Design Patterns in Java™ Technology

James W. Cooper
IBM T J Watson Research Center

Design Patterns

- Writing objects is easy
- Designing how objects communicate is the new challenge
- Design patterns are a set of tested approaches for specifying these interactions

What Are Design Patterns?

- Recurring solutions to design problems you see over and over (Alpert et al, 1998)
- A set of rules describing how to accomplish certain tasks in the realm of software development (Pree, 1994)
- Focus on reuse of recurring architectural design themes (Coplien and Schmidt, 1995)
- Address a recurring design problem that arises in a specific context and presents a solution to it (Buschmann et al, 1996)
- Identify and specify abstractions that are above the level of single classes or instances, or of components (GoF)

JavaOne

Design Pattern Lingua Franca

- The field has developed its own jargon
- Some writing on this subject has been a bit obscure
- Learning about design patterns entitles you to join an elite fraternity with its own language

Design Patterns from Gang of Four

Creational	Structural	Behavioral
Abstract Factory	Adapter	Chain of Responsibility
Builder	Bridge	Command
Factory	Composite	Interpreter
Prototype	Decorator	Iterator
Singleton	Facade	Mediator
	Flyweight	Memento
	Proxy	Observer
		State
		Strategy
		Template
		Visitor

Design Patterns: Elements of Reusable Object-Oriented Software, Gamma, Helm, Johnson, Vlissides (Addison Wesley)

JavaOne

The Command Pattern

- Let’s begin our study by looking at the powerful, but simple command design pattern
- Used to implement user commands, such as buttons and menus

Setting Up Menu and Buttons

```
mnuFile = new Menu("File", true);
mbar.add(mnuFile);              //File item on menu bar

mnuOpen = new MenuItem("Open...");
mnuFile.add(mnuOpen);           //File | Open
mnuExit = new MenuItem("Exit");
mnuFile.add(mnuExit);           //File | Ext

mnuOpen.addActionListener(this);
mnuExit.addActionListener(this);

btnRed=new Button("Red");       //"Red" button
btnRed.addActionListener(this);
```

The Problem: Menus and Buttons in Java™ Technology All Call Same Method

```
//complicated actionPerformed method
public void actionPerformed(ActionEvent evt) {

Object obj = evt.getSource();        //get object

//menu item File | Exit
if (obj == Exit)
  System.exit(0); //if this is the exit command exit

//some OK button
if (obj == OK)
  OK_Clicked();   //if OK pressed call that method
//etc., etc.      //may be many more such tests
}
```

- This is clearly rather awkward

We Want to Eliminate This If-Else Chain Testing in the Event Listener

- We create a command object interface
- All this means is that any object with a Command interface must have an Execute() method

```
public abstract interface Command
{
  abstract public void Execute();
}
```

Then We Extend the Menu Class to Implement Command

```
public class fileExitCommand extends MenuItem implements Command
{
   public fileExitCommand(String caption)
   {
      super(caption);   //parent class sets up menu
   }
   public void Execute()
   {
      System.exit(0);   //exit when called
   }
}
```

How Do We Create This Menu Item?

```
mnuExit = new fileExitCommand("Exit");
mnuFile.add(mnuExit);   //add to menu bar
mnuExit.addActionListener(this);
```

Then the Action Routine Is Simplified to This

```
//simple actionPerformed method using command objects
public void actionPerformed(ActionEvent evt)
{
//get the object which caused the event
  Command obj = (Command)evt.getSource();
  obj.Execute();              //execute that command
}
```

- You don't have to know which command it is to execute it

You Can Do the Same Thing with Buttons

```
//==================inner class--------------
   public class btnRedCommand extends Button implements Command
   {
      public btnRedCommand(String caption)
      {
         super(caption);
      }
      public void Execute()
      {
         p.setBackground(Color.red);
      }
   }
```

- Note that in order to set the background of a Panel, you need to have access to that Panel as an
  - Inner class, or
  - Argument

Another Take on Command Objects

- Suppose every button or menu item were given its own ActionListener

```
mnuOpen.addActionListener(new fileOpen());
mnuExit.addActionListener(new fileExit());

btnRed=new Button("Red");
btnRed.addActionListener(new btnRed());
```

Typical ActionListeners

- Each of these ActionListeners is in fact a kind of Command pattern

```
public class btnRed implements ActionListener
   {
      public void actionPerformed(ActionEvent e)
      {
          p.setBackground(Color.red);
      }
   }
   //------------------------------------
  public class fileExit implements ActionListener
   {
      public void actionPerformed(ActionEvent e)
      {
         System.exit(0);
      }
   }
```

Giving Each Action Item Its Own Actionlistener

- Means each action is sent to its own object
- This is the intent of the Command pattern
- Since there is not an Execute method this may not be as obvious
- Still generates a lot of little classes

Problems with This Implementation

- The invoking class (menu, button) knows about the details of the command that is executed
- While the actionListener does not know about the kind of action, the UI object does
- A better way would be to have the UI object hold a reference to a command object

We'll Create a CommandHolder

- It keeps and returns the actual command

```
public interface CommandHolder {
   public void setCommand(Command comd);
   public Command getCommand();
}
```

Then We Create a Cmdmenu Object

- Can create several instances for each actual menuItem

```
public class cmdMenu extends JMenuItem
      implements CommandHolder {
   protected Command menuCommand;
   protected JFrame frame;
//------------------------
   public cmdMenu(String name, JFrame frm) {
      super(name);
      frame = frm;
   }
//------------------------
   public void setCommand(Command comd) {
      menuCommand = comd;
   }
//------------------------
   public Command getCommand() {
      return menuCommand;
   }
}
```

We Create the Menus and Put In the Commands

```
mnuOpen = new cmdMenu("Open...", this);
mnuFile.add(mnuOpen);

mnuOpen.setCommand (new fileCommand(this));
mnuExit = new cmdMenu("Exit", this);
mnuExit.setCommand (new ExitCommand());

mnuFile.add(mnuExit);
```

Our FileCommand Is Then Very Simple

```
public class fileCommand implements Command {
   JFrame frame;

   public fileCommand(JFrame fr) {
      frame = fr;
   }
//------------------------------
   public void Execute() {
      FileDialog fDlg = new FileDialog(frame, "Open file");
      fDlg.show();
   }
}
```

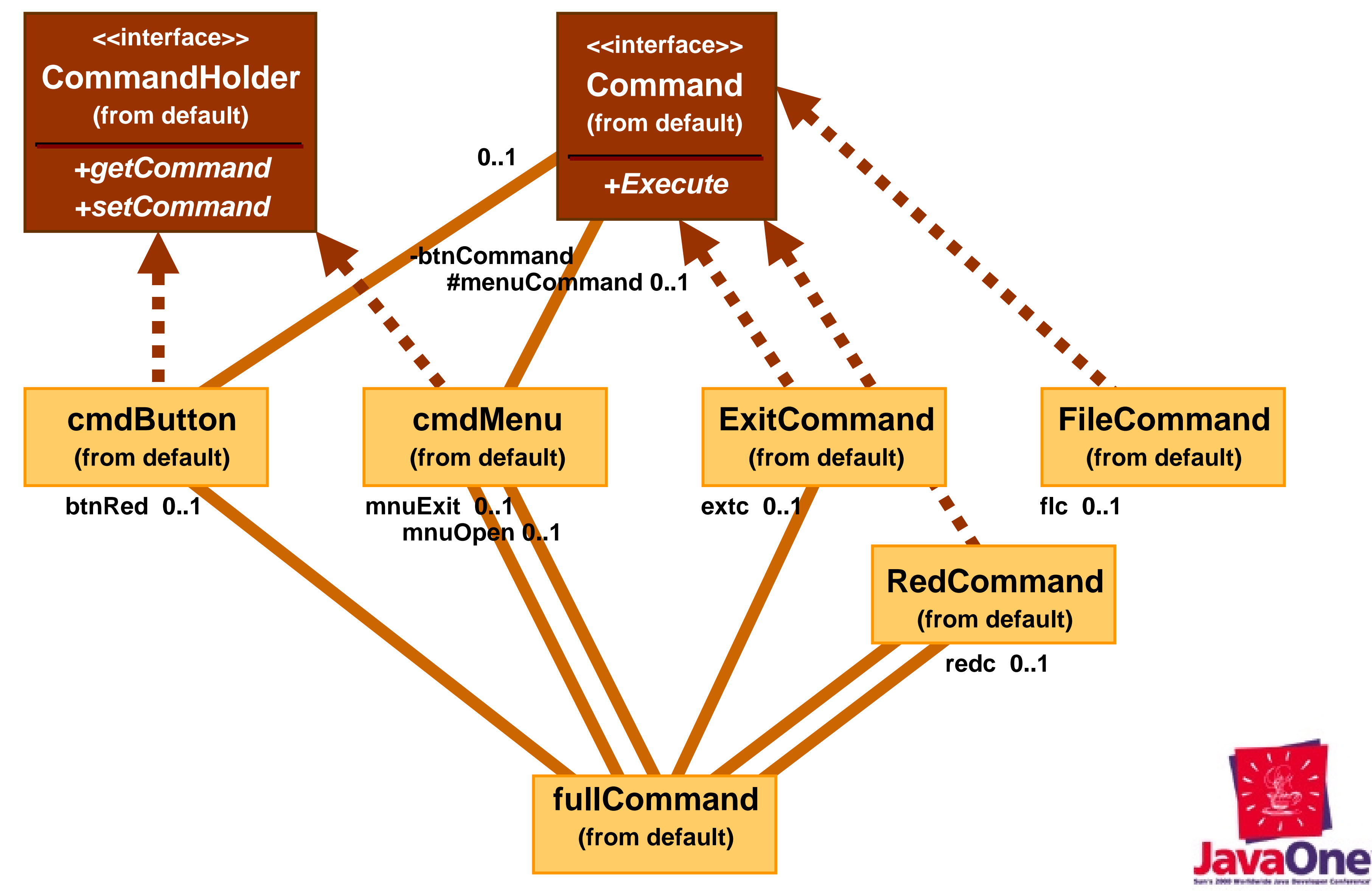
The ExitCommand and RedCommand Are Even Simpler

```
public class ExitCommand implements Command {
  public void Execute () {
     System.exit(0);
  }
}
//=========================================
public class RedCommand implements Command {
   private JFrame frame;
   private JPanel pnl;

   public RedCommand(JFrame fr, JPanel p) {
      frame = fr;
      pnl = p;
   }
   public void Execute() {
      pnl.setBackground(Color.red);
   }
}
```

And Our actionPerformed Is Not Much More Complicated

```
public void actionPerformed(ActionEvent e)   {
        CommandHolder obj = (CommandHolder)e.getSource();
        obj.getCommand().Execute();
    }
```

The Class Diagram for This Pattern



JavaOne

The Command Pattern Can Also Help Us With

- Providing a simple method for Undo
- Providing command logging
- Let’s consider how we could carry out Undo
  - Revise Command interface definition

```
public interface Command    {
   public void Execute();
   public void unDo();
}
```

Both Red and Blue Line Buttons Can Be Undone

Undo Commands

Red Undo Blue

The cmdButton Is the Button Class

```
public class cmdButton extends JButton
   implements CommandHolder {
        private Command btnCommand;
        private JFrame frame;

  public cmdButton(String name, JFrame fr) {
     super(name);
     frame = fr;
  }
  public void setCommand(Command comd) {
      btnCommand = comd;
   }
   public Command getCommand() {
      return btnCommand;
   }
}
```

One for Red, Blue and Undo

```
btRed = new cmdButton("Red", this);
red_command = new redCommand(cp);
btRed.setCommand (red_command);
btRed.addActionListener (this);

btBlue = new cmdButton("Blue", this);
blue_command = new blueCommand(cp);
btBlue.setCommand (blue_command);
btBlue.addActionListener (this);

btUndo = new cmdButton("Undo", this);
u_cmd = new undoCommand();
btUndo.setCommand (u_cmd);
btUndo.addActionListener (this);
```

