. These algorithms can be related in an inheritance hierarchy or they can be unrelated as long as they implement a common interface. Since the Context switches between strategies at your request, you have more flexibility than if you simply called the desired derived class. This approach also avoids the sort of condition statements than can make code hard to read ad maintain.

On the other hand, strategies don't hide everything. The client code must be aware that there are a number of alternative strategies and have some criteria for choosing among them. This shifts an algorithmic decision to the client programmer or the user.

Since there are a number of different parameters that you might pass to different algorithms, you have to develop a Context interface and strategy methods that are broad enough to allow for passing in parameters that are not used by that particular algorithm. For example the *setPenColor* method in our PlotStrategy is actually only used by the LineGraph strategy. It is ignored by the BarGraph strategy, since it sets up its own list of colors for the successive bars it draws.