THE FLYWEIGHT PATTERN

There are cases in programming where it seems that you need to generate a very large number of small class instances to represent data. Sometimes you can greatly reduce the number of different classes that you need to instantiate if you can recognize that the instances are fundamentally the same except for a few parameters. If you can move those variables outside the class instance and pass them in as part of a method call, the number of separate instances can be greatly reduced.

The Flyweight design pattern provides an approach for handling such classes. It refers to the instance's *intrinsic* data that makes the instance unique, and the *extrinsic* data which is passed in as arguments. The Flyweight is appropriate for small, fine-grained classes like individual characters or icons on the screen. For example, if you are drawing a series of icons on the screen in a folder window, where each represents a person or data file, it does not make sense to have an individual class instance for each of them that remembers the person's name and the icon's screen position. Typically these icons are one of a few similar images and the position where they are drawn is calculated dynamically based on the window's size in any case.

In another example in *Design Patterns*, each character in a font is represented as a single instance of a character class, but the positions where the characters are drawn on the screen are kept as external data so that there needs to be only one instance of each character, rather than one for each appearance of that character.

Discussion

Flyweights are sharable instances of a class. It might at first seem that each class is a Singleton, but in fact there might be a small number of instances, such as one for every character, or one for every icon type. The number of instances that are allocated must be decided as the class instances are needed, and this is usually accomplished with a FlyweightFactory class. This factory class usually *is* a Singleton, since it needs to keep track of whether or not a particular instance has been generated yet. It then either returns a new instance or a reference to one it has already generated.

To decide if some part of your program is a candidate for using Flyweights, consider whether it is possible to remove some data from the

class and make it extrinsic. If this makes it possible to reduce greatly the number of different class instances your program needs to maintain, this might be a case where Flyweights will help.

Example Code

Suppose we want to draw a small folder icon with a name under it for each person in a an organization. If this is a large organization, there could be a large number of such icons, but they are actually all the same graphical image. Even if we have two icons, one for “is Selected” and one for “not Selected” the number of different icons is small. In such a system, having an icon object for each person, with its own coordinates, name and selected state is a waste of resources.

Instead, we’ll create a FolderFactory that returns either the selected or the unselected folder drawing class, but does not create additional instances once one of each has been created. Since this is such a simple case, we just create them both at the outset and then return one or the other:

```
class FolderFactory
{
   Folder unSelected, Selected;
   public FolderFactory()
      {
      Color brown = new Color(0x5f5f1c);
      Selected =  new Folder(brown);
      unSelected = new Folder(Color.yellow);
      }
//-------------------------------
   public Folder getFolder(boolean isSelected)
   {
      if (isSelected)
         return Selected;
      else
         return unSelected;
   }
}
```

For cases where more instances could exist, the factory could keep a table of the ones it had already created and only create new ones if they weren’t already in the table.

The unique thing about using Flyweights, however, is that we pass the coordinates and the name to be drawn into the folder when we draw it. These coordinates are the extrinsic data that allow us to share the folder objects, and in this case create only two instances. The complete folder class

shown below simply creates a folder instance with one background color or the other and has a public Draw method that draws the folder at the point you specify.

```
class Folder extends JPanel
{
   private Color color;
   final int W = 50, H = 30;
   public Folder(Color c)
   {
      color = c;
   }
//--------------------------------
public void Draw(Graphics g, int tx, int ty, String name)
   {
    g.setColor(Color.black);            //outline
    g.drawRect(tx, ty, W, H);
    g.drawString(name, tx, ty + H+15);  //title

    g.setColor(color);                  //fill rectangle
    g.fillRect(tx+1, ty+1, W-1, H-1);

    g.setColor(Color.lightGray);        //bend line
    g.drawLine(tx+1, ty+H-5, tx+W-1, ty+H-5);

    g.setColor(Color.black);            //shadow lines
    g.drawLine(tx, ty+H+1, tx+W-1, ty+H+1);
    g.drawLine(tx+W+1, ty, tx+W+1, ty+H);

    g.setColor(Color.white);            //highlight lines
    g.drawLine(tx+1, ty+1, tx+W-1, ty+1);
    g.drawLine(tx+1, ty+1, tx+1, ty+H-1);
   }
}
```

To use a Flyweight class like this, your main program must calculate the position of each folder as part of its paint routine and then pass the coordinates to the folder instance. This is actually rather common, since you need a different layout depending on the window's dimensions, and you would not want to have to keep telling each instance where its new location is going to be. Instead, we compute it dynamically during the paint routine.