Inside the undoCommand Object

- In order to perform Undo we must keep a list of the commands
- We decide we can't Undo an Undo

```
public class undoCommand implements Command {
   Vector undoList;

   public undoCommand(){
      undoList = new Vector();      //list of commands to undo
   }
   //------------------------------
   public void add(Command cmd) {
      if(! (cmd instanceof undoCommand))
         undoList.add(cmd);         //add commands into list
   }

   //------------------------------
   public void unDo() {    //does nothing
   }
}
```

JavaOne

The Undo Execute Calls the unDo Method of the Last Stored Command

```
//------------------------------
   public void Execute() {
      int index = undoList.size () -1;

      if (index >=  0) {
         //get last command executed
         Command cmd =  (Command)undoList.elementAt (index);
         cmd.unDo ();                    //undo it
         undoList.remove (index);   //and remove from list
      }
   }
```

Each Button's Command Object

- Stores a list of lines drawn
- Uses a small drawData object

```
public class drawData {
   private int x, y, dx, dy;

   public drawData(int xx, int yy, int dxx, int dyy) {
      x = xx; y = yy;
      dx = dxx; dy = dyy;
   }
   public int getX() {return x;}
   public int getY() {return y;}
   public int getDx() {return dx; }
   public int getDy() {return dy; }

}
```

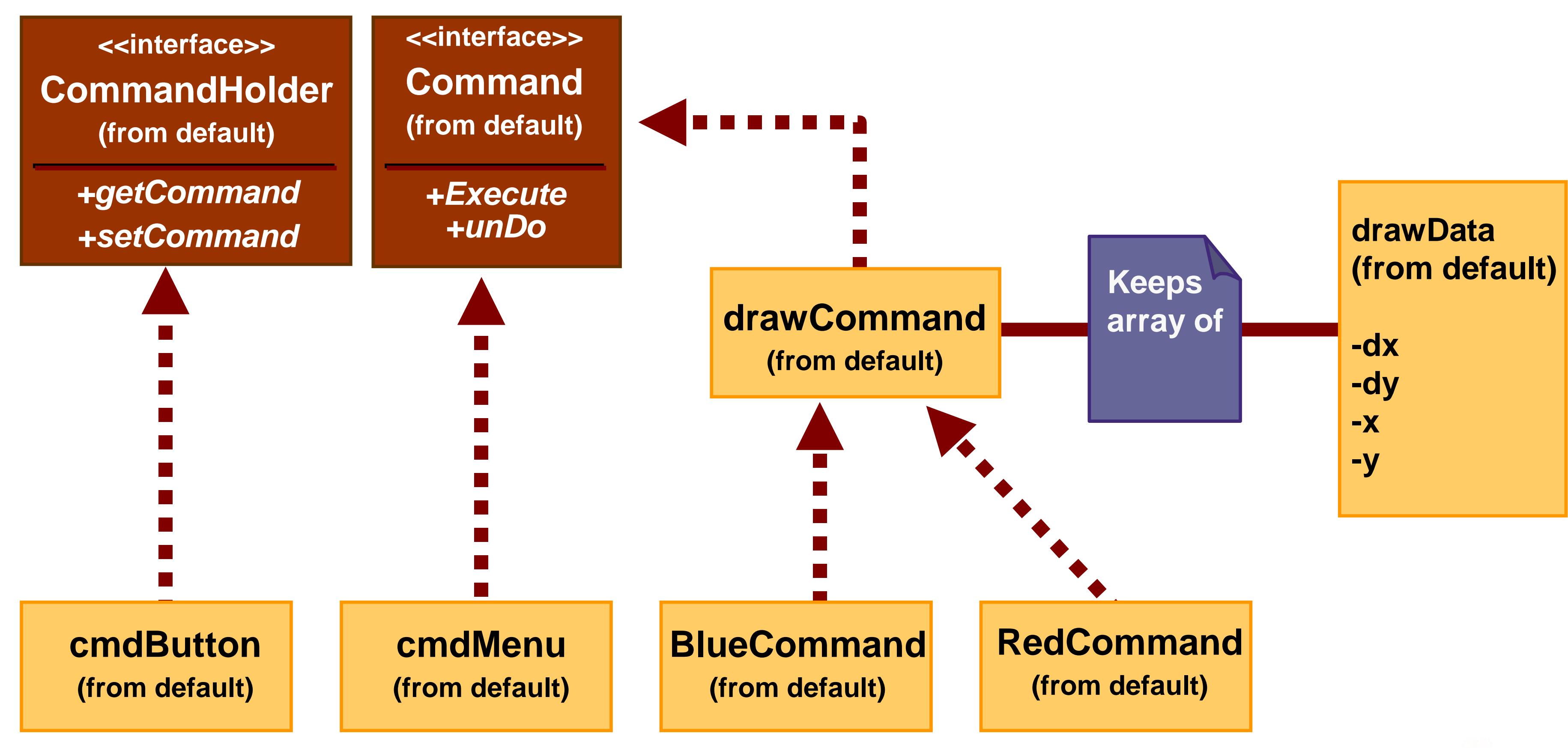
drawCommand Object

```
public drawCommand(JPanel pn) {
        drawList = new Vector();
        p = pn;      //save panel we draw on
    }
    //--------------------------------------
    public void Execute() {
        drawList.add(new drawData(x, y, dx, dy));
        x += dx;    //increment to next position
        y += dy;
        p.repaint();
    }
    //--------------------------------------
    public void unDo() {
        int index = drawList.size() -1;
        //remove last-drawn line from list
        if(index >= 0)  {
                drawData d = (drawData)drawList.elementAt (index);
                drawList.remove (index);  //Removes lines from list
             x = d.getX ();
            y = d.getY ();
        }
        p.repaint();
    }
```

Must Redraw Lines On Repaint

```
public void draw(Graphics g) {
      //draw all remaining lines in list
      //called by panel's paint method
      Dimension sz = p.getSize();
      g.setColor (color);
      for (int i=0; i < drawList.size (); i++) {
         drawData d = (drawData)drawList.elementAt (i);
         g.drawLine (d.getX (), d.getY (),
                     d.getX()+dx, d.getY()+sz.height );
      }
   }
```

Class Structure for Undoables



In Summary, a Command Pattern

- Adds an Execute method to each object
- The event listener can then call the Execute event on the object without worrying about what type it is
- Cleanly separates the event processing from the actions to be carried out

Implications of Command Pattern

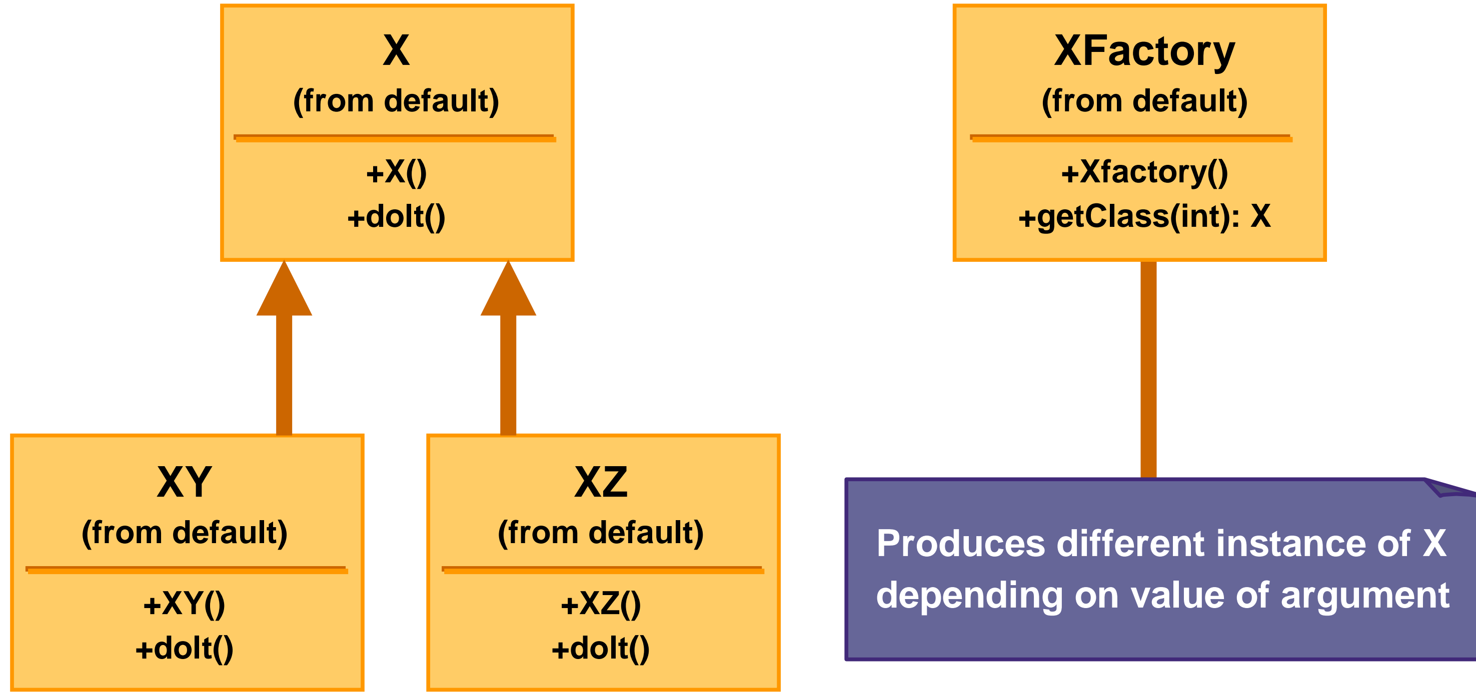
- Separates action from UI
- Keeps actions associates with individual objects
- Requires that every button be its own class
- All Command objects need access to UI somehow
  - Inner classes
  - Constructor or set…methods

Now Let's Consider the Factory Pattern

- A Factory pattern is one that returns an instance of one of several possible classes depending on the data which it is presented with
- Usually each of these classes is derived from a common parent class

Simple Factory Pattern

- X is a base class
  - Xy and xz are derived from it
- Getclass sends argument to factory
- Factory returns one derived class of x
- Since all have same methods it doesn't matter which



Consider a Name Splitting Class

- Should recognize
  - First, last or
  - Last, first

```
class Namer {
//a simple class to take a string apart into two names
   protected String last;  //store last name here
   protected String first; //store first name here

   public String getFirst()    {
      return first;     //return first name
   }
   public String getLast()    {
        return last;    //return last name
   }
}
```

First Name First

```
public class FirstFirst extends Namer   {
   //extracts first name from last name when sep'd by a space
   public FirstFirst(String s)   {
      int i = s.lastIndexOf(" ");
      if (i>0)
         {
         first = s.substring(0, i).trim();
         last =s.substring(i+1).trim();
         }
      else
         {
         first = "";
         last = s;   // if no space, put all in last name
         }
   }
}
```

Last Name First

```
public class LastFirst extends Namer   {
//extracts last name from first name when sep'd by comma
   public LastFirst(String s)   {
      int i = s.indexOf(",");
      if (i > 0)
         {
         last = s.substring(0, i).trim();
         first = s.substring(i + 1).trim();
         }
      else
         {
         last = s;      //if no comma, put all in last
         first = "";
         }
   }
}
```

The Factory

```
Public class NameFactory {
//Factory decides which class to return based on
//presence of a comma
   public Namer getNamer(String entry) {
      int i = entry.indexOf(",");   //comma det name order
      if (i > 0)
         return new LastFirst(entry);
      else
         return new FirstFirst(entry);
   }
}
```

