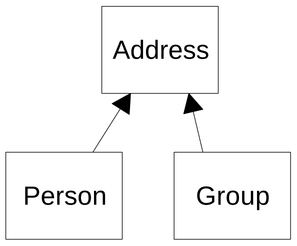
THE BUILDER PATTERN

We have already seen that the Factory Pattern returns one of several different subclasses depending on the data passed to in arguments to the creation methods. But suppose we don't want just a computing algorithm, but a whole different user interface depending on the data we need to display. A typical example might be your E-mail address book. You probably have both people and groups of people in your address book, and you would expect the display for the address book to change so that the People screen has places for first and last name, company, E-mail address and phone number.

On the other hand if you were displaying a group address page, you'd like to see the name of the group, its purpose, and a list of members and their E-mail addresses. You click on a person and get one display and on a group and get the other display. Let's assume that all E-mail addresses are kept in an object called an Address and that people and groups are derived from this base class as shown below:



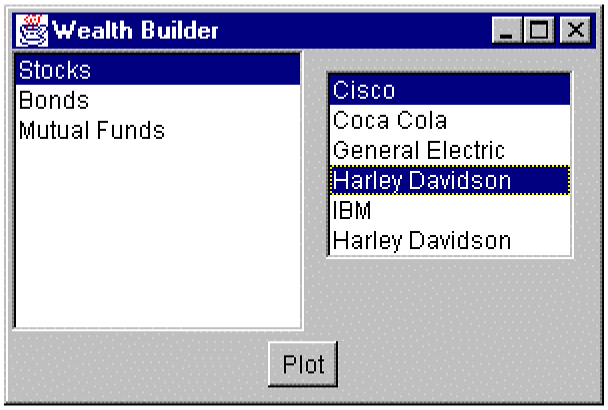
Depending on which type of Address object we click on, we'd like to see a somewhat different display of that object's properties. This is a little more than just a Factory pattern, because we aren't returning objects which are simple descendents of a base display object, but totally different user interfaces made up of different combinations of display objects. The *Builder Pattern* assembles a number of objects, such as display widgets, in various ways depending on the data. Furthermore, since Java is one of the few languages where you can cleanly separate the data from the display methods into simple objects, Java is the ideal language to implement Builder patterns.

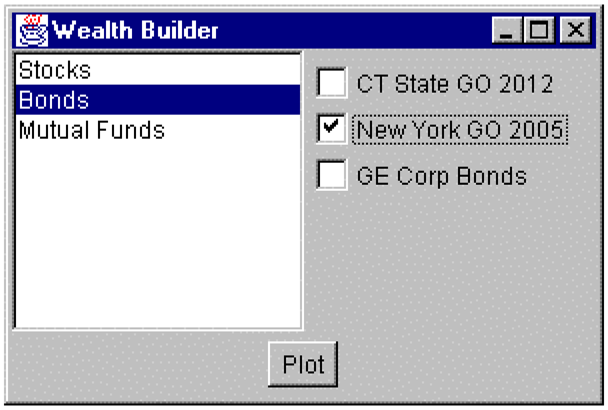
An Investment Tracker

Let’s consider a somewhat simpler case where it would be useful to have a class build our UI for us. Suppose we are going to write a program to keep track of the performance of our investments. We might have stocks, bonds and mutual funds, and we’d like to display a list of our holdings in each category so we can select one or more of the investments and plot their comparative performance.

Even though we can’t predict in advance how many of each kind of investment we might own at any given time, we’d like to have a display that is easy to use for either a large number of funds (such as stocks) or a small number of funds (such as mutual funds). In each case, we want some sort of a multiple-choice display so that we can select one or more funds to plot. If there is a large number of funds, we’ll use a multi-choice list box and if there are 3 or fewer funds, we’ll use a set of check boxes. We want our Builder class to generate an interface that depends on the number of items to be displayed, and yet have the same methods for returning the results.

Our displays are shown below. The first display contains a large number of stocks and the second a small number of bonds.





Now, let’s consider how we can build the interface to carry out this variable *multiChoice* abstract class that defines the methods we need to implement:

```
abstract class multiChoice
{
   //This is the abstract base class
   //that the listbox and checkbox choice panels
   //are derived from
   Vector choices;    //array of labels
//---------------------------------------------
   public multiChoice(Vector choiceList)
   {
     choices = choiceList;    //save list
   }
   //to be implemented in derived classes
 abstract public Panel getUI();  //return a Panel of components
 abstract public String[] getSelected(); //get list of items
 abstract public void clearAll();       //clear selections
}
```

The *getUI* method returns a Panel container with a multiple-choice display. The two displays we’re using here -- a checkbox panel or a list box panel -- are derived from this abstract class:

```
class listboxChoice extends multiChoice
```

or

```
class checkBoxChoice extends multiChoice
```

Then we create a simple Factory class that decides which of these two classes to return:

```
class choiceFactory
{
   multiChoice ui;
   //This class returns a Panel containing
   //a set of choices displayed by one of
   //several UI methods.
   public multiChoice getChoiceUI(Vector choices)
   {
      if(choices.size() <=3)
        //return a panel of checkboxes
        ui = new checkBoxChoice(choices);
      else
        //return a multi-select list box panel
        ui = new listboxChoice(choices);
      return ui;
   }
}
```

In the language of *Design Patterns*, this factory class is called the Director, and the actual classes derived from *multiChoice* are each Builders.

