Object Orientation with Design Patterns

Lecture 4: Builder Pattern

Builder Pattern

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

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**.

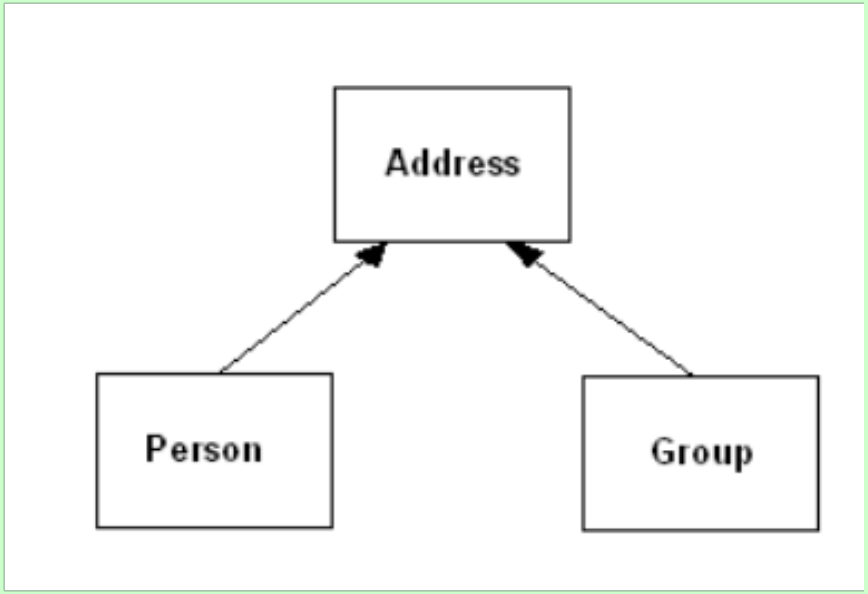
Builder Pattern

- 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

- 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 on 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

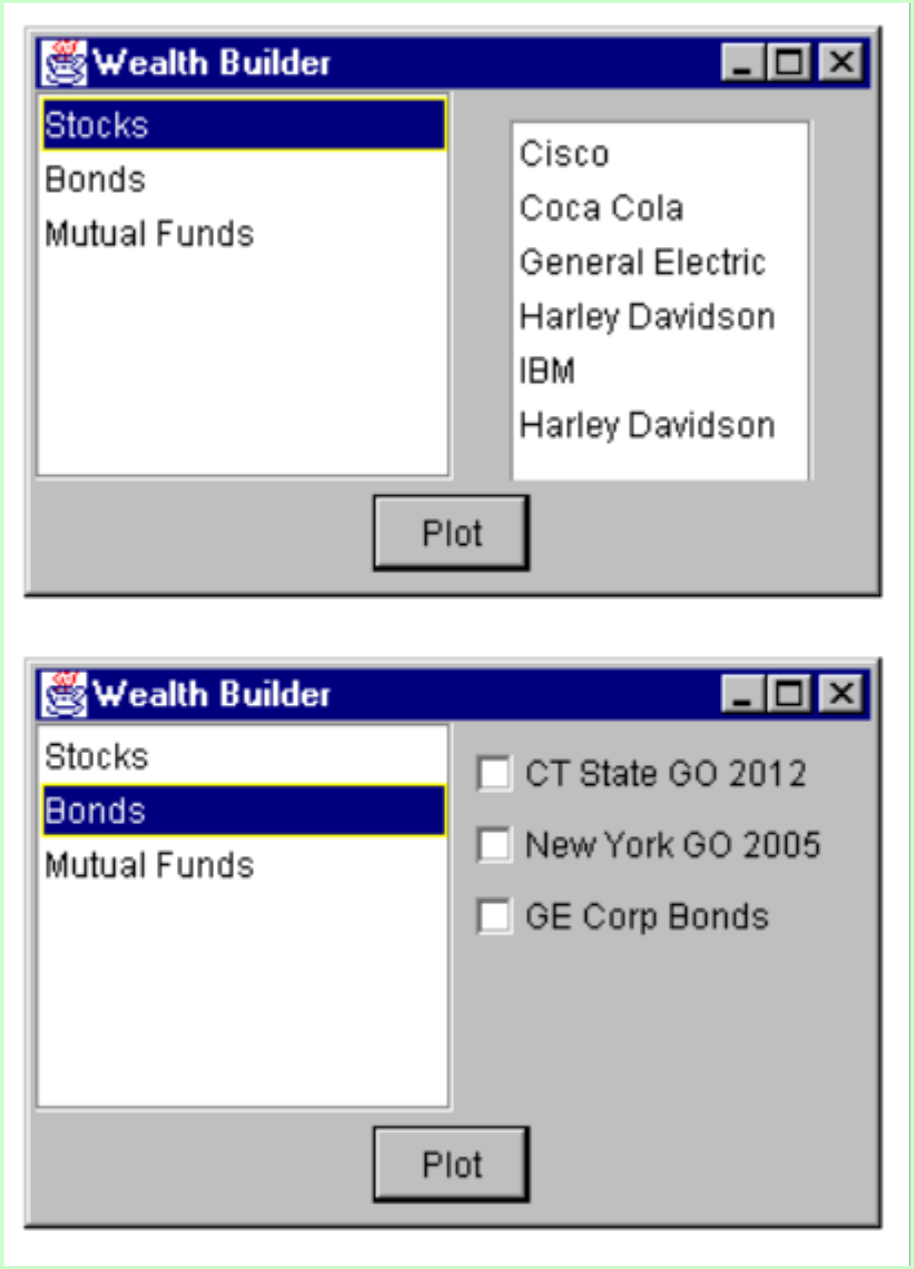
- In each case we want some sort of **multiple choice selection** so that we can select one or more funds to plot.

