THE TEMPLATE PATTERN

Whenever you write a parent class where you leave one or more of the methods to be implemented by derived classes, you are in essence using the Template pattern. The Template pattern formalizes the idea of defining an algorithm in a class, but leaving some of the details to be implemented in subclasses. In other words, if your base class is an abstract class, as often happens in these design patterns, you are using a simple form of the Template pattern.

Motivation

Templates are so fundamental, you have probably used them dozens of times without even thinking about it. The idea behind the Template pattern is that some parts of an algorithm are well defined and can be implemented in the base class, while other parts may have several implementations and are best left to derived classes. Another main theme is recognizing that there are some basic parts of a class that can be factored out and put in a base class so that they do not need to be repeated in several subclasses.

For example, in developing the PlotPanel classes we used in the Strategy pattern examples, we discovered that in plotting both line graphs and bar charts we needed similar code to scale the data and compute the x-and y pixel positions.

```
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 scaling factors
```

```
  int w = getWidth();
  int h = getHeight();
  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;
 }
}
```

Thus, these methods all belonged in a base PlotPanel class without any actual plotting capabilities. Note that the *plot* method sets up all the scaling constants and just calls *repaint*. The actual paint method is deferred to the derived classes. Since the JPanel class always has a *paint* method, we don't want to declare it as an abstract method in the base class, but we do need to override it in the derived classes.

Kinds of Methods in a Template Class

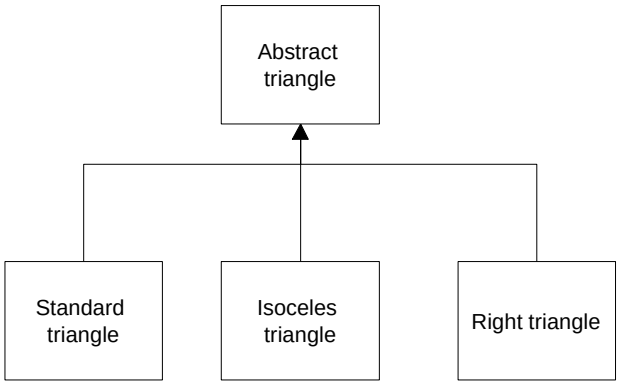
A Template has four kinds of methods that you can make use of in derive classes:

1. Complete methods that carry out some basic function that all the subclasses will want to use, such as *calcx* and *calcy* in the above example. These are called *Concrete methods.*
2. Methods that are not filled in at all and must be implemented in derived classes. In Java , you would declare these as *abstract* methods, and that is how they are referred to in the pattern description.
3. Methods that contain a default implementation of some operations, but which may be overridden in derived classes. These are called *Hook* methods. Of course this is somewhat arbitrary, because in Java you can override any public or protected method in the derived class, but Hook methods are intended to be overridden, while Concrete methods are not.

4. Finally, a Template class may contain methods which themselves call any combination of abstract, hook and concrete methods. These methods are not intended to be overridden, but describe an algorithm without actually implementing its details. *Design Patterns* refers to these as Template methods.

Sample Code

Let's consider a simple program for drawing triangles on a screen. We'll start with an abstract Triangle class, and then derive some special triangle types from it.



Our abstract Triangle class illustrates the Template pattern:

```
public abstract class Triangle
{
  Point p1, p2, p3;
  //--------------------------------------
  public Triangle(Point a, Point b, Point c)
  {
     //save
     p1 = a; p2 = b; p3 = c;
  }
  //--------------------------------------
  public void draw(Graphics g)
  {
   //This routine draws a general triangle
     drawLine(g, p1, p2);
     Point current = draw2ndLine(g, p2, p3);
     closeTriangle(g, current);
  }
  //--------------------------------------
  public void drawLine(Graphics g, Point a, Point b)
```

```
  {
     g.drawLine(a.x, a.y, b.x, b.y);
  }
  //--------------------------------------
  //this routine has to be implemented
  //for each triangle type.
        abstract public Point
             draw2ndLine(Graphics g, Point a, Point b);
  //--------------------------------------
  public void closeTriangle(Graphics g, Point c)
  {
     //draw back to first point
     g.drawLine(c.x, c.y, p1.x, p1.y);
  }
}
```

This Triangle class saves the coordinates of three lines, but the *draw* routine draws only the first and the last lines. The all important *draw2ndLine* method that draws a line to the third point is left as an abstract method. That way the derived class can move the third point to create the kind of rectangle you wish to draw.

This is a general example of a class using the Template pattern. The *draw* method calls two concrete base class methods and one abstract method that must be overridden in any concrete class derived from Triangle.

Another very similar way to implement the case triangle class is to include default code for the *draw2ndLine* method.

```
public Point draw2ndLine(Graphics g, Point a, Point b)
{
  g.drawLine(a.x, a.y, b.x, b.y);
  return b;
}
```

In this case, the *draw2ndLine* method becomes a Hook method that can be overridden for other classes.

