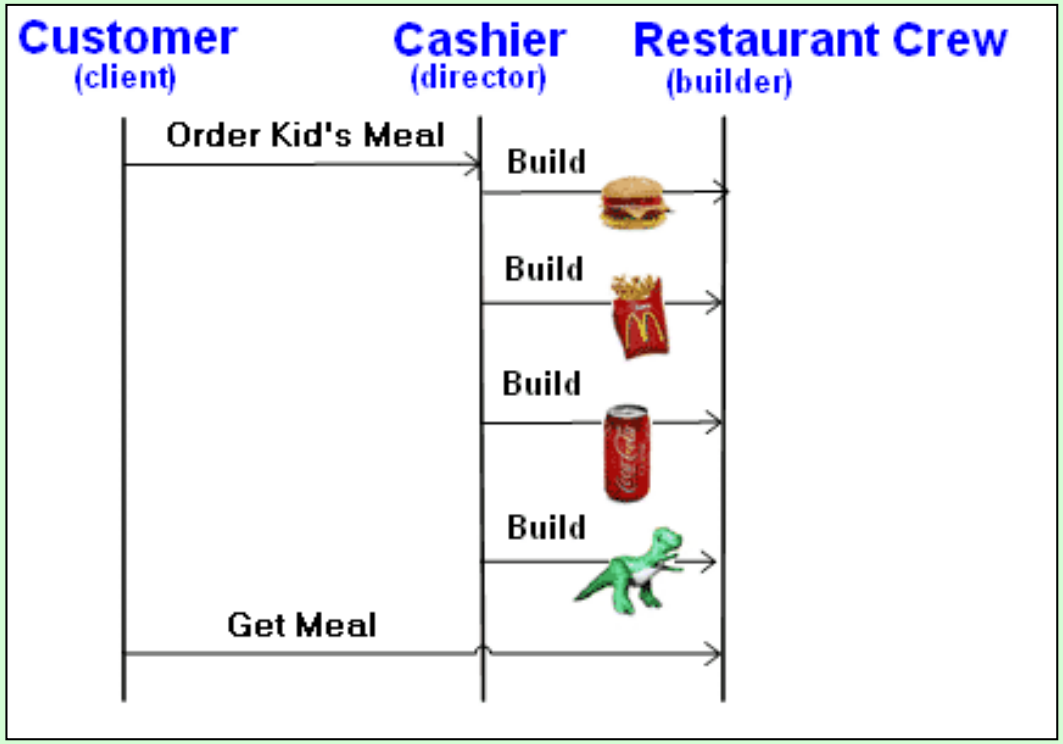
# Object Orientation with Design Patterns

## Builder Pattern

## Structural Patterns (Decorator)

# Builder Pattern

- **Intent :**
  Separate the **construction of a complex object** from it's representation so that the **same construction process** can create different representations.




http://sourcemaking.com/design_patterns/builder

# Builder Pattern

- In this lecture we will consider how to use the Builder pattern to **construct objects** from **components**.
- We have already seen that the **Factory pattern** returns one of **several different subclasses**, depending on the data passed in arguments to the creation methods.

- But suppose we don't want just a computing algorithm, but rather a **whole different user interface**, depending **on the data we need to display**.

# Builder Pattern

- An example of this might be your **email address book**.
- You probably have both **people** and **groups of people** in your address book, and you would expect the display of the address book to change so that the **People screen** has places for the **first and last names, company name, email address, and telephone 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 its members and their email addresses**.

# Builder Pattern

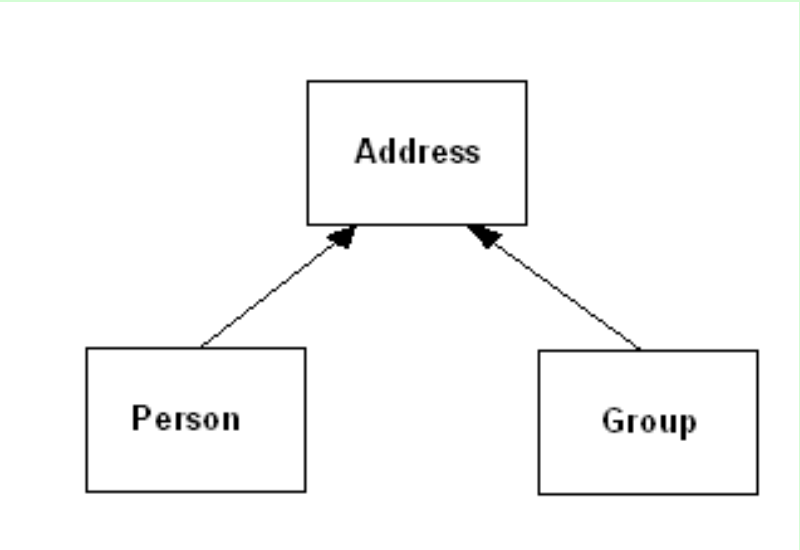
- eMail – People
  - First and Last Name
  - Company Name
  - Email address
  - Telephone

- Group Address Page
  - Name of group
  - Purpose
  - List of it's members
  - Email addresses

# Builder Pattern

- You click on a **persons name** and **get one display** and **click on a groups name** and get **another type** of display.

- So if all email addresses were kept in an address object we might have something like this:



# Builder Pattern

- So depending on what **type of Address object** we click we would like to **see a somewhat different display** of that objects properties.