- 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.
- We'll 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 well 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);
}
```

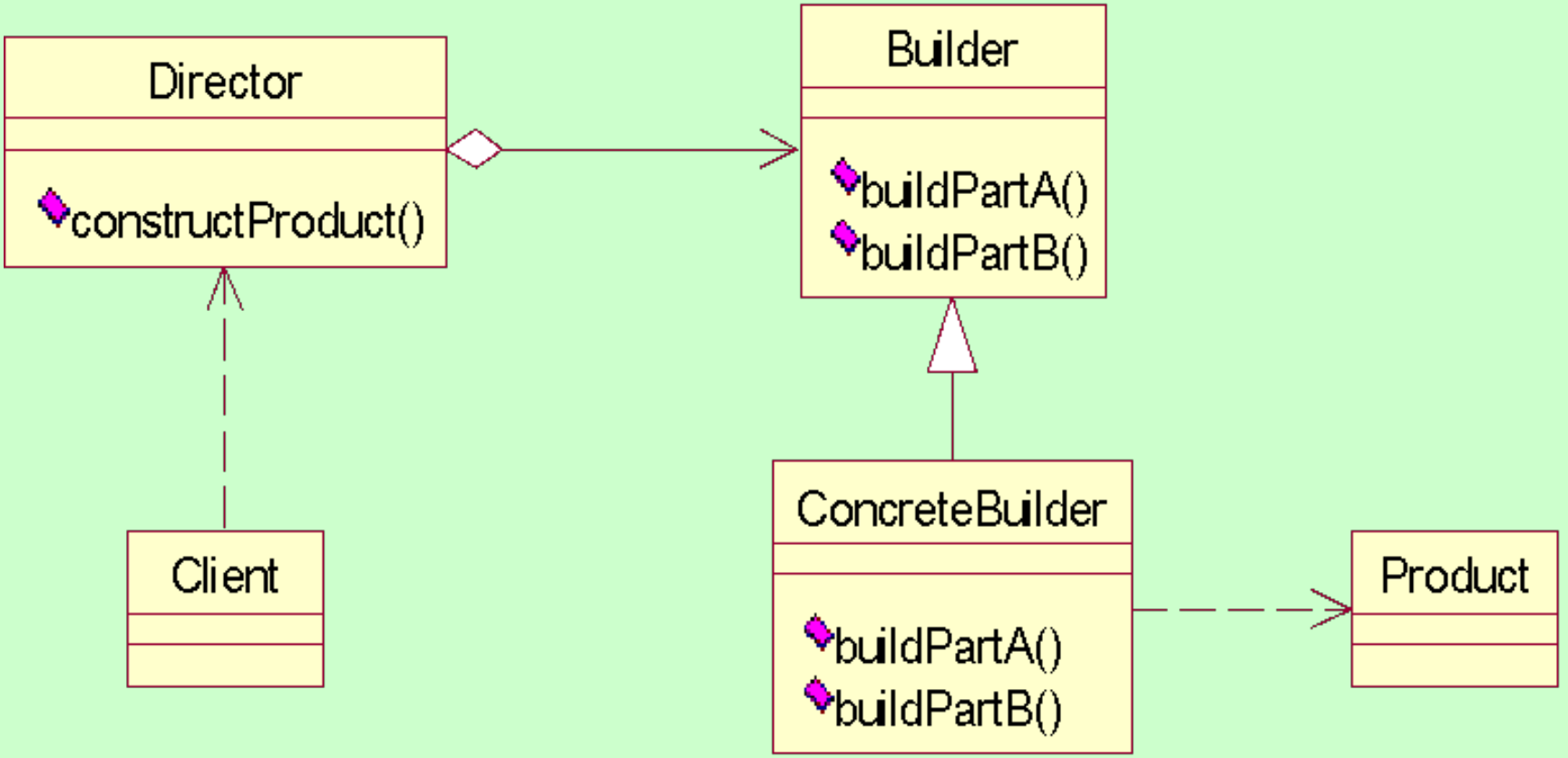
Investment Tracker

- 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

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

Product

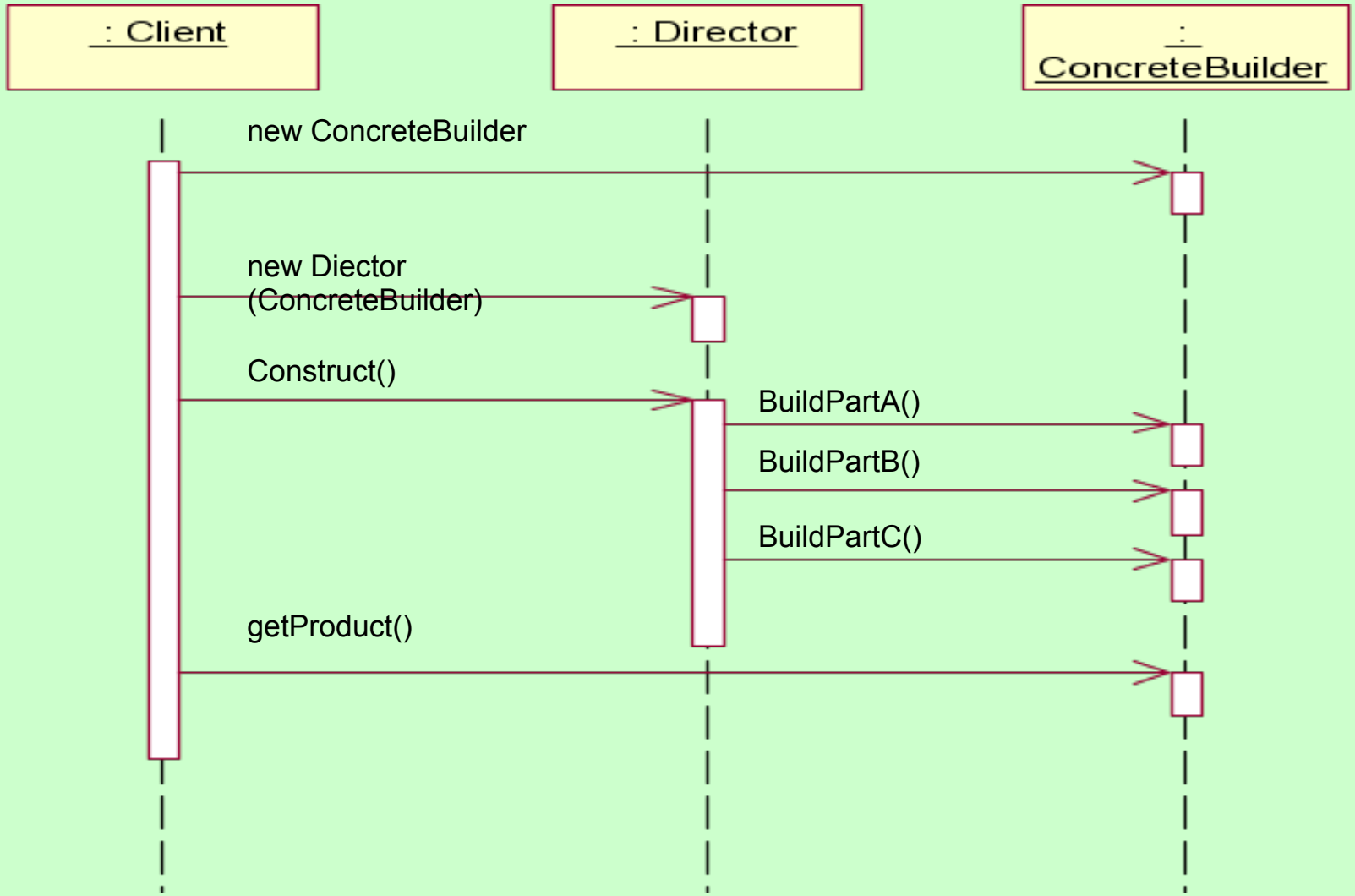