Drawing a Standard Triangle

To draw a general triangle with no restrictions on its shape, we simple implement the *draw2ndLine* method in a derived *stdTriangle* class:

```
public class stdTriangle extends Triangle
{
   public stdTriangle(Point a, Point b, Point c)
   {
    super(a, b, c);
   }
   public Point draw2ndLine(Graphics g, Point a, Point b)
```

```
   {
      g.drawLine(a.x, a.y, b.x, b.y);
      return  b;
   }
}
```

Drawing an Isoceles Triangle

This class computes a new third data point that will make the two sides equal and length and saves that new point inside the class.

```
public class IsocelesTriangle extends Triangle
{
   Point newc;
   int newcx, newcy;
   int incr;

   public IsocelesTriangle(Point a, Point b, Point c)
   {
      super(a, b, c);
      double dx1 = b.x - a.x;   double dy1 = b.y - a.y;
      double dx2 = c.x - b.x;   double dy2 = c.y - b.y;

      double side1 = calcSide(dx1, dy1);
      double side2 = calcSide(dx2, dy2);

      if (side2 < side1)
         incr = -1;
      else
         incr = 1;

      double slope = dy2 / dx2;
      double intercept = c.y - slope* c.x;

      //move point c so that this is an isoceles triangle
      newcx = c.x; newcy = c.y;
      while(Math.abs(side1 - side2) > 1)   {
         newcx += incr;    //iterate a pixel at a time
         newcy = (int)(slope* newcx + intercept);
         dx2 = newcx - b.x;
         dy2 = newcy - b.y;
         side2 = calcSide(dx2, dy2);
      }
      newc = new Point(newcx, newcy);
   }
   //---------------------------------------
   //calculate length of side
   private double calcSide(double dx, double dy)
   {
      return Math.sqrt(dx*dx + dy*dy);
   }
```

When the Triangle class calls the *draw* method, it calls this new version of *draw2ndLine* and draws a line to the new third point. Further, it returns that new point to the *draw* method so it will draw the closing side of the triangle correctly.

```
//draws 2nd line using saved new point
   public Point draw2ndLine(Graphics g, Point b, Point c)
   {
    g.drawLine(b.x, b.y, newc.x, newc.y);
    return newc;
   }
```

The Triangle Drawing Program

The main program simple creates instances of the triangles you want to draw. Then, it adds them to a Vector in the TPanel class.

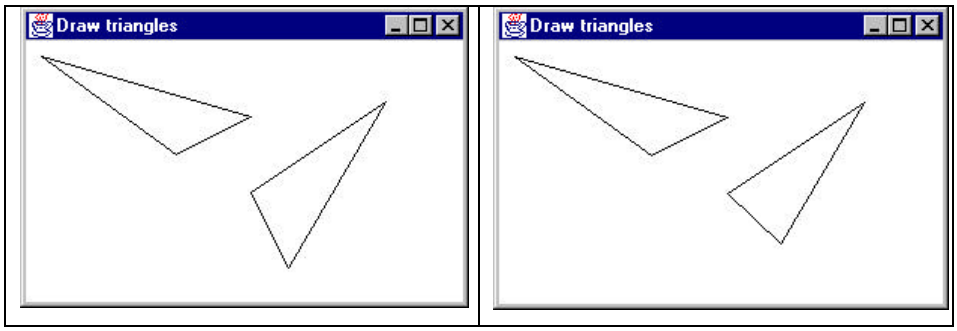
```
public TriangleDrawing()
{
 super("Draw triangles");
 TPanel tp = new TPanel();
 t = new stdTriangle(new Point(10,10), new Point(150,50),
                     new Point(100, 75));
 t1 = new stdTriangle(new Point(150,100), new Point(240,40), \
                          new Point(175, 150));
 tp.addTriangle(t);        //add to triangle list
 tp.addTriangle(t1);       //in the TPanel

 getContentPane().add(tp);
 setSize(300, 200);
 setBackground(Color.white);
 setVisible(true);
}
```

It is the *paint* routine in this class that actually draws the triangles.

```
class TPanel extends Jpanel {
  Vector triangles;
 public TPanel()  {
   triangles = new Vector();   //list of triangles
 }
//-----------------------------------------
 public void addTriangle(Triangle t)    {
   triangles.addElement(t);      //add more to list
 }
//-----------------------------------------
//draw all the triangles
 public void paint(Graphics g)    {
    for (int i = 0; i < triangles.size(); i++)  {
Triangle tngl = (Triangle)triangles.elementAt(i);
tngl.draw(g);
```

```
    }
  }
}
```

An example of two standard triangles is shown below in the left window, and the same code using an isoceles triangle in the right window.



Templates and Callbacks

Design Patterns points out that Templates can exemplify the "Hollywood Principle," or "Don't call us, we'll call you." The idea here is that methods in the base class seem to call methods in the derived classes. The operative word here is *seem*. If we consider the *draw* code in our base Triangle class, we see that there are 3 method calls:

```
drawLine(g, p1, p2);
Point current = draw2ndLine(g, p2, p3);
closeTriangle(g, current);
```

Now *drawLine* and *closeTriangle* are implemented in the base class. However, as we have seen, the *draw2ndLine* method is not implemented at all in the base class, and various derived classes can implement it differently. Since the actual methods that are being called are in the derived classes, it appears as though they are being called from the base class.

If this idea make you uncomfortable, you will probably take solace in recognizing that *all* the method calls originate from the derived class, and that these calls move up the inheritance chain until they find the first class which implements them. If this class is the base class, fine. If not, it could be any other class in between. Now, when you call the *draw* method, the derived class moves up the inheritance tree until it finds an implementation of draw. Likewise, for each method called from within draw, the derived class starts at the currently class and moves up the tree to find each method. When it gets to

the *draw2ndLine* method, it finds it immediately in the current class. So it isn't "really" called from the base class, but it does sort of seem that way.

Summary and Consequences

Template patterns occur all the time in OO software and are neither complex nor obscure in intent. They are normal part of OO programming and you shouldn't try to make them more abstract than they actually are.

The first significant point is that your base class may only define some of the methods it will be using, leaving the rest to be implemented in the derived classes. The second major point is that there may be methods in the base class which call a sequence of methods, some implemented in the base class and some implemented in the derived class. This Template method defines a general algorithm, although the details may not be worked out completely in the base class.

Template classes will frequently have some abstract methods that you must override in the derived classes, and may also have some classes with a simple "place-holder" implementation that you are free to override where this is appropriate. If these place-holder classes are called from another method in the base class, then we refer to these overridable methods are "Hook" methods.