How to Use the Factory

```
private void computeName()
 {
    namer = nfactory.getNamer(entryField.getText());

    //get the right class from the factory
    txFirstName.setText(namer.getFirst());
    txLastName.setText(namer.getLast());
 }
```

Name Divider

Enter name:

Schlunks, Seymour

First name: Seymour

Last name: Schlunks

Compute | Clear | Close

The Factory Method Pattern

- Defines an interface for creating an object, but lets the subclasses decide which one to instantiate
- Parent class is abstract
  - Each subclass may instantiate a different class of a different class hierarchy

Consider a Series of Swimming Championship Races

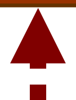
- Short events are swum in preliminaries and the top swimmers return to swim finals
- Long events are swum only once because
  - they tire the competitors too much
  - watching them is like watching paint dry
- Preliminary events are “circle seeded”
  - Top 3 heats mix fastest competitors
- Long (timed final) events are “straight seeded”
  - Seeded slowest to fastest
  - Fastest in each heat are in middle lanes

There are Two Hierarchies of Classes

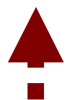
- Event classes can be
  - TimedFinalEvent
  - PrelimEvent
- Seeding can be
  - StraightSeeding
  - CircleSeeding

Classes in Seeding Swimmers

Seeding (from default)
- +Seeding
- +getCount
- +getHeats
- +getSwimmers



StraightSeeding (from default)



CircleSeeding (from default)

Swimmer (from default)
- -age
- -club
- -firstName
- -heat
- -last Name
- -time

Returns Straight Seeding

Event (from default)
- +Event
- +getSeeding
- +isFinal
- +isPrelim
- +isTimedFinal

PrelimEvent (from default)
- +PrelimEvent
- +getSeeding
- +isFinal
- +isPrelim
- +isTimedFinal

TimedFinalEvent (from default)
- +TimedFinalEvent
- +getSeeding
- +isFinal
- +isPrelim
- +isTimedFinal

Each Class Starts With an Abstract Definition

```
public abstract class Event
 {
   protected int numLanes;
   protected Vector swimmers;
   public Event(String filename, int lanes) {
      numLanes = lanes;
      swimmers = new Vector();
       //read in swimmers from database or file
  }
   public abstract Seeding getSeeding();
   public abstract boolean isPrelim();
   public abstract boolean isFinal();
   public abstract boolean isTimedFinal();
}
//==================================================
public abstract class Seeding {
   public abstract Enumeration getSwimmers();
   public abstract int getCount();
   public abstract int getHeats();
}
```

Derived Event Classes

```
public class TimedFinalEvent extends Event {
//creates an event that will be straight seeded
   public TimedFinalEvent(String filename, int lanes) {
      super(filename, lanes);
   }
   public  Seeding getSeeding() {
      return new StraightSeeding(swimmers, numLanes);
   }
   public boolean isPrelim() {return false;}
   public boolean isFinal() {return false;}
   public boolean isTimedFinal() {return true;}
}
//==================================================
public class PrelimEvent extends Event {
 //creates a preliminary event which is circle seeded
   public PrelimEvent(String filename, int lanes) {
      super(filename, lanes);
   }
   public  Seeding getSeeding() {
      return new CircleSeeding(swimmers, numLanes);
   }
   public boolean isPrelim() { return true;}
   public boolean isFinal() { return false;}
   public boolean isTimedFinal() {return false;}
 }
```

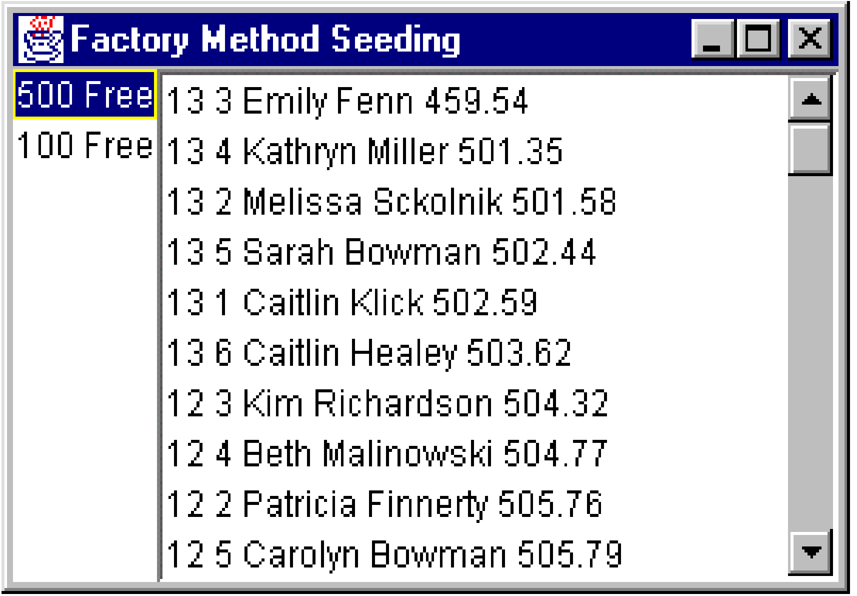
Derived StraightSeeding Class

```
public class StraightSeeding extends Seeding {

   protected Vector Swimmers;   //swimmers here
   protected Swimmer[] asw;     //sorted here
   protected int numLanes;      //# of lanes
   protected int[] lanes;       //order of lanes
   protected int count;         //number of swimmers
   protected int numHeats;      //number of heats

   public StraightSeeding(Vector sw, int lanes) {
      Swimmers = sw;
      numLanes = lanes;
      count = sw.size();
      calcLaneOrder();     //determine lane order
      seed();               //seed into order
   }
   //--------------------------------
   protected void seed() {
```

Derived Circle Seeding Class

```
public class CircleSeeding extends
StraightSeeding {

  public CircleSeeding(Vector sw, int lanes)
{
  super(sw, lanes);
  super.seed();        //straight seed
  seed();              //circle seed top heats
}
//----------------------------
protected void seed() {     //do circle
seeding
```

Result of Seeding



Now Let's Consider the Abstract Factory

- It is a Factory object that returns several possible Factories
- Use when you want to return one of several related classes of objects
- Examples
  - GUI Factory for variable look and feel
  - Auto Factory for different kinds of cars

Consider a Garden Maker Factory

- Program to plan the layout of gardens
  - Vegetable gardens
  - Annual gardens
  - Perennial gardens
- In all cases you ask the same questions
  - List good border plants
  - List good central plants
  - List good partial-shade plants

The Base Garden Class

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

Simple Plant Object

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

Simple Vegetable Garden

- Just returns the name of one plant of each kind

```
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");
  }
}
```

Garden Maker

```
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;
   }

}
```

The Abstract Factory in Action

Garden planner

Garden type

Vegetable

Annual

Perennial

Broccoli

Corn

Peas

Central Border Shade

Quit

Abstract Factory Class Structure

Annual Garden (from default)
+Annual Garden
+getBorder
+getCenter
+getShade

Garden (from default)
+Garden
+getBorder
+getCenter
+getShade

VegieGarden (from default)
+VegieGarden
+getBorder
+getCenter
+getShade

Plant (from default)
name
+Plant
+getName

PerennialGarden (from default)
+Perennial Garden
+getBorder
+getCenter
+getShade

-gd 0..1

GardenMaker (from default) {local to package}
GardenMaker
+getGarden

Returns instances of Garden

Summary of the Abstract Factory

- Use when you want to return one of several related classes of objects, each of which can return a group of objects
- Isolates the actual classes that are generated
- Can change or replace generated classes without affecting program

The Builder Pattern

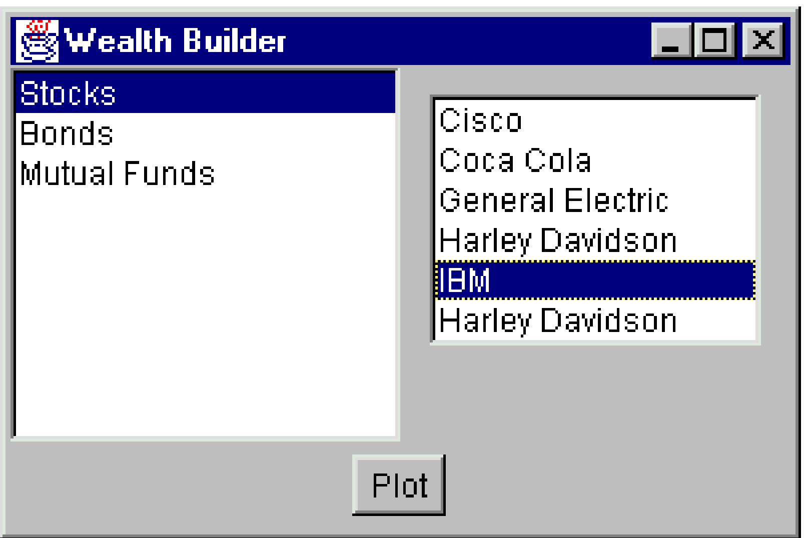
- Constructs a complex object from several component objects
- Constructs different complex objects based on some input data
- Consider an E-mail program Address book
  - One format for people
  - Another format for groups

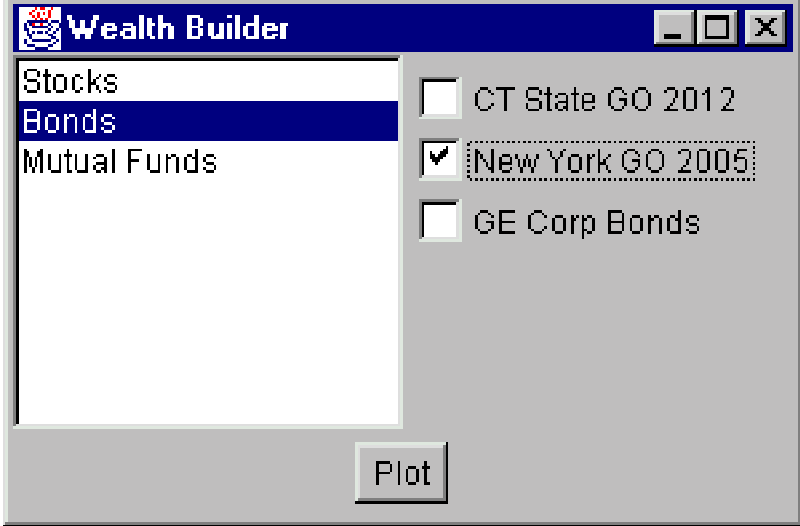
Consider an Investment Tracker Program

- …since this is a simpler example to implement…
- We want to display a list of our holdings in
  - Stocks
  - Bonds
  - Mutual funds
- The number of each we have determines the kind of display we see

Variable Display

- List box for large number of holdings
- Check boxes for small number of holdings
- Can multi-select in either case

