THE ABSTRACT FACTORY PATTERN

The Abstract Factory pattern is one level of abstraction higher than the factory pattern. You can use this pattern when you want to return one of several related classes of objects, each of which can return several different objects on request. In other words, the Abstract Factory is a factory object that returns one of several factories.

One classic application of the abstract factory is the case where your system needs to support multiple "look-and-feel" user interfaces, such as Windows-9x, Motif or Macintosh. You tell the factory that you want your program to look like Windows and it returns a GUI factory which returns Windows-like objects. Then when you request specific objects such as buttons, check boxes and windows, the GUI factory returns Windows instances of these visual interface components.

In Java 1.2 the pluggable look-and-feel classes accomplish this at the system level so that instances of the visual interface components are returned correctly once the type of look-and-feel is selected by the program. Here we find the name of the current windowing system and then tell the PLAF abstract factory to generate the correct objects for us.

```
String laf = UIManager.getSystemLookAndFeelClassName();
try {
UIManager.setLookAndFeel(laf);
 }
catch (UnsupportedLookAndFeelException exc)
  {System.err.println("UnsupportedL&F: " + laf);}
catch (Exception exc)
  {System.err.println("Error loading " + laf);
   }
```

A GardenMaker Factory

Let's consider a simple example where you might want to use the abstract factory at the application level.

Suppose you are writing a program to plan the layout of gardens. These could be annual gardens, vegetable gardens or perennial gardens. However, no matter which kind of garden you are planning, you want to ask the same questions:

1. What are good border plants?

2. What are good center plants?
3. What plants do well in partial shade?

…and probably many other plant questions that we'll omit in this simple example.

We want a base Garden class that can answer these questions:

```
public abstract class Garden {
   public abstract Plant getCenter();
   public abstract Plant getBorder();
   public abstract Plant getShade();
}
```

where our simple Plant object just contains and returns the plant name:

```
public class Plant {
String name;
  public Plant(String pname)   {
    name = pname;     //save name
  }
  public String getName()   {
     return name;
  }
}
```

Now in a real system, each type of garden would probably consult an elaborate database of plant information. In our simple example we'll return one kind of each plant. So, for example, for the vegetable garden we simply write

```
public class VegieGarden extends Garden {
   public Plant getShade()    {
      return new Plant("Broccoli");
   }
 public Plant getCenter()  {
    return new Plant("Corn");
 }
 public Plant getBorder()  {
    return new Plant("Peas");
 }
}
```

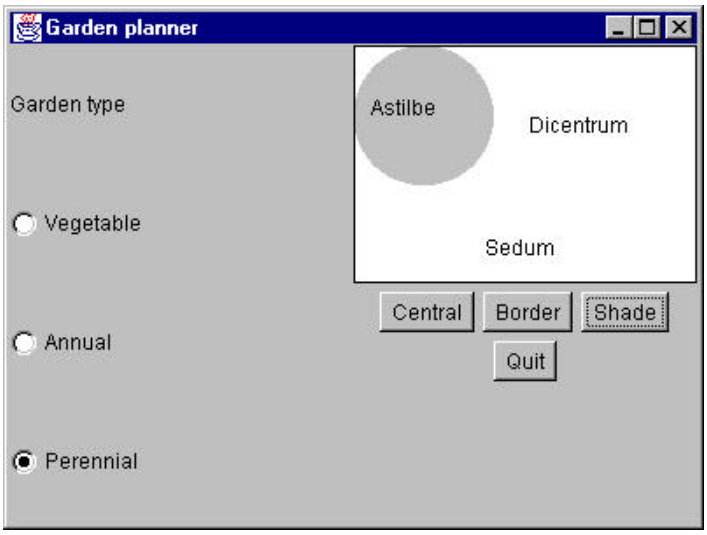
Now we have a series of Garden objects, each of which returns one of several Plant objects. We can easily construct our abstract factory to return one of these Garden objects based on the string it is given as an argument:

```
class GardenMaker
{
 //Abstract Factory which returns one of three gardens
   private Garden gd;
```

```
   public Garden getGarden(String gtype)
   {
      gd = new VegieGarden();   //default
      if(gtype.equals("Perennial"))
            gd = new PerennialGarden();
      if(gtype.equals("Annual"))
            gd = new AnnualGarden();
      return gd;
   }

}
```