- Complex object under construction. ConcreteBuilder, e.g. ListBoxChoice, builds the products internal representation and defines the process by which it's assembled

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

Collaborations

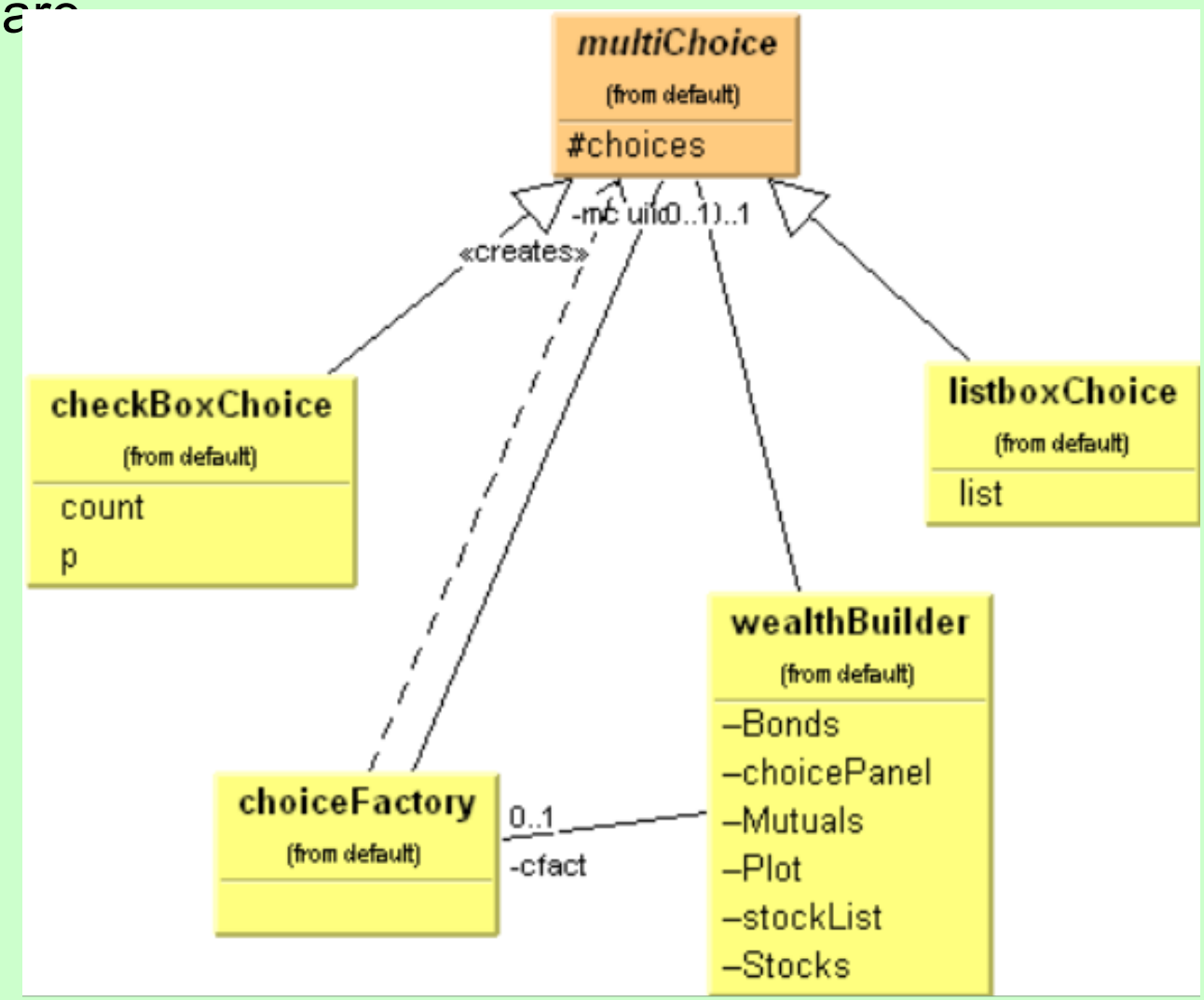
- The **client creates a Director object** and configures it with the desired object
- **Director** notifies **the builder** whenever a part of the product should be built, e.g. the Factory for the transaction GUI is a Director
- **Builder** handles **Director request** and adds parts to the product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

Exercises – Week 4 Builder Pattern

Extend the example in the lecture whereby you now include
Japanese stocks.
Include 5 of these stocks and use a different user display
such as radio buttons.

Continue Assignment 1!