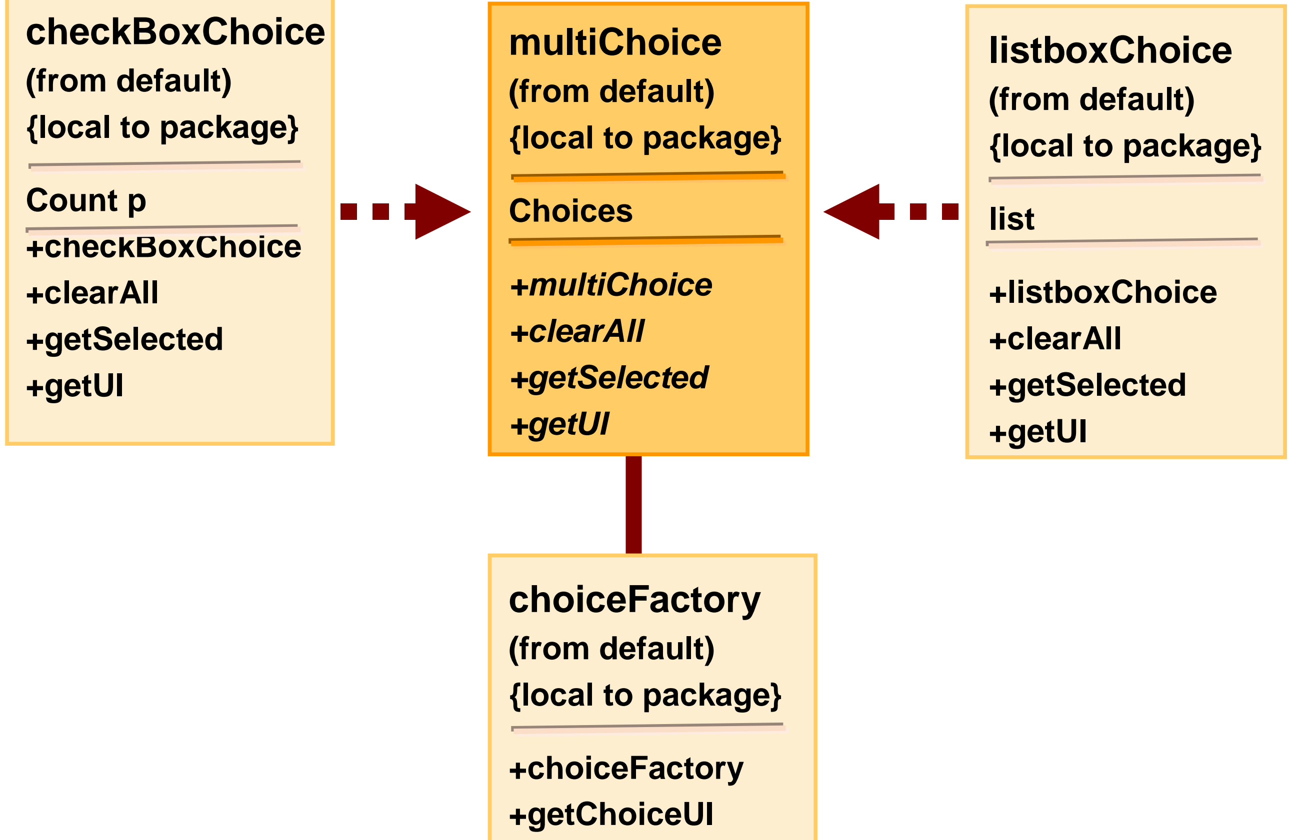




The MultiChoice Abstract Class

```
abstract class multiChoice
{
   //This is the abstract base class
   //that the listbox and checkbox choice panels
   //are derived from
   private 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 selected
   abstract public void clearAll();          //clear all items
}
```

The Class Structure



The Choice Factory

```
Public class choiceFactory
{
   private 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 checQkboxes
        ui = new checkBoxChoice(choices);
      else
        //return a multi-select listbox panel
        ui = new listboxChoice(choices);
      return ui;
   }
}
```


JavaOne

The Checkbox Interface Builder

```
public class checkBoxChoice extends multiChoice {
   //This derived class creates
   //vertical grid of checkboxes
   int count;            //number of checkboxes
   Panel p;              //contained in here
//---------------------------------------------
   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 List Interface Builder

```
public class listboxChoice extends multiChoice {
  private List list;
//---------------------------------------------
   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.setMultipleMode(true);
      p.add(list);
      for (int i=0; i< choices.size(); i++)
         list.addItem((String)choices.elementAt(i));
     return p;
  }
```


JavaOne Sun's 2000 Worldwide Java Developer Conference

Calling the Builder

```
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 rt panel
choicePanel.validate();          //re-layout and display
Plot.setEnabled(true);           //allow plots
```

Consequences of the Builder Pattern

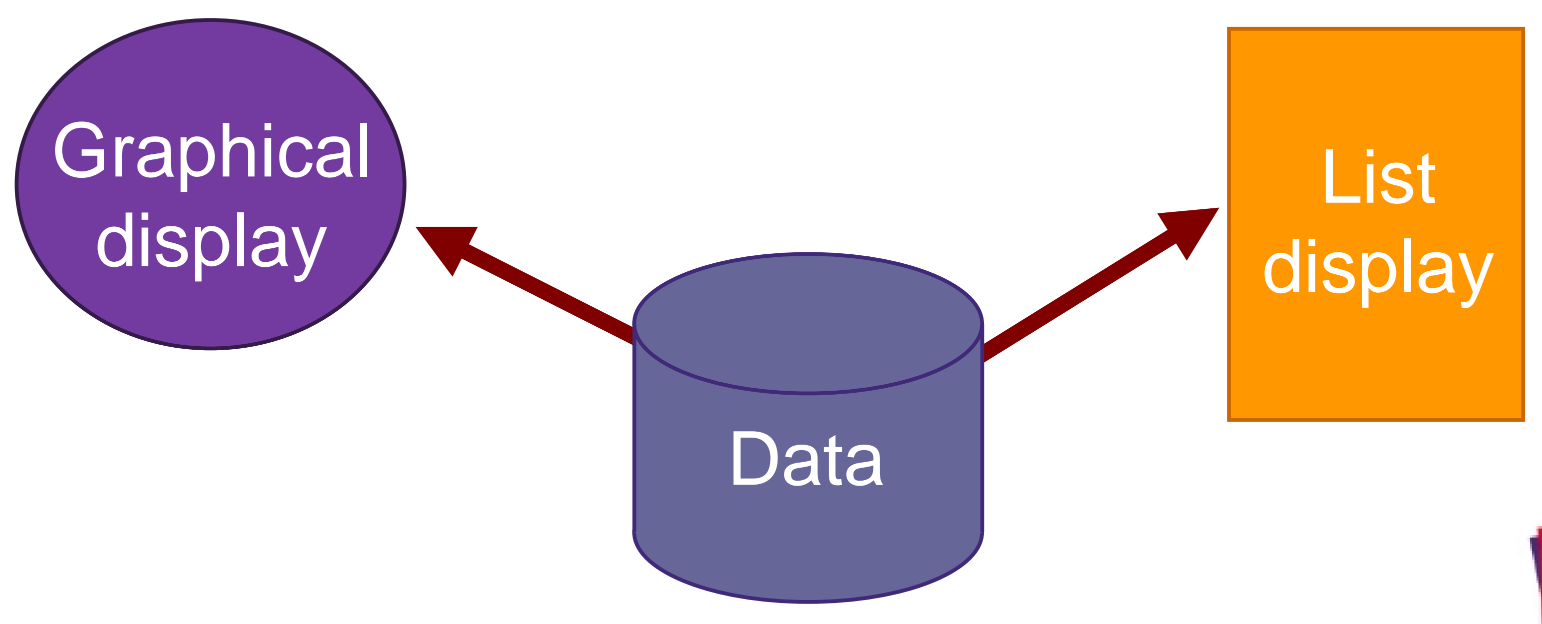
- Lets you vary the internal representation of the object you build
  - Hides how it is assembled
- Each builder is independent of the others
  - Improves modularity and makes other builders easy to add

Differences Between Patterns

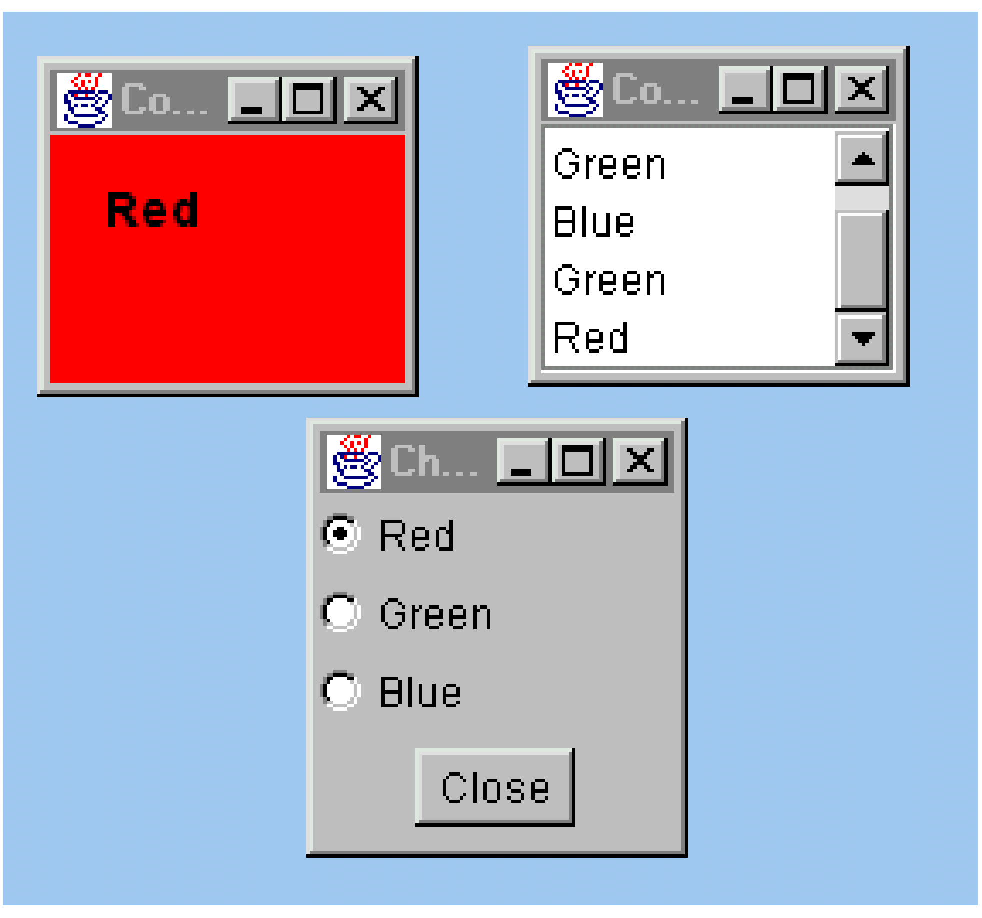
- Factories
  - Return one of several related classes
- Abstract Factories
  - Returns families of related classes
- Builders
  - Construct classes from components and return the right one
  - Often, a Factory is used to pick the right one

The Observer Pattern

- Helps you separate data objects from display objects
- Can have several displays of the data at once



Display Changes in Data in Two Or More Views



An Observer Pattern

- Assumes that the data object is separate from the display objects
- We call the data the Subject and the display windows the Observers
- Each observer registers its interest in changes in the data
  - There could be several types of changes
- Each observer has a known interface the subject can call

The Observer and Subject Interfaces

```
abstract interface Observer
{
   public void sendNotify(String s);
}
//=============================================
abstract interface Subject
{
   public void registerInterest(Observer obs);
}
```

The Main Program Is Subject

- The data are the list of color changes
- The observer windows register their interest in these changes

```
public class Watch2L extends JFrame
   implements Subject, ActionListener, ItemListener
{
   Button Close;
   JRadioButton red, green, blue;
   Vector observers;
//------------------------------------------
   public Watch2L()   {
   super("Change 2 other frames");
    observers = new Vector();          //list of observing frames
```

Change ...

Red

Green

Blue

Close

Register Interest and Notify Observers

```
public void registerInterest(Observer obs)
  {
     //adds observer to list
     observers.addElement(obs);
  }
 //-----------------------------------------
  private void notifyObservers(JRadioButton rad)
  {
     //sends text of selected button to all observers
     String color = rad.getText();
     for (int i=0; i< observers.size(); i++)
     {
        ((Observer)(observers.elementAt(i))).sendNotify(color);
     }
  }
```

Notify Observers When Color Changes

```
public void itemStateChanged(ItemEvent e)
   {
      //responds to radio button clicks
      //if the button is selected
      if(e.getStateChange() == ItemEvent.SELECTED)
         notifyObservers((JRadioButton)e.getSource());
   }
```

Observers Receive Notification