- **This is a little more than just a factory pattern because the objects returned are not just simple descendants of a base display object, but rather 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 in the data.

- Furthermore, since Java is one of the few languages with which you **can cleanly separate the data from the display methods** into simple objects, **Java** is the **ideal language** to implement the Builder pattern.

# Investment Tracker

- Let's consider a somewhat simpler case in which it would be useful to have a class build our GUI for us.

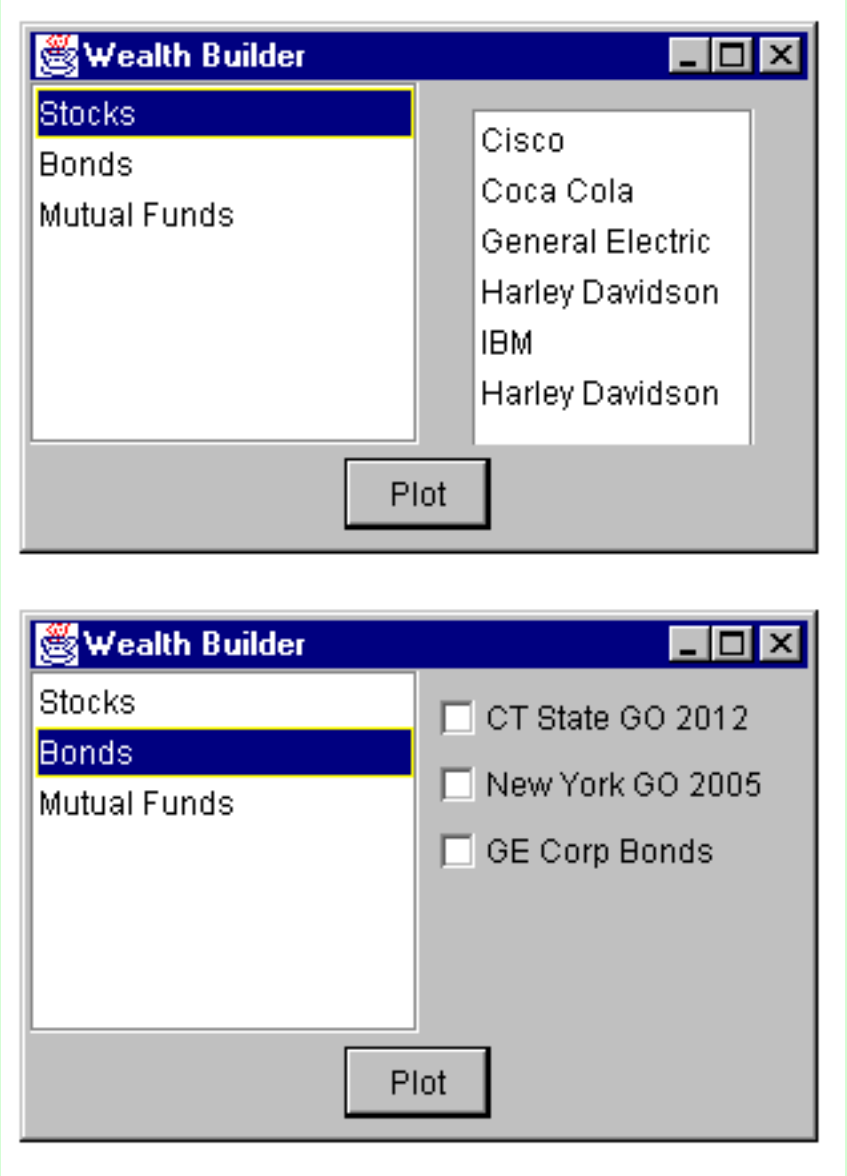
- Suppose that we want to write a program to **keep track of the performance of our investments**, for example, **stocks, bonds**, and **mutual funds**. We want to display a list of our holdings in each category so that we can select one or more of the investments and plot their comparative performances.

- Even though we can't predict in advance how many of each kind of investment we might own at any given time, we want 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).

# Investment Tracker

- In each case we want some sort of **multiple choice selection** so that we can select one or more funds to plot.
- If there are a large number of funds we'll use a **multichoice list box**; if there are three 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.
- The two different displays are shown on the next slide.

# Investment Tracker



# Investment Tracker

- Now lets consider how to build the interface to carry out this variable display.

- Well start with a MultiChoice ***abstract*** class that defines the methods that we will need to implement.

# Investment Tracker

```
import java.util.*;
import java.awt.*;
import javax.swing.*;
public abstract class multiChoice {
    //This is the abstract base class
    //that the listbox and checkbox choice panels
    //are derived from
    protected Vector choices;     //array of labels

    public multiChoice(Vector choiceList) {
        choices = choiceList;     //save list
    }
    //to be implemented in derived classes


    //return a Panel of components
    abstract public JPanel getUI();
    //get a list those selected
    abstract public String[] getSelected();
    //clear all the selected items
    abstract public void clearAll();
}
```

# Investment Tracker

- The ***getUI* method** returns a Panel that has **a multiple choice display.** The two displays we're using here -- a **checkbox panel and** a **listbox panel** -- are derived from this abstract class.