Here we note that we could have generated an array or Vector of folders at the outset and simply scan through the array to draw each folder. Such an array is not as wasteful as a series of different instances because it is actually an array of references to one of only two folder instances. However, since we want to display one folder as "selected," and we would like to be

able to change which folder is selected dynamically, we just use the FolderFactory itself to give us the correct instance each time:

```
public void paint(Graphics g)
  {
     Folder f;
     String name;

     int j = 0;      //count number in row
     int row = Top;  //start in upper left
     int x = Left;

     //go through all the names and folders
     for (int i = 0; i< names.size(); i++)
     {
     name = (String)names.elementAt(i);
     if(name.equals(selectedName))
        f = fact.getFolder(true);
     else
        f = fact.getFolder(false);
     //have that folder draw itself at this spot
     f.Draw(g, x, row, name);

     x = x + HSpace;          //change to next posn
     j++;
     if (j >= HCount)         //reset for next row
        {
        j = 0;
        row += VSpace;
        x = Left;
        }
     }
  }
```

Selecting A Folder

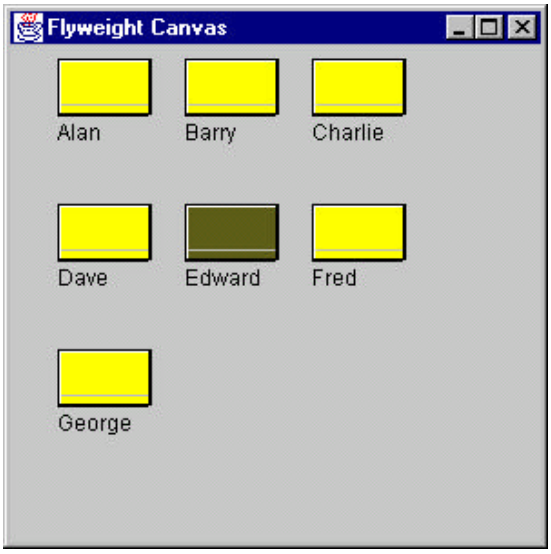
Since we have two folder instances, that we termed selected and unselected, we'd like to be able to select folders by moving the mouse over them. In the paint routine above, we simply remember the name of the folder which was selected and ask the factory to return a "selected' folder for it. Since the folders are not individual instances, we can't listen for mouse motion within each folder instance. In fact, even if we did listen within a folder, we'd have to have a way to tell the other instances to deselect themselves.

Instead, we check for mouse motion at the window level and if the mouse is found to be within a Rectangle, we make that corresponding name the selected name. This allows us to just check each name when we redraw and create a selected folder instance where it is needed:

```
public void mouseMoved(MouseEvent e)
  {
  int j = 0;      //count number in row
  int row = Top;  //start in upper left
  int x = Left;

  //go through all the names and folders
  for (int i = 0; i< names.size(); i++)
  {
  //see if this folder contains the mouse
  Rectangle r = new Rectangle(x,row,W,H);
  if (r.contains(e.getX(), e.getY()))
   {
   selectedName=(String)names.elementAt(i);
   repaint();
   }
  x = x + HSpace;          //change to next posn
  j++;
  if (j >= HCount)         //reset for next row
     {
     j = 0;
     row += VSpace;
     x = Left;
     }
  }
```

The display program for 10 named folders is shown below:



Flyweight Uses in Java

Flyweights are not frequently used at the application level in Java. They are more of a system resource management technique, used at a lower level than Java. However, it is useful to recognize that this technique exists so you can use it if you need it.

One place where we have already seen the Flyweight is in the cell renderer code we use for tables and list boxes. Usually the cell renderer is just a JLabel, but there may be two or three types of labels or renderers for different colors or fonts. However, there are far fewer renderers than there are cells in the table or list.

Some objects within the Java language could be implemented under the covers as Flyweights. For example, if there are two instances of a String constant with identical characters, they could refer to the same storage location. Similarly, it might be that two Integer or Float objects that contain the same value could be implemented as Flyweights, although they probably are not. To prove the absence of Flyweights here, just run the following code:

```
Integer five = new Integer(5);
Integer myfive = new Integer(5);
System.out.println(five == myfive);

String fred=new String("fred");
String fred1 = new String("fred");
System.out.println(fred == fred1);
```

Both cases print out “false.” However it is useful to note that you can easily determine that you are dealing with two identical instances of a Flyweight by using the “==” operator. It compares actual object references (memory addresses) rather than the “equals” operator which will probably be slower if it is implemented at all.

Sharable Objects

The *Smalltalk Companion* points out that sharable objects are much like Flyweights, although the purpose is somewhat different. When you have a very large object containing a lot of complex data, such as tables or bitmaps, you would want to minimize the number of instances of that object. Instead, in such cases, you’d return one instance to every part of the program that asked for it and avoid creating other instances.

A problem with such sharable objects occurs when one part of a program wants to change some data in a shared object. You then must decide

whether to change the object for all users, prevent any change, or create a new instance with the changed data. If you change the object for every instance, you may have to notify them that the object has changed.

Sharable objects are also useful when you are referring to large data systems outside of Java, such as databases. The Database class we developed above in the Façade pattern could be a candidate for a sharable object. We might not want a number of separate connections to the database from different program modules, preferring that only one be instantiated. However, should several modules in different threads decide to make queries simultaneously, the Database class might have to queue the queries or spawn extra connections.