```
class ListFrame extends JFrame
   implements Observer
{
   JList list;
   JPanel p;
   JScrollPane lsp;
   JListData listData;

   public ListFrame(Subject s)
   {
    super("Color List");
    s.registerInterest(this);
   // :  etc.
   //--------------------------------
   public void sendNotify(String s)
   {
      listData.addElement(s);
   }
```

Color Window Observer

```
class ColorFrame extends JFrame  implements Observer
{
   Color color;
   String color_name="black";
   Font font;
   JPanel p = new JPanel(true);

   public ColorFrame(Subject s)
  {
    super("Colors");
    getContentPane().add("Center",  p);
    s.registerInterest(this);
    //etc..
   public void sendNotify(String s)
   {
      color_name = s;
      if(s.toUpperCase().equals("RED"))
         color = Color.red;
      if(s.toUpperCase().equals("BLUE"))
        color =Color.blue;
      if(s.toUpperCase().equals("GREEN"))
        color = Color.green;
      //p.repaint();
      setBackground(color);
   }
```

Problems to Resolve

- Data formats may vary
  - Color list wants text strings
  - Color frame wants Color objects
- There may be more than one interesting event
  - Data added
  - Data deleted
  - Data changed
  - Each might need its own kind of Notify

The JList class in the Java™ Foundation Classes API (JFC)

- Differs from the List class in the AWT
- You put your data in an array or Vector
  - Inside a class derived from AbstractListModel
- Put that data in the JList constructor
- (You are also responsible for the scroller)

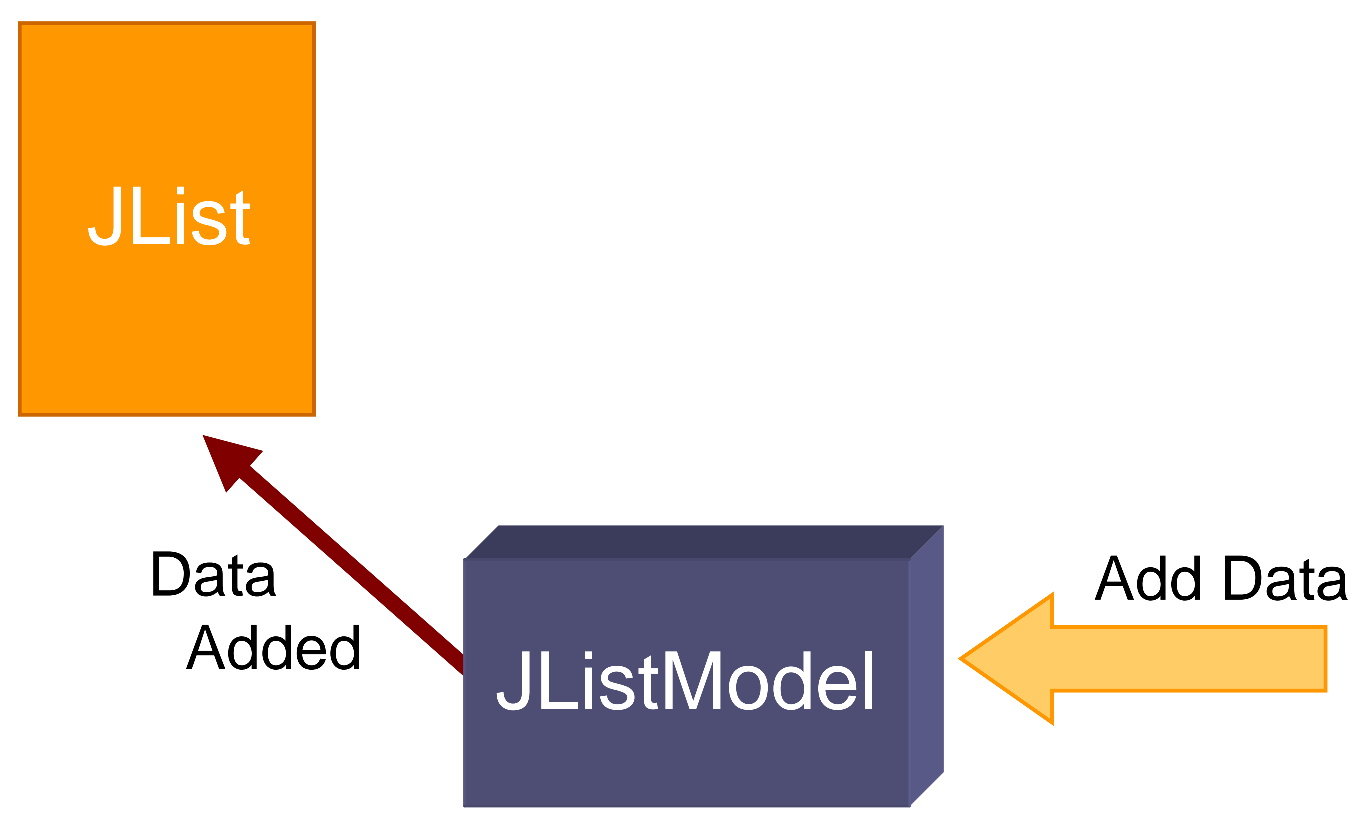
```
//Create the list
listData = new JListData();   //the list model
list = new JList(listData);   //the visual list
lsp = new JScrollPane();      //the scroller
lsp.getViewport().add(list);
p.add("Center", lsp);
```

Using the JListData Class

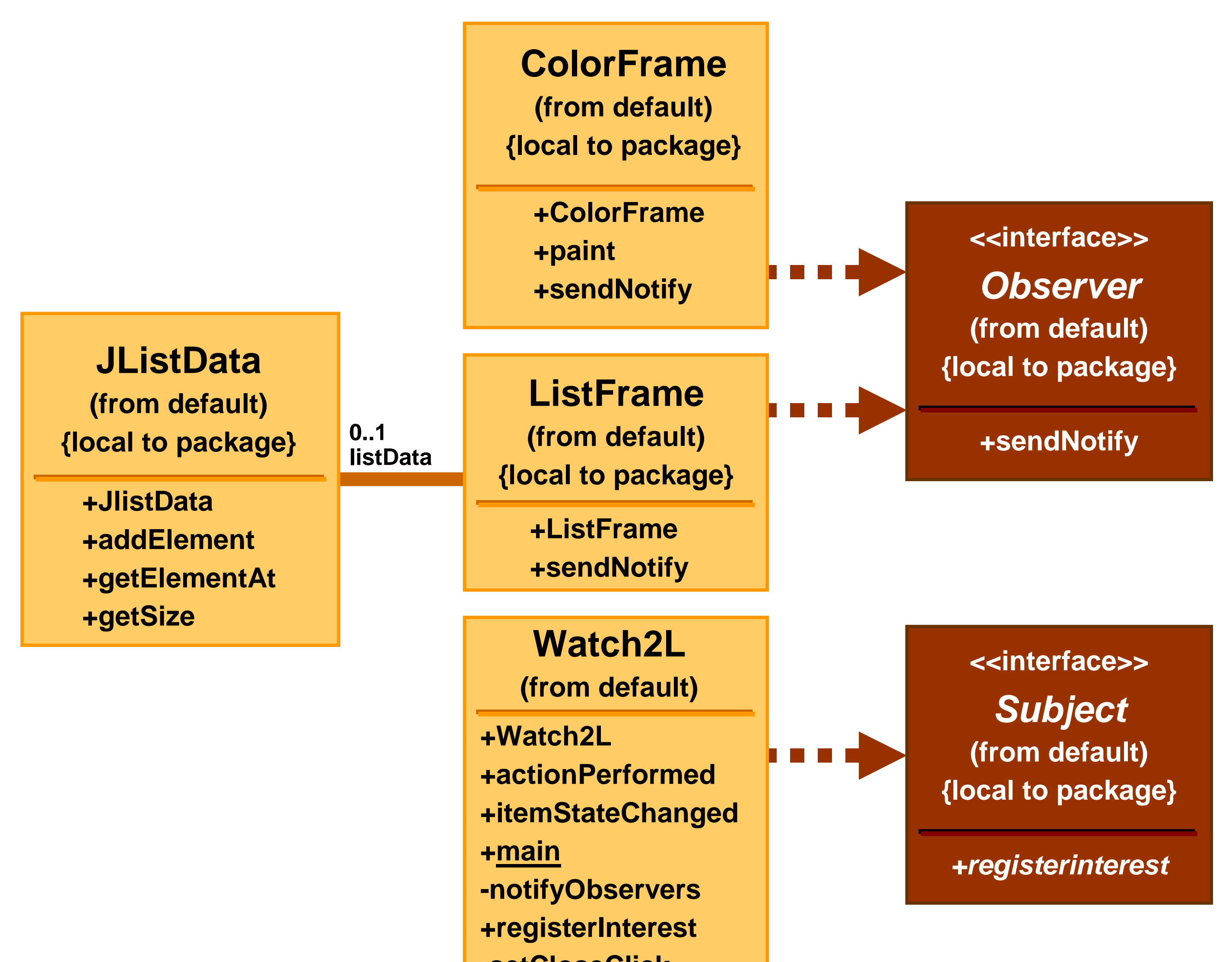
```
class JListData extends AbstractListModel
{
   private Vector data;
   public JListData()