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

# Investment Tracker

```
import java.util.*;
public 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 listbox panel
               ui = new listboxChoice(choices);
          return ui;
     }
}
```

# Investment Tracker

- In the language of **Design Patterns** this **factory** is called a **Director,** and each actual class derived from **multiChoice** is a **Builder.**

- Since we are going to need more builders, we might call our main class Architect or Contractor. However since were dealing with lists of investments we'll 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 of the grid contains our list of investment types and the right part of the grid is an empty panel that we'll fill depending on the kinds of investments selected.

# Investment Tracker

```
import java .awt.*;
import java.awt.event.*;
import java.util.*;
import javax.swing.*;
import javax.swing.event.*;

//This program illustrates the

public class wealthBuilder extends JxFrame
  implements ListSelectionListener, ActionListener {

    private JawtList stockList;          //list of funds
    private JButton Plot;                //plot command button
    private JPanel choicePanel;          //right panel
    private multiChoice mchoice;         //ui for right panel
    private Vector Bonds, Stocks, Mutuals;   //3 lists of investments
    private choiceFactory cfact;         //the factory

    public wealthBuilder()
    {
        super("Wealth Builder");      //frame title bar
        setGUI();                     //set up display
        buildStockLists();            //create stock lists
        cfact = new choiceFactory(); //create builder-factory
    }
```

# Investment Tracker

- In this simple program we will keep our list of investments in a Vector. We can load each Vector with arbitrary values as part of the program initialization.

```
private void buildStockLists() {
    //arbitrary list of stock, bond and fund holdings
    Bonds = new Vector();
    Bonds.addElement("CT State GO 2012");