This simple factory system can be used along with a more complex user interface to select the garden and begin planning it as shown below:



How the User Interface Works

This simple interface consists of two parts: the left side, that selects the garden type and the right side, that selects the plant category. When you click on one of the garden types, this actuates the MakeGarden Abstract Factory. This returns a type of garden that depends on the name of the text of the radio button caption.

```
public void itemStateChanged(ItemEvent e)
   {
      Checkbox ck = (Checkbox)e.getSource();
//get a garden type based on label of radio button
      garden = new GardenMaker().getGarden(ck.getLabel());
// Clear names of plants in display
      shadePlant=""; centerPlant=""; borderPlant = "";
```

```
      gardenPlot.repaint();             //display empty garden
   }
```

Then when a user clicks on one of the plant type buttons, the plant type is returned and the name of that plant displayed:

```
public void actionPerformed(ActionEvent e)    {
      Object obj = e.getSource();//get button type
      if(obj == Center)          //and choose plant type
         setCenter();
      if(obj == Border)
         setBorder();
      if(obj == Shade)
         setShade();
      if(obj == Quit)
         System.exit(0);
   }
   //---------------------------------
   private void setCenter()    {
      if (garden != null)
             centerPlant = garden.getCenter().getName();
      gardenPlot.repaint();
   }
   private void setBorder()    {
      if (garden != null)
             borderPlant = garden.getBorder().getName();
      gardenPlot.repaint();
   }
   private void setShade()    {
      if (garden != null)
             shadePlant = garden.getShade().getName();
      gardenPlot.repaint();
   }
```

The key to displaying the plant names is the garden plot panel, where they are drawn.

```
class GardenPanel extends Panel
 {
   public void paint (Graphics g)
   {
      //get panel size
       Dimension sz = getSize();
       //draw tree shadow
      g.setColor(Color.lightGray);
      g.fillArc( 0, 0, 80, 80,0, 360);
//draw plant names, some may be blank strings
      g.setColor(Color.black);
      g.drawRect(0,0, sz.width-1, sz.height-1);
      g.drawString(centerPlant, 100, 50);
      g.drawString( borderPlant, 75, 120);
      g.drawString(shadePlant, 10, 40);
```

```
    }
  }
}
```

You will find the complete code for Gardene.java on the example CDROM.

Consequences of Abstract Factory

One of the main purposes of the Abstract Factory is that it isolates the concrete classes that are generated. The actual class names of these classes are hidden in the factory and need not be known at the client level at all.

Because of the isolation of classes, you can change or interchange these product class families freely. Further, since you generate only one kind of concrete class, this system keeps you for inadvertently using classes from different families of products. However, it is some effort to add new class families, since you need to define new, unambiguous conditions that cause such a new family of classes to be returned.

While all of the classes that the Abstract Factory generates have the same base class, there is nothing to prevent some derived classes from having additional methods that differ from the methods of other classes. For example a BonsaiGarden class might have a Height or WateringFrequency method that is not present in other classes. This presents the same problem as occur in any derived classes-- you don't know whether you can call a class method unless you know whether the derived class is one that allows those methods. This problem has the same two solutions as in any similar case: you can either define all of the methods in the base class, even if they don't always have a actual function, or you can test to see which kind of class you have:

```
if (gard instanceof BonsaiGarden)
       int h = gard.Height();
```

Thought Questions

If you are writing a program to track investments, such as stocks, bonds, metal futures, derivatives, etc., how might you use an Abstract Factory?