   {
      data = new Vector();
   }
   public int getSize()
   {
      return data.size();
   }
   public Object getElementAt(int index)
   {
      return data.elementAt(index);
   }
   public void addElement(String s)
   {
      data.addElement(s);
      fireIntervalAdded(this, data.size()-1, data.size());
   }
}
```

This event tells the JList that new data has been added

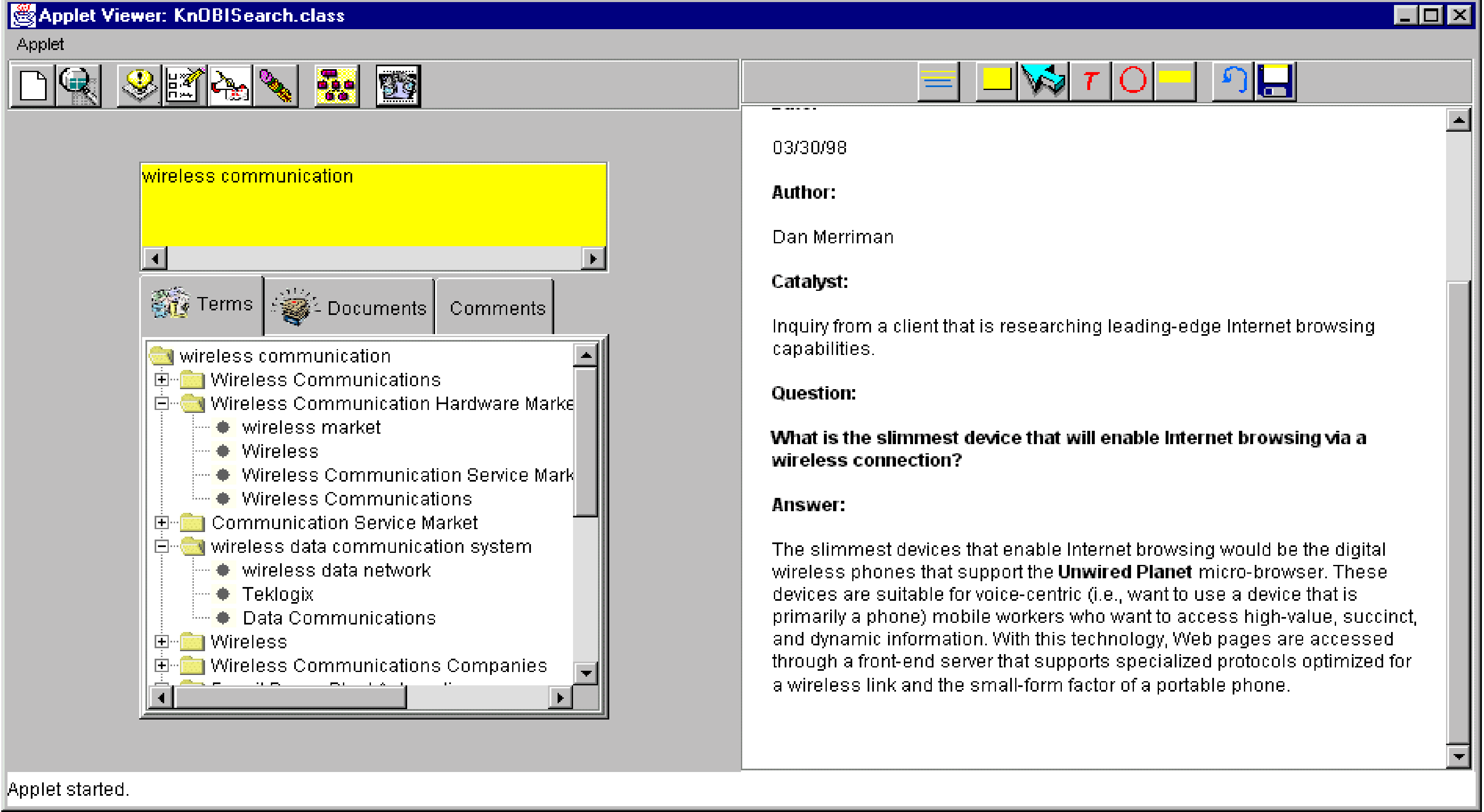
In Other Words JList Is Itself an Observer of the JListModel Data!

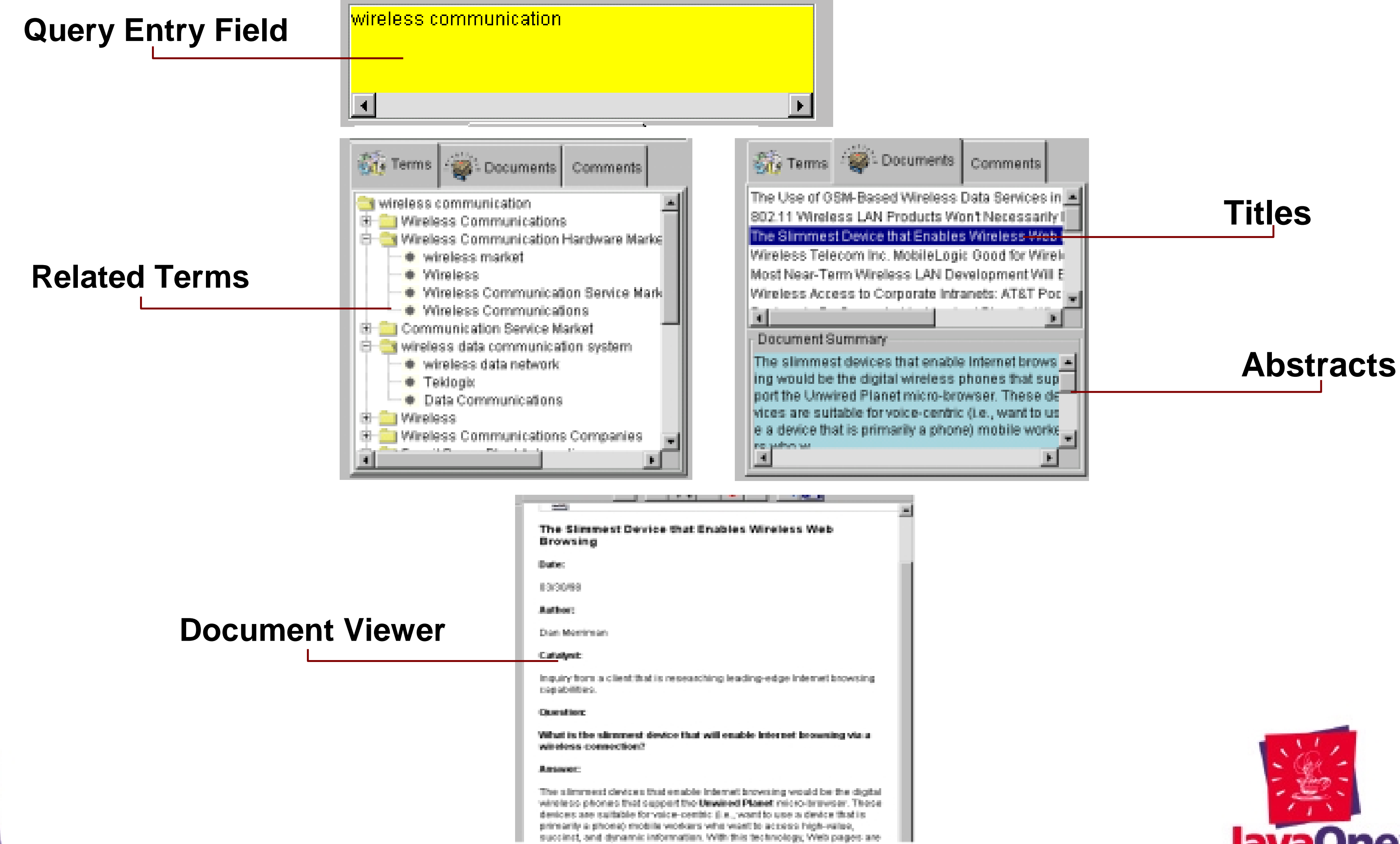


Overall Class Structure



A Case Study: A Search Applet



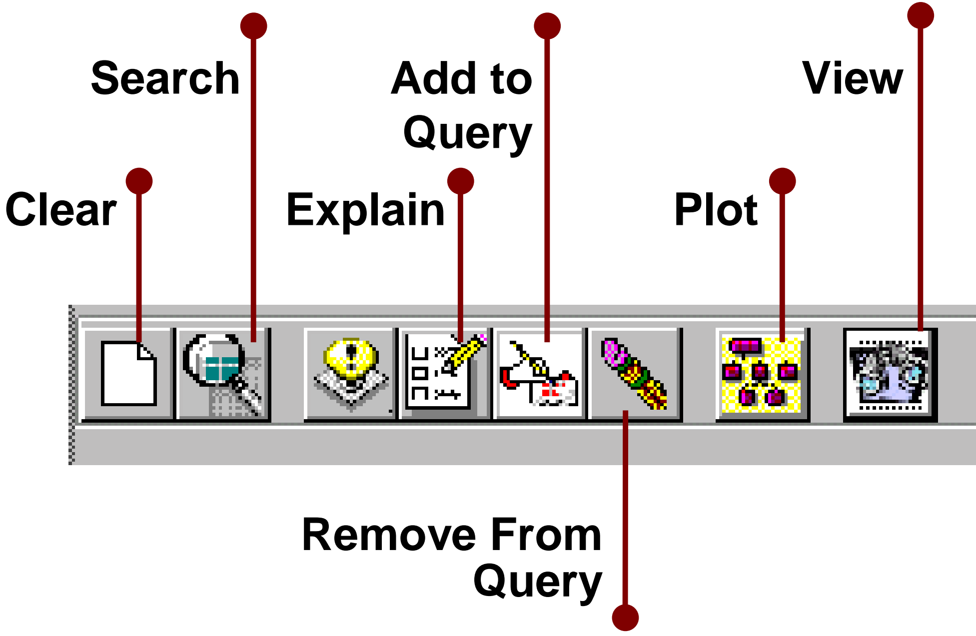

Query Entry Field
wireless communication
Related Terms
Terms
Documents
Comments
wireless communication
Wireless Communications
Wireless Communication Hardware Marke
wireless market
Wireless
Wireless Communication Service Mark
Wireless Communications
Communication Service Market
wireless data communication system
wireless data network
Teklogix
Data Communications
Wireless
Wireless Communications Companies
Titles
The Use of GSM-Based Wireless Data Services in
802.11 Wireless LAN Products Won't Necessarily
The Slimmest Device that Enables Wireless Web
Wireless Telecom Inc. MobileLogic Good for Wirel
Most Near-Term Wireless LAN Development Will E
Wireless Access to Corporate Intranets: AT&T Poc
Document Summary
The slimmest devices that enable Internet brows
ing would be the digital wireless phones that sup
port the Unwired Planet micro-browser. These de
vices are suitable for voice-centric (i.e., want to us
e a device that is primarily a phone) mobile worke
Abstracts
Document Viewer
The Slimmest Device that Enables Wireless Web Browsing
Date:
Author:
Catalyst:
Question:
What is the slimmest device that will enable Internet browsing via a wireless connection?
Answer:

Like Many Programs, This Grew By Attrition

- Started with toolbar and treelist in a “lets learn the JFC” program