    Bonds.addElement("New York GO 2005");
    Bonds.addElement("GE Corp Bonds");

    Stocks = new Vector();
    Stocks.addElement("Cisco");
    Stocks.addElement("Coca Cola");
    Stocks.addElement("General Electric");
    Stocks.addElement("Harley Davidson");
    Stocks.addElement("IBM");
    Stocks.addElement("Harley Davidson");

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

}
```

# Investment Tracker

- When the user clicks on one of the three investment types in the left listbox, **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 between 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
    choicePanel.repaint ();
    Plot.setEnabled(true);            //allow plots
}
```

# Investment Tracker

- The simpler of the two builders is the listbox builder. The getUI method returns a panel containing a list box showing the list of investments.

# Investment Tracker

```
import java.util.*;
import java.awt.*;
import javax.swing.*;

public class listboxChoice extends multiChoice {
    JawtList list;

    public listboxChoice(Vector choices) {
        super(choices);
    }


    public JPanel  getUI() {

        //create a panel containing a list box
        JPanel p = new JPanel();
        list = new JawtList(choices.size());
        list.setMultipleMode(true);
        p.add(list);
        for (int i=0; i< choices.size(); i++)
            list.add((String)choices.elementAt(i));
        return p;
    }
```

# Investment Tracker

- The other important method in the listBoxChoice class is the **getSelected method**, which returns a String array of investments that the user selects.

```
public String[] getSelected() {
    String[] slist = list.getSelectedItems ();
    return(slist);
}
```

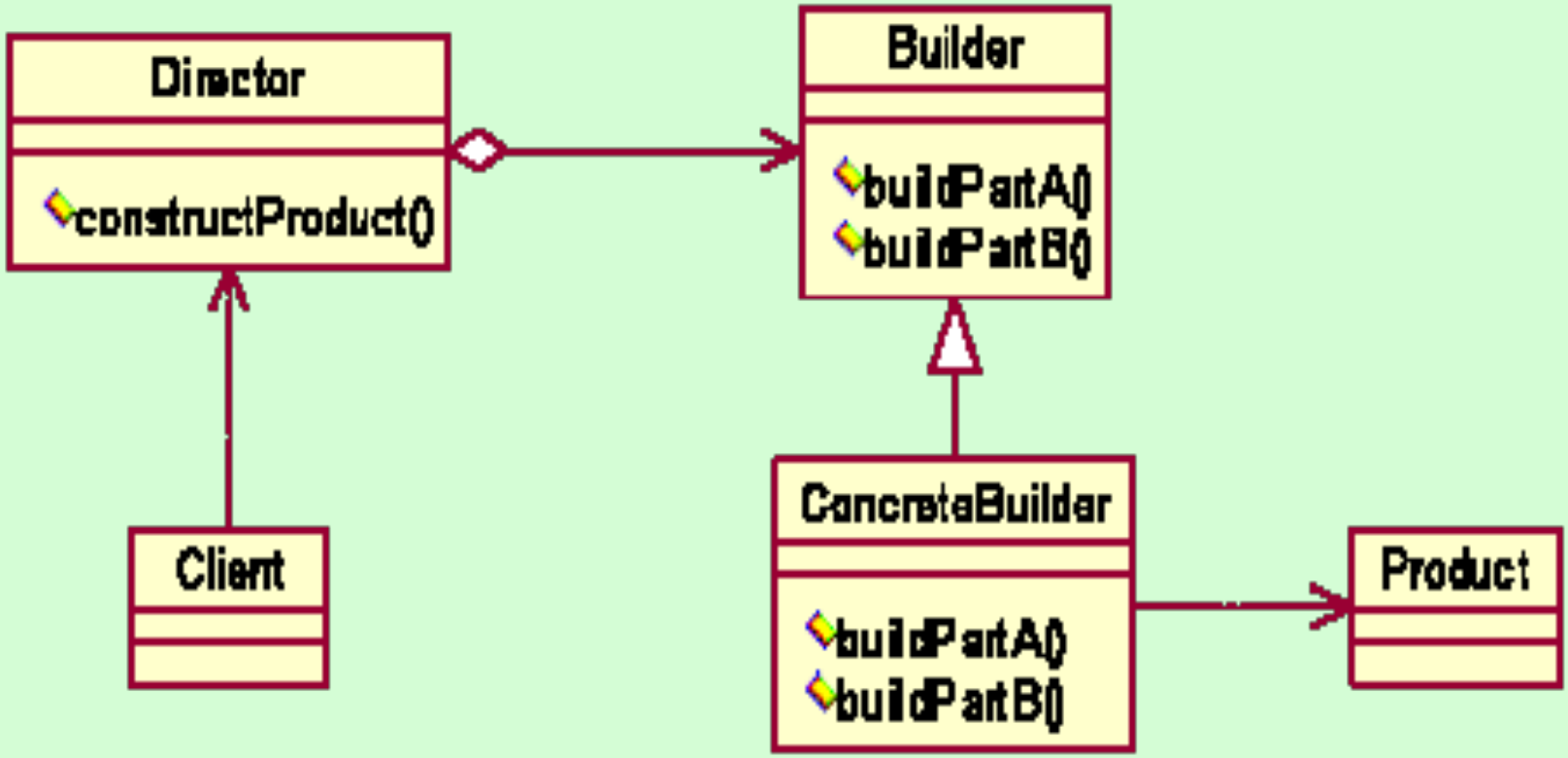
# Investment Tracker

- The checkBox Builder is more difficult.
  Here we need to find out how many elements are to be displayed and then create a horizontal grid of that many divisions.  Then we insert a checkbox into each grid line.

# Investment Tracker