Calling the Builders

Since we're going to need one or more builders, we might have called our main class Architect or Contractor, but since we're dealing with lists of investments, we'll just call it WealthBuilder. In this main class, we create the user interface, consisting of a BorderLayout with the center divided into a 1 x 2 GridLayout. The left part contains our list of investment types and the right an empty panel that we'll fill depending on which kind of investments are selected.

```
public wealthBuilder()
    {
       super("Wealth Builder");      //frame title bar
       setGUI();                     //set up display
       buildStockLists();            //create stock lists
        choiceFactory cfact;        //the factory
    }
    //-----------------------------------
    private void setGUI()
    {
       setLayout(new BorderLayout());
       Panel p = new Panel();
       add("Center", p);
  //center contains left and right panels
       p.setLayout(new GridLayout(1,2));

  //left is list of stocks
      stockList= new List(10);
      stockList.addItemListener(this);
       p.add(stockList);
       stockList.add("Stocks");
```

```
        stockList.add("Bonds");
        stockList.add("Mutual Funds");
        stockList.addItemListener(this);

    //Plot button along bottom of display
        Panel p1 = new Panel();
        p1.setBackground(Color.lightGray);
        add("South", p1);
        Plot = new Button("Plot");
        Plot.setEnabled(false);    //disabled until stock picked
        Plot.addActionListener(this);
        p1.add(Plot);

   //right is empty at first
        choicePanel = new Panel();
        choicePanel.setBackground(Color.lightGray);
        p.add(choicePanel);

        w = new Winder();  //intercepts WindowClosing
        addWindowListener(w);
        setBounds(100, 100, 300, 200);
        setVisible(true);
     }
```

In this simple pro gram, we keep our three lists of investments in three Vectors called Stocks, Bonds and Mutuals. We load them with arbitrary values as part of program initialization:

```
       Mutuals = new Vector();
        Mutuals.addElement("Fidelity Magellan");
        Mutuals.addElement("T Rowe Price");
        Mutuals.addElement("Vanguard PrimeCap");
        Mutuals.addElement("Lindner Fund");
```

and so forth. In a real system, we'd probably read them in from a file or database. Then, when the user clicks on one of the three investment types in the left list box, we pass the equivalent vector to our Factory, which returns one of the builders:

```
private void stockList_Click()
{
   Vector v = null;
   int index = stockList.getSelectedIndex();
   choicePanel.removeAll();  //remove previous ui panel

   //this just switches among 3 different Vectors
   //and passes the one you select to the Builder pattern
   switch(index)
   {
   case 0:
      v = Stocks;  break;
   case 1:
      v = Bonds;   break;
   case 2:
      v = Mutuals;
   }
   mchoice = cfact.getChoiceUI(v);      //get one of the UIs
   choicePanel.add(mchoice.getUI());    //insert in right panel
   choicePanel.validate();              //re-layout and display
   Plot.setEnabled(true);               //allow plots
 }
```

We do save the multiChoice panel the factory creates in the **mchoice** variable so we can pass it to the Plot dialog.

The List Box Builder

The simpler of the two builders is the List box builder. It returns a panel containing a list box showing the list of investments.

```
class listboxChoice extends multiChoice
{
   List list;       //investment list goes here
//---------------------------------------------
   public listboxChoice(Vector choices)
   {
      super(choices);
   }
//---------------------------------------------
   public Panel getUI()
   {
      //create a panel containing a list box
      Panel p = new Panel();
      list = new List(choices.size());  //list box
      list.setMultipleMode(true);       //multiple
      p.add(list);
//add investments into list box
      for (int i=0; i< choices.size(); i++)
         list.addItem((String)choices.elementAt(i));
     return p;             //return the panel
  }
```

The other important method is the *getSelected* method that returns a String array of the investments the user selects:

```
public String[] getSelected()
   {
      int count =0;
      //count the selected listbox lines
      for (int i=0; i < list.getItemCount(); i++ )
         {
```

```
         if (list.isIndexSelected(i))
            count++;
         }
      //create a string array big enough for those selected
      String[] slist = new String[count];

      //copy list elements into string array
      int j = 0;
      for (int i=0; i < list.getItemCount(); i++ )
         {
         if (list.isIndexSelected(i))
            slist[j++] = list.getItem(i);
         }
      return(slist);
   }
```

The Checkbox Builder

The Checkbox builder is even simpler. Here we need to find out how many elements are to be displayed and create a horizontal grid of that many divisions. Then we insert a check box in each grid line:

```
public checkBoxChoice(Vector choices)
   {
      super(choices);
      count = 0;
      p = new Panel();
   }
//----------------------------------------------
   public Panel getUI()
   {
      String s;

      //create a grid layout 1 column by n rows
      p.setLayout(new GridLayout(choices.size(), 1));

      //and add labeled check boxes to it
      for (int i=0; i< choices.size(); i++)
         {
         s =(String)choices.elementAt(i);
         p.add(new Checkbox(s));
         count++;
         }
      return p;
   }
```

The *getSelected* method is analogous to the one we showed above, and is included in the example code on the CDROM.

Consequences of the Builder Pattern

1. A Builder lets you vary the internal representation of the product it builds. It also hides the details of how the product is assembled.
2. Each specific builder is independent of the others and of the rest of the program. This improves modularity and makes the addition of other builders relatively simple.
3. Because each builder constructs the final product step-by-step, depending on the data, you have more control over each final product that a Builder constructs.

A Builder pattern is somewhat like an Abstract Factory pattern in that both return classes made up of a number of methods and objects. The main difference is that while the Abstract Factory returns a family of related classes, the Builder constructs a complex object step by step depending on the data presented to it.

Thought Questions

1. Some word-processing and graphics programs construct menus dynamically based on the context of the data being displayed. How could you use a Builder effectively here?
2. Not all Builders must construct visual objects. What might you use a Builder to construct in the personal finance industry? Suppose you were scoring a track meet, made up of 5-6 different events? Can you use a Builder there?