- Added communication to search engine
- Added tabbed dialog and displays of documents and abstracts
- Added document viewer

The Action Toolbar

- Each tool button affects several other objects on the display

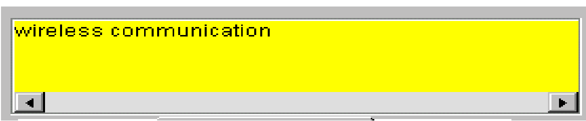


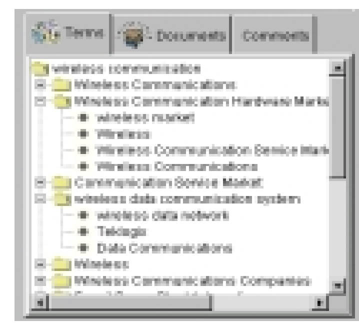
Problem of Tangled Events

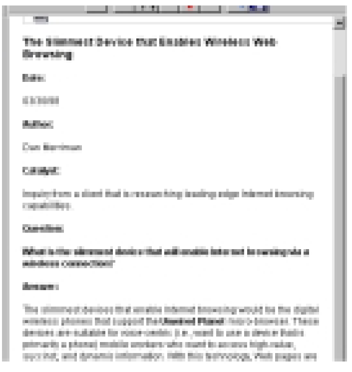
- Tool button events call searches, clear panels and send data to display windows
- Every button needs to know about internals of several windows
- Can lead to a long tangle of inner classes so they can all access the UI elements

Define the Major Objects

- Visual objects
  - Toolbar
  - Query field
  - Tabbed dialog
  - View area

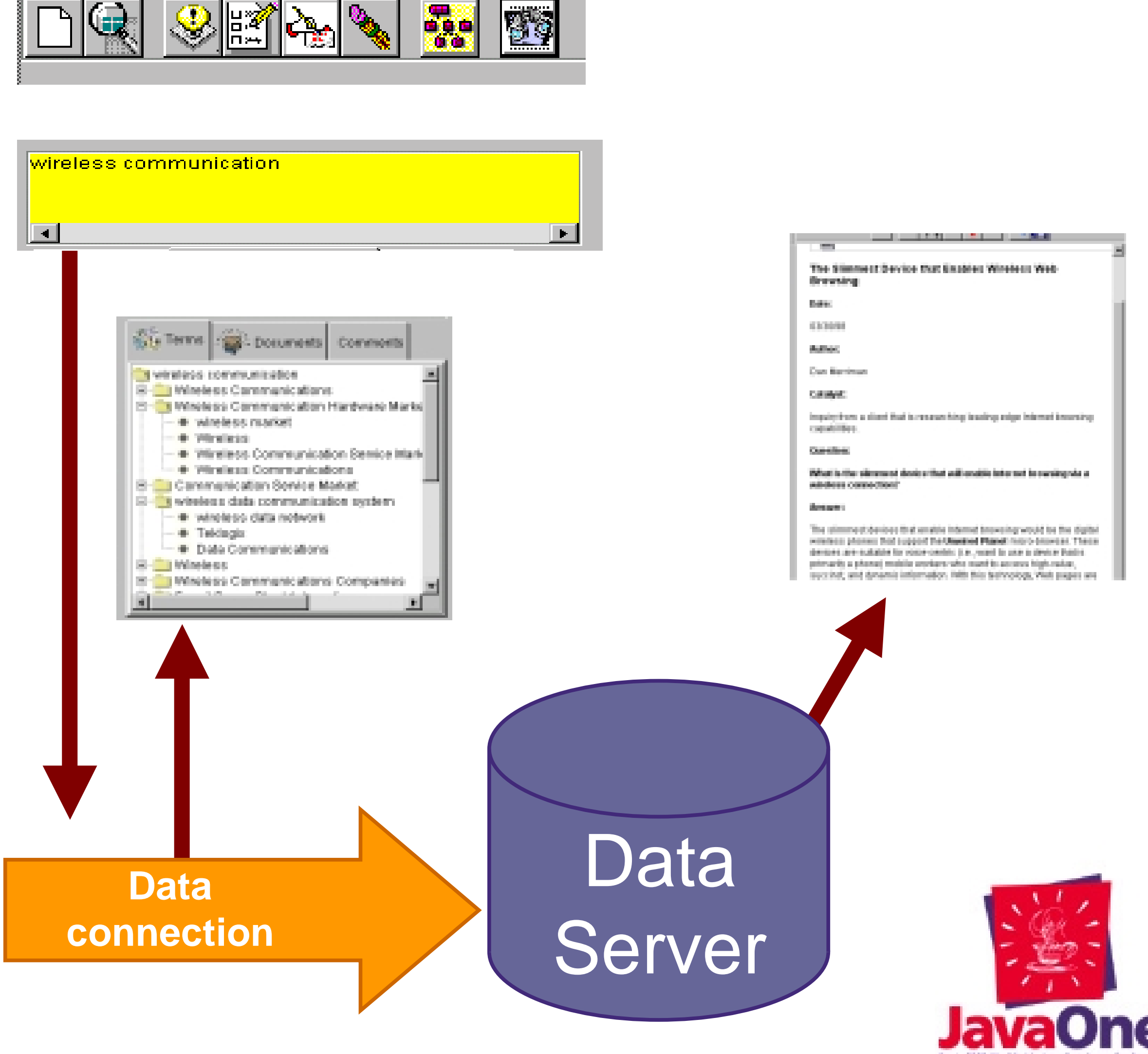






One Possible Object Model

- But how do buttons communicate with data and displays?



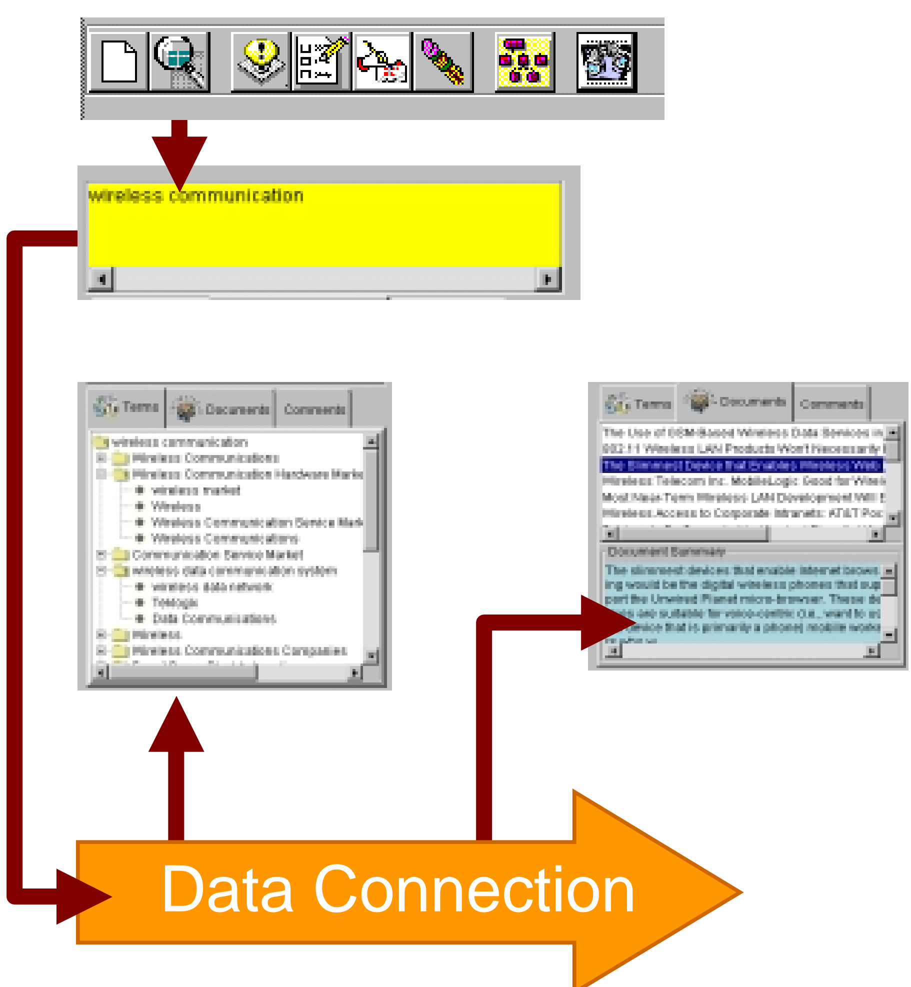
Buttons Could Be Command Pbjects

- Toolbar can be an actionListener
- But whose methods do the Execute methods call?

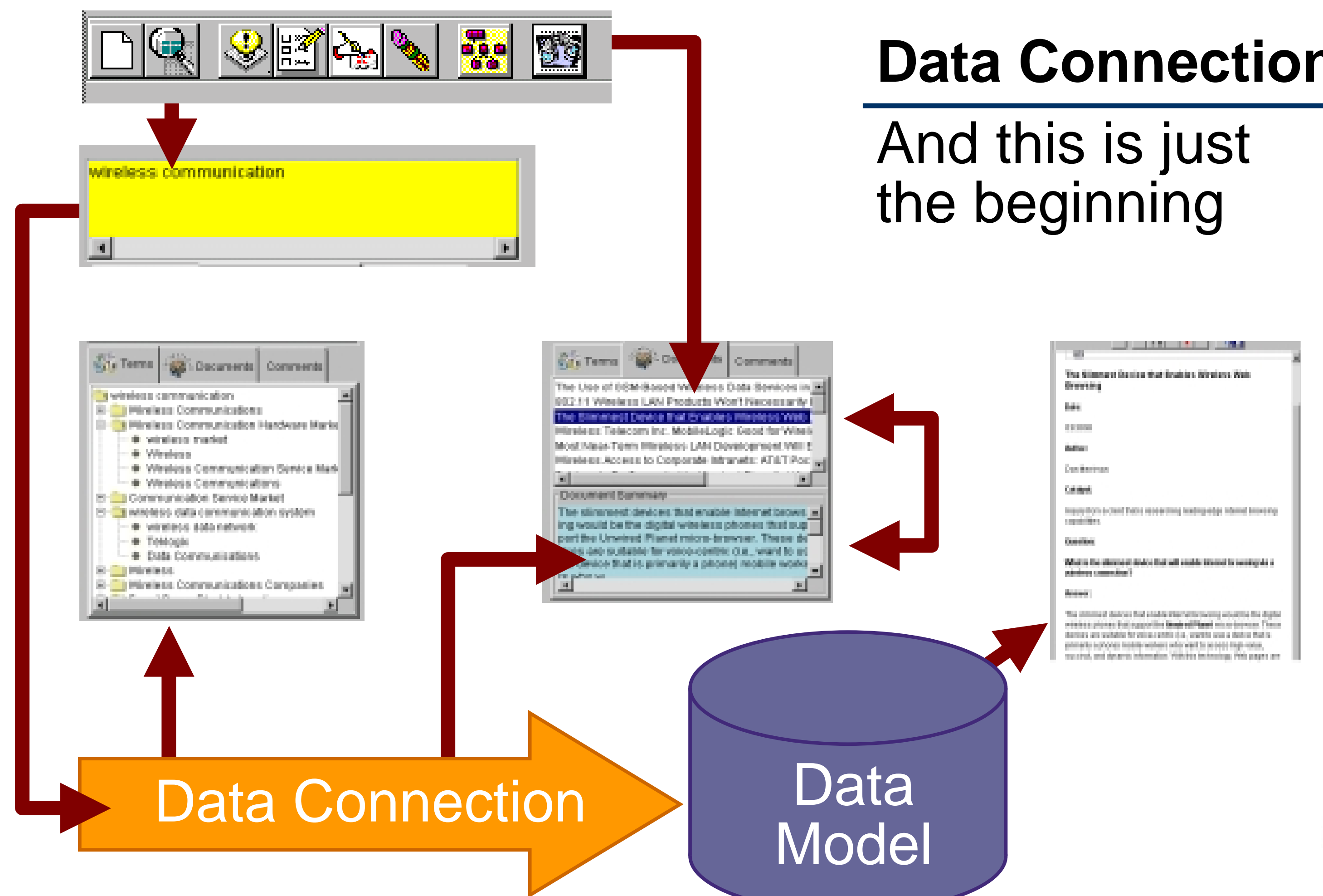
One Scenario

- Toolbar "Search95" is clicked
- Search Execute method gets text from input field
- Pass text to search server
- Gets terms
  - Loads terms into first tab of tabbed window
- Gets documents
  - Loads doc titles into list in 2nd tab

The Following Connections Are Needed Just for the Search



And If We Add Document Viewing…



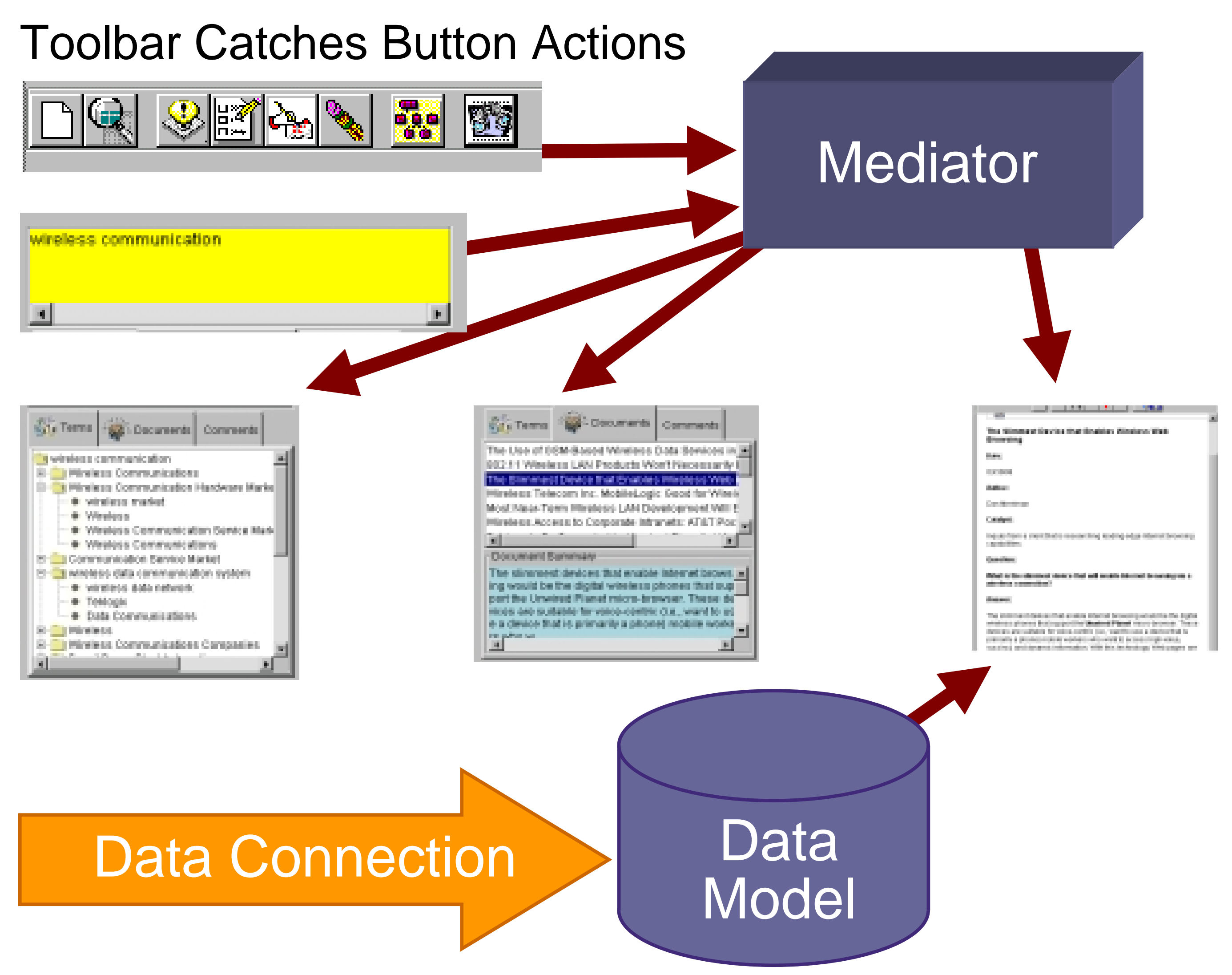
Data Connection

And this is just the beginning

Let's Consider the Mediator Pattern

- Defines how a set of objects interact
- Keeps coupling between objects loose
- Allows interface/method changes without each object needing to be aware of them

Using a Mediator



But How Do We Show a Wait-Cursor?

- The mediator has no window to set the cursor to Wait in
- The Applet window is also an object in this system

Including the Applet

Toolbar Catches Button Actions

wireless communication

Mediator

Data Model

Data Connection

The Mediator Class