```
import java.awt.*;
import java.util.*;
import javax.swing.*;


public class checkBoxChoice extends multiChoice {
    //This derived class creates
    //vertical grid of checkboxes
    int count;              //number of checkboxes
    JPanel p;               //contained in here

    public checkBoxChoice(Vector choices) {
        super(choices);
        count = 0;
        p = new JPanel();
    }

    public JPanel 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 JCheckBox(s));
            count++;
        }
        return p;
    }
```

# Structure of Builder Pattern



# Participants

**Builder: (multiChoice)**

- Specify an abstract interface for creating parts of Product object e.g.

**ConcreteBuilder: (listBoxChoice/checkBoxChoice)**

- Constructs and assembles parts, implement Builder interface (or the parts of the interface that's in it's product)
- Defines and keeps track of the representation it creates (usually some reference in the class which is returned on completion e.g. Panel p in ListBox Builder)
- Provides an interface for retrieving the product e.g. getGui()

# Participants

**Director (choiceFactory)**

- Constructs an object using the Builder interface
  (In example if <=3  use checkboxchoice else use litboxchoice

**Product**

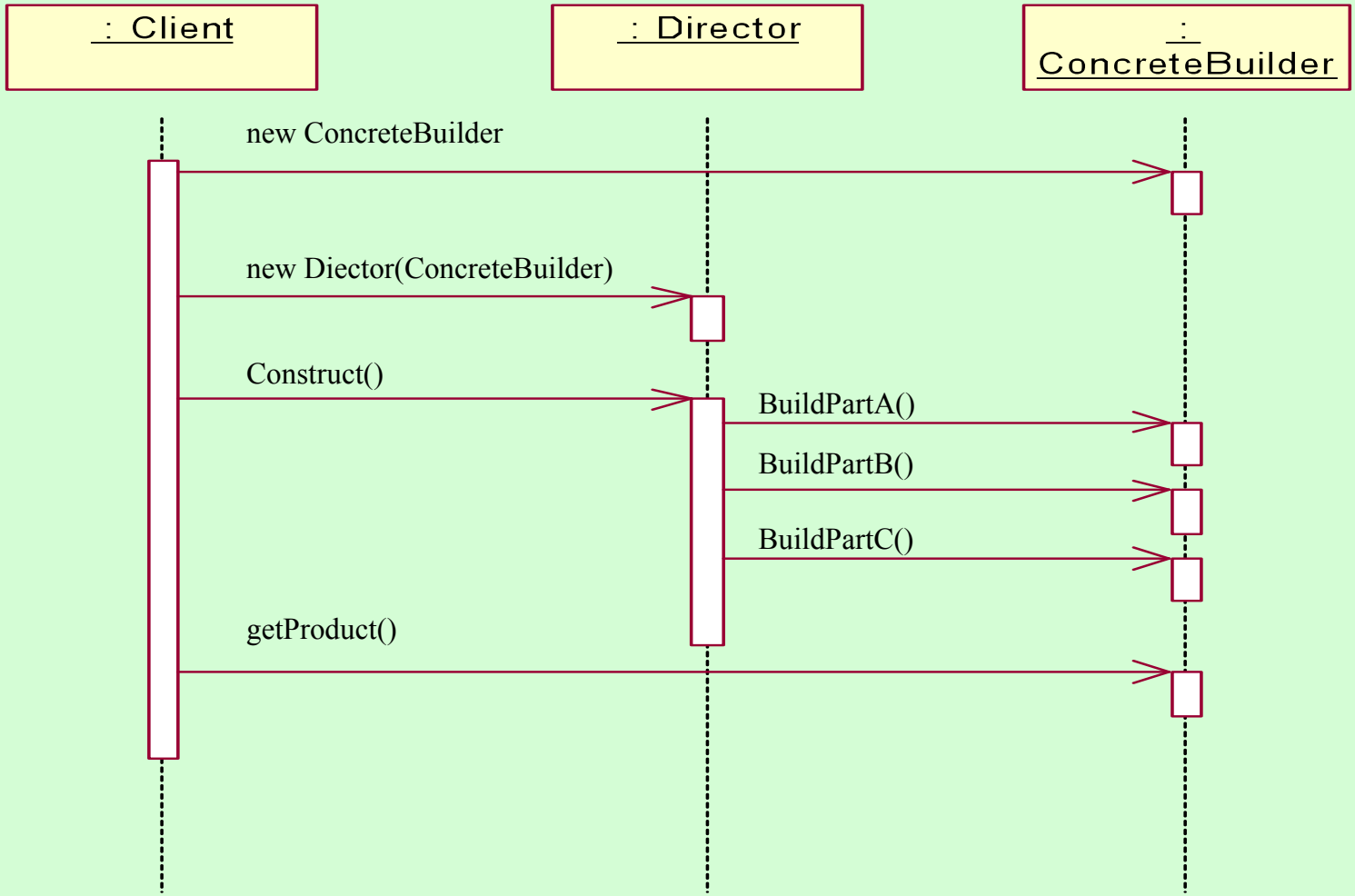
- Complex object under construction. ConcreteBuilder, e.g. ListBoxChoice, builds the products internal representation and defines the process by which it's assembled

- Includes classes that define the constituent parts including interfaces for assembling the parts into the final result

# Collaborations

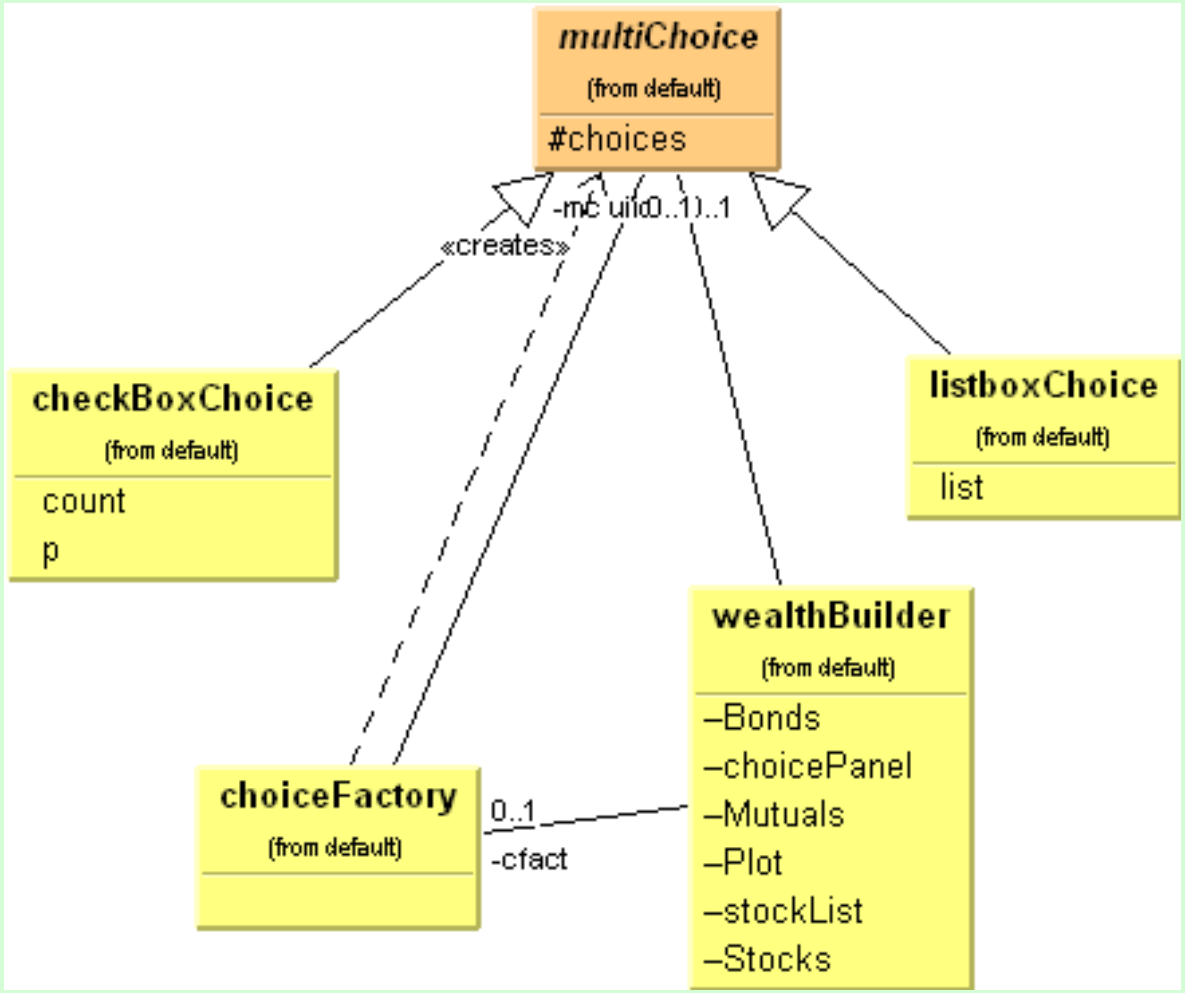
- The **client creates a Director object** and configures it with the desired object
- **Director** notifies **the builder** whenever a part of the product should be built, e.g. the Factory for the transaction GUI is a Director
- **Builder** handles **Director request** and adds parts to the product
- Client retrieves the product from the builder e.g. getGUI()

# Collaborations



# Investment Tracker

- The getSelected method is analogous to the method shown previously and is included in the example code on the student share.



# Consequences of Builder Pattern

- Using the Builder pattern has the following consequences:

- A Builder pattern lets you **vary the internal representation of the product that build it.** It also **hides the details of how the product is assembled.**

- Each **specific builder** is **independent** of any others and of the rest of the program.
  This **improves modularity** and makes the **addition** of other **builders** relatively **simple.**

