# CMSC 433 – Programming Language Technologies and Paradigms Spring 2007

Builder Pattern

Mar. 13, 2007



## Revisit the Factory Pattern

- Factory Pattern returns one of several different subclasses depending on the data passed via parameters to the creation method(s).
- Suppose we aren't interested in returning objects that are simple descendents of a base object, but are assembled from different combinations of (unrelated) objects.



## What is the Builder Pattern?

- The Builder Pattern assembles and returns a number of objects in various ways depending on the data passed via parameters to the creation method(s).



## An Example

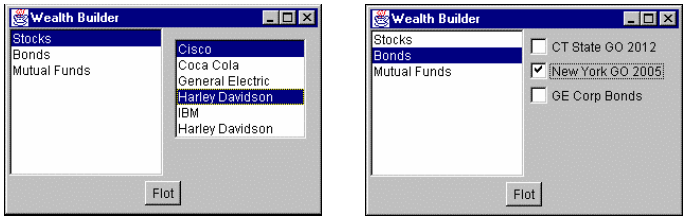
- Lets design a class that will build a User Interface for us.
- Requirements: write a program to keep track of the performance of our investments. We might have stocks, bonds and mutual funds, and we'd like to display a list of our holdings in each category so we can select one or more of the investments and plot their comparative performance.
  - Wealth Builder

## Example (contd…)

- We cannot 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 kind 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.



## An Example Final Display






## Let's See Some Code!

- start with a 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
}
```

## Lets Plan for a Second

- 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

## Throw in a Factory for Variety!

- 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;
   }
}
```

**Fine Print: this factory class is called the Director, and the actual classes derived from *multiChoice* are each Builders.**

## Main Class

- In the main class
  - 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.
- In the main class code
  - choiceFactory cfact;  //the factory



## Invoking the Factory

- 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 pa
    choicePanel.validate();              //re-layout and disp
    Plot.setEnabled(true);               //allow plots
}
```

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

## Finally, the Builders!

- the List box builder 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
  }
```

## getSelected()

```
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 *getSelected* method returns a String array of the investments the user selects.**



## Summary

- A Builder lets you vary the internal representation of the product it builds.
  - It also hides the details of how the product is assembled.
- 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.
- 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 a Factory pattern in that:
  - The main difference is that while the Factory returns a family of related objects, the Builder constructs a complex object step by step depending on the data presented to it.