```
public class Mediator
{
   DataModel dmodel;    //holds data and communicates with server
   checkText ctx;       //hold current query
   kmfTabbedPane tab;   //holds results
   MarkupHTMLPanel html;//displays documents
   JApplet applet;      //User interface element
   String currentURL;   //current document
   String selectedTerm; //term just clicked on
//----------------------------------------------------------
  public Mediator(DataModel dm, JApplet app)
  {
     dmodel = dm;
     applet  = app;
  public void registerTabPanel(kmfTabbedPane tb) {tab=tb;}
  public void registerText(checkText tx)   {ctx=tx;}
  public void registerHTMLPane(MarkupHTMLPanel htm)  {html = htm}
//-------------------------------------------------
  public void clear()   {
     ctx.setText("");
     dmodel.clearData();
     tab.clearPanels();
  }
//-------------------------------------------------
  public void search()
  {
     applet.setCursor(Wait);
     searchResult[] src = dmodel.searchContext(getText());
     tab.loadTree(src);
     String[] titles = dmodel.searchDocs(getText());
     tab.loadList(titles);
     applet.setCursor(Default);
  }
```

The Toolbar Class

```
public class knToolbar extends JToolBar implements ActionListener
{
   kmfToolButton clear, search, explain, markup, addTerm, removeTerm,
plot, view;

   public knToolbar(JApplet app, Mediator md)
   {
    //clear button
    add(clear = new ClearButton(app, this, md));
    //search button
    add(search = new SearchButton(app, this, md));
    //separate buttons
    addSeparator();
    //explain button
    add(explain = new ExplainButton(app, this, md));
   //markup button
    add(markup = new MarkupButton(app, this, md));
   //add and remove buttons
    add (addTerm = new AddTermButton(app, this, md));
    add (removeTerm = new RemoveTermButton(app, this, md));
    //separate buttons
    addSeparator();
    //plot  related terms
    add (plot = new PlotButton(app, this, md));
    //separate buttons
     addSeparator();
    //view URL
    add (view = new ViewButton(app, this, md));
}
```

Then the Action Routine Is Simplified to This:

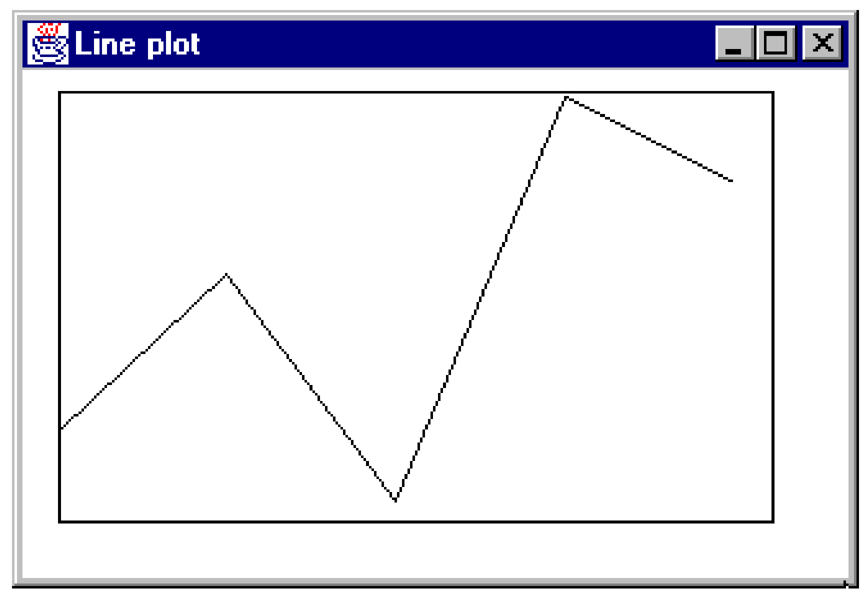
- You don’t have to know which command it is to execute it

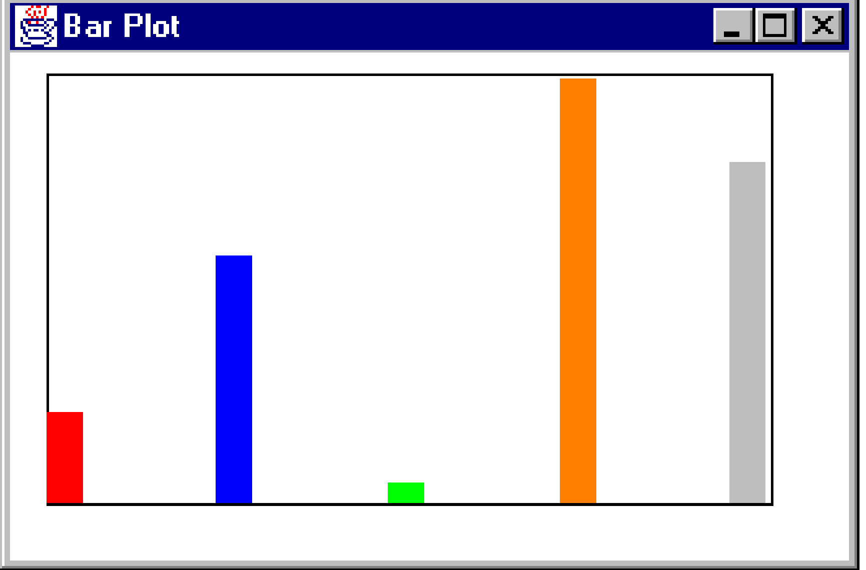
```
//simple actionPerformed method using command objects
public void actionPerformed(ActionEvent evt)
{
//get the object which caused the event
  Command obj = (Command)evt.getSource();
  obj.Execute();        //execute that command
}
```

The Template Pattern

- Occurs whenever you
  - Write a parent class
  - Leave some methods to be implemented in derived classes
- Formalizes the idea of defining an algorithm at the base level
  - And leaving details for the subclasses

Suppose We Have Two Plot Classes





Both Plots Require Similar Basic Methods

- But the plot method will vary
  - public void setSize(Dimension sz)
  - public void setPenColor(Color c)
  - public void findBounds()
  - public abstract void plot(x[], y[]);

In Our Simplest Case…

- The Template pattern simply says
- Put the common code in the base class and
- Put the unique code in each derived class

Kinds of Methods in Templates

- Concrete methods in base class
- Abstract methods filled in later
- Methods with a default implementation that subclasses may override
  – Known as Hook methods
- Methods which call a sequence of abstract, hook and concrete methods
  – These define an algorithm
  – Referred to as Template methods

Drawing a Triangle

```
public abstract class Triangle {
  private Point p1, p2, p3;
  //---------------------------------------
  public Triangle(Point a, Point b, Point c)   {
     //save
     p1 = a; p2 = b; p3 = c;
  }
  //---------------------------------------
  public void draw(Graphics g)   {
   //This Template methods draws a general triangle
     drawLine(g, p1, p2);
     Point current = draw2ndLine(g, p2, p3);
     closeTriangle(g, current);
  }
  //---------------------------------------
  public void drawLine(Graphics g, Point a, Point b)   {
     g.drawLine(a.x, a.y, b.x, b.y);
  }
  //---------------------------------------
  //this routine is the "Hook" that has to be implemented for each triangle type.
  abstract public Point draw2ndLine(Graphics g, Point a, Point b);
  //---------------------------------------
  public void closeTriangle(Graphics g, Point c)
  {
     //draw back to first point
     g.drawLine(c.x, c.y, p1.x, p1.y);
  }
}
```

The Standard Triangle

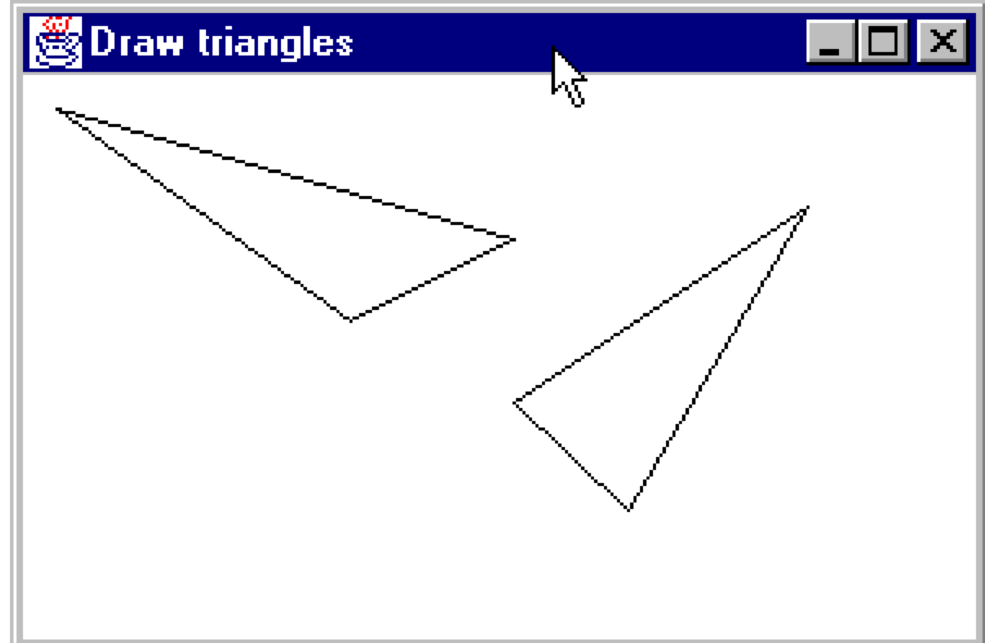
```
public class stdTriangle extends Triangle
{
   public stdTriangle(Point a, Point b, Point c)
   {
    super(a, b, c);
   }
   public Point draw2ndLine(Graphics g, Point a, Point b)
   {
      g.drawLine(a.x, a.y, b.x, b.y);
      return  b;
   }
}
```

The Isoceles Triangle