- The Builder supplies a method to retrieve the final product, eg getGUI

# Consequences of Builder Pattern

- 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 **abstact factory returns a family of classes**, the **builder pattern constructs a complex object** depending on the data given.

- The builder also gives you **more control** of the building process as it's step-by-step

- Also the **Abstract Factory** will **return object(s) straight away** whereas the Builder **supplies a method** to retrieve the **final product**.

# Structural Patterns

- A structural pattern describes how **classes can be combined together to form larger structures.**
- A *class pattern* describes how **inheritance can be used to provide more useful program interfaces.**
- An *object pattern* describes how **objects can be composed into larger structures using object composition** or by including objects within objects (aggregation).

# Decorator Pattern

- **Intent:**
  Attach **additional responsibilities to an object dynamically**. Decorators provide a **flexible alternative** to **subclassing** for extending functionality.

- Also known as a **Wrapper class**.

# Decorator Pattern - Applicability

**Use Decorator:**

- To **add responsibilities to individual objects dynamically and transparently**, that is, **without affecting** other objects.
- For **responsibilities** that can be **withdrawn**.
- When extension by **subclassing** is **impractical**.
  Sometimes a large number of independent extensions are possible and would **produce an explosion of subclasses to support every combination**.
  Or a class definition may be hidden or otherwise unavailable for subclassing.

# Decorator Pattern - Participants

- **Component**

  Defines the **interface for objects** that can have responsibilities added to them dynamically.

- **ConcreteComponent**

  Defines an **object to which additional responsibilities** can be attached.

# Decorator Pattern

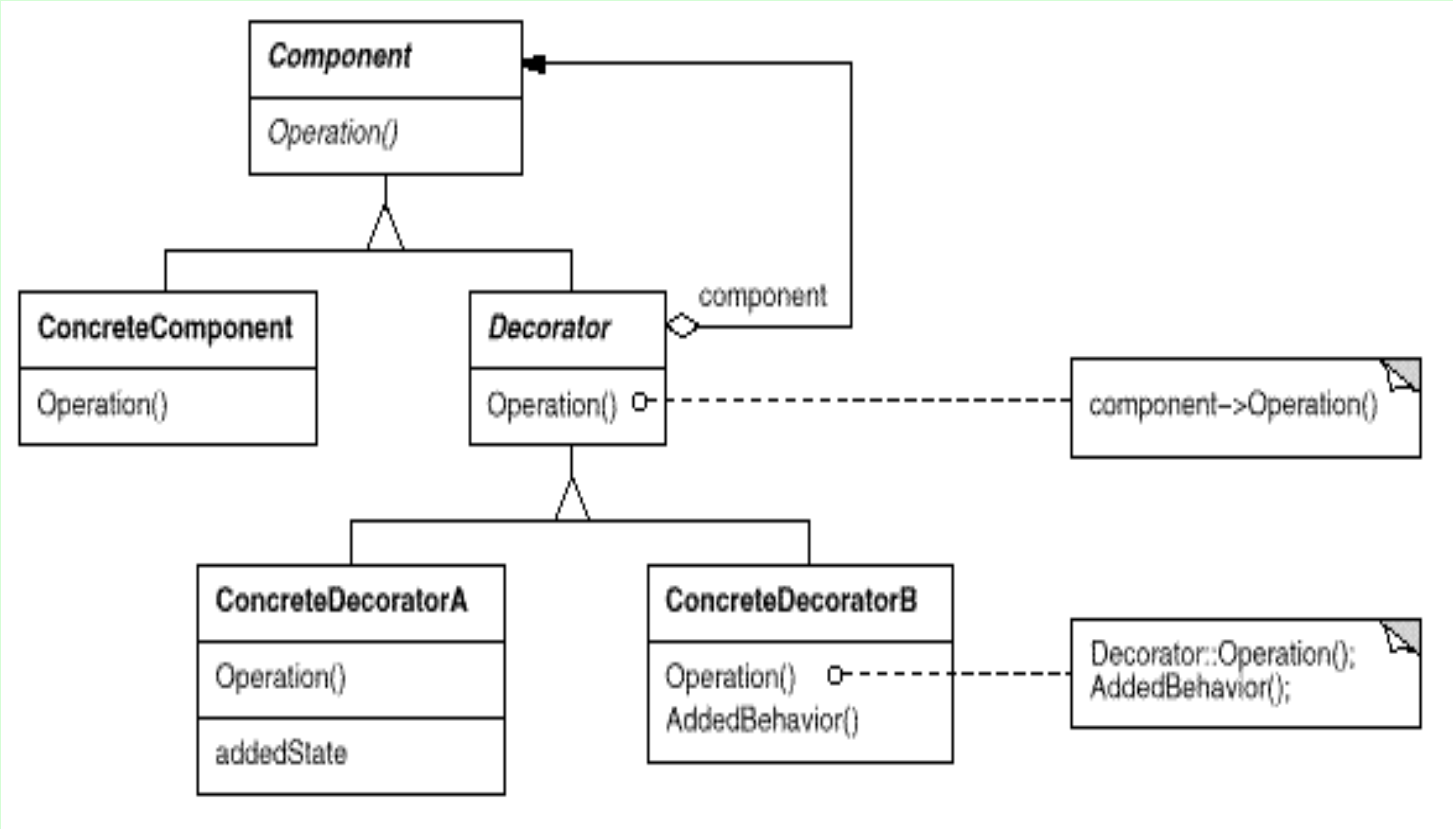
- **Decorator**

  Maintains a reference to a Component object and defines an interface that conforms to Component's interface.

- **ConcreteDecorator**

  **Adds responsibilities** to the component.

# Decorator Pattern



# Decorator Collaborations

- Decorator **forwards requests** to its Component object. It may optionally perform additional operations before and after forwarding the request.

# Decorator Pattern

- The ***Decorator Pattern*** provides us with a way to **modify the behavior of individual objects** without having to create  a new derived class (subclass).

- Suppose we have a program that uses **eight objects**, but three of them need an **additional features**.  You could create a subclass for each of these objects, and in many cases this would be an acceptable solution.

- However, if each of the three objects requires **different features**, you would have to **create three separate subclasses**.  Furthermore if one of the classes has features of both of the other classes , you begin to create complexity that is both confusing and unnecessary.

# Decorator Pattern

- For example suppose we wanted to draw a **special border** around some of the buttons in a toolbar. If we created a **new derived button class**, this means that **all of the buttons** in this new class would have the **same border**, when this might not be our intent.

- Instead we can use a **Decorator Class** which ***decorates the buttons***. Then we derive any number of specific decorators from the main decorator class, each of which performs a specific kind of decoration.

- In order to **decorate a button** there has to be an **object derived (subclassed) from an object** in the visual environment so that it can receive paint messages.

# Decorator Pattern

- **The decorator is a graphical object, but it contains the object that it is decorating.**

- It might intercept some graphical method calls and perform some additional computation, and then it might pass them on to the underlying object that it is decorating.

# Decorating a CoolButton

- Recent Windows applications such as Internet Explorer and Netscape Navigator have a row of flat, unbordered buttons that highlight themselves with outline borders when you move the mouse over them.

- In Java there is no analogous button behavior, but we can get that behavior by ***decorating* a JButton**.

- In this case we decorate it by **drawing plain grey lines** over the **button borders**, effectively erasing them.

- The next examples illustrates how this can be achieved.

# Decorating a CoolButton

- Decorators should be **derived from** some **general visual component class** and then **every message** for the actual button should be **forwarded from the decorator**.

- In Java we can derive our decorator class from the **JComponent class**. Then, since JComponent itself behaves as a container, it will **forward all method calls** to the components that it contains.

- In this case we simply **add the button** to the **JComponent**, and that button will receive all of the GUI method calls that the JComponent receives.

# Decorating a CoolButton

```
public class Decorator extends JComponent {
    public Decorator(JComponent c) {
        setLayout(new BorderLayout());
        add("Center", c);
    }
}
```

- In order to create a CoolButton  all we really need to do is draw the button as usual from the base class and then draw grey lines around the border to remove the button highlighting.

# Decorating a CoolButton

```
public class CoolDecorator extends Decorator {
    boolean mouse_over;     //true when mose over button
    JComponent thisComp;

    public CoolDecorator(JComponent c) {
        super(c);
        mouse_over = false;
        thisComp = this;        //save this component
        //catch mouse movements in inner class
        c.addMouseListener(new MouseAdapter() {
                //set flag when mouse over
                public void mouseEntered(MouseEvent e) {
                        mouse_over = true;
                        thisComp.repaint();
                }
               //clear flag when mouse not over
               public void mouseExited(MouseEvent e) {
                        mouse_over = false;
                        thisComp.repaint();
                }
            });
    }
```

# Decorating a CoolButton

```
    //paint the button
    public void paint(Graphics g) {
        super.paint(g);       //first draw the parent button
        //if the mouse is not over the button
        //erase the borders
        if (! mouse_over) {
            Dimension d = super.getSize();
            g.setColor(Color.lightGray);
            g.drawRect(0, 0, d.width-1, d.height-1);
            g.drawLine(d.width-2, 0, d.width-2, d.height-1);
            g.drawLine(0, d.height-2, d.width-2, d.height-2);
        }
    }
}
```

- There are two important features of the CoolDecorator
  - The **constructor which adds mouse adapters** so it can trap mouse movement.
  - The **overloaded paint** so that it can **erase the button borders**

# Using a Decorator

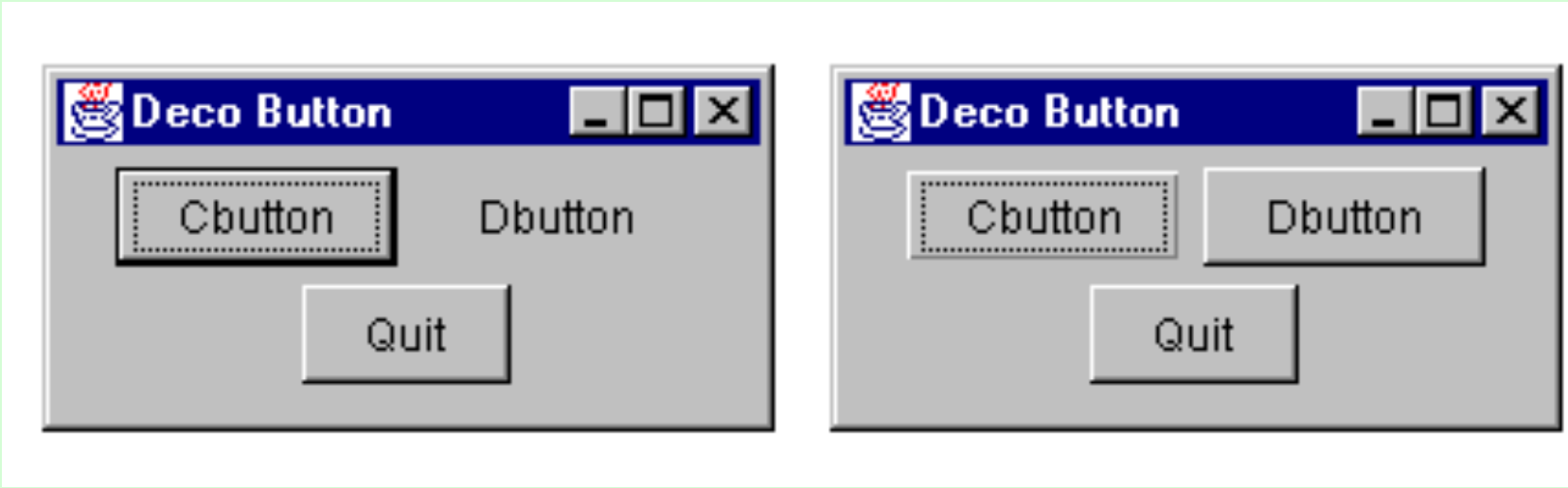
- Now that we have written a CoolDecorator class, **how do we use it?** We simply **create an instance of the CoolDecorator** and pass it the button that it to be decorated.
  We can do all of this in the **call to the constructor**.
  Lets consider a simple program that has two CoolButtons and an ordinary JButton.

- The next slide shows a code segment that illustrates the use of the CoolDecorator class.

# Using a Decorator

```
super ("Deco Button");
JPanel jp = new JPanel();

getContentPane().add(jp);

jp.add( new CoolDecorator(
        CButton = new JButton("Cbutton")));

jp.add( new CoolDecorator(
        DButton = new JButton("Dbutton")));


jp.add(Quit = new JButton("Quit"));
Quit.addActionListener(this);
setSize(new Dimension(200,100));

setVisible(true);
Quit.requestFocus();
```

# Using a Decorator




- In the first screen shot the mouse has not moved over the any of the CoolButtons.
- In the second screen shot the mouse has moved over the Dbutton and the border is drawn.

# Nonvisual Decorators

- Decorators are not **limited to objects** that enhance visual classes. You can add or modify the **methods** of any object in a similar fashion.

- Consider a program to **filter an email stream** to make it more readable. The user of the email program types a message in lower case. The sentences can be made **more readable** by making the **first letter in each sentence upper case**.

- We can achieve this by **decorating a FileInputStream** to read the data a transform it into a more readable form.

# Nonvisual Decorators

- The FileFilter class simply extends the FileInputStream class and overloads its read method so that we can intercept the text and capitalize the first letter of each sentence.

```
public class FileFilter extends FilterInputStream {

    static int MAX=1000;

    public FileFilter(InputStream f) {
        super(f);
    }
```

- The next slide show the overloaded read method of the FileFilter class.

# Using a Decorator

```
public String readLine() {
    String s;
    int length = 0;

    byte b[] = new byte[MAX];
    try {
        length = super.read(b);
        s = new String(b, 0, length);
    } catch (IOException e) {
        s = "";
    }

    StringBuffer buf = new StringBuffer(s);
    boolean punctFound = true;

    for (int i=0; i < length; i++) {
        char ch = buf.charAt (i);
        if (punctFound && (ch != ' ') ) {
            ch = Character.toUpperCase (ch);
            buf.setCharAt (i, ch);
            punctFound = false;
        }
        if (ch == '.')
            punctFound = true;
    }
    s = buf.toString ();
    return s;
}
```

# Decorator Consequences

- The decorator pattern provides a **more flexible** way to **add responsibilities to a class** than using inheritance, since it can add these responsibilities to **selected instances** of the class.
- It also allows the programmer to **customize a class** without subclassing it.
- The full source code for the examples given in this lecture are available on the student share folder and should be studied.
- You can combine decorators by nesting one decorator inside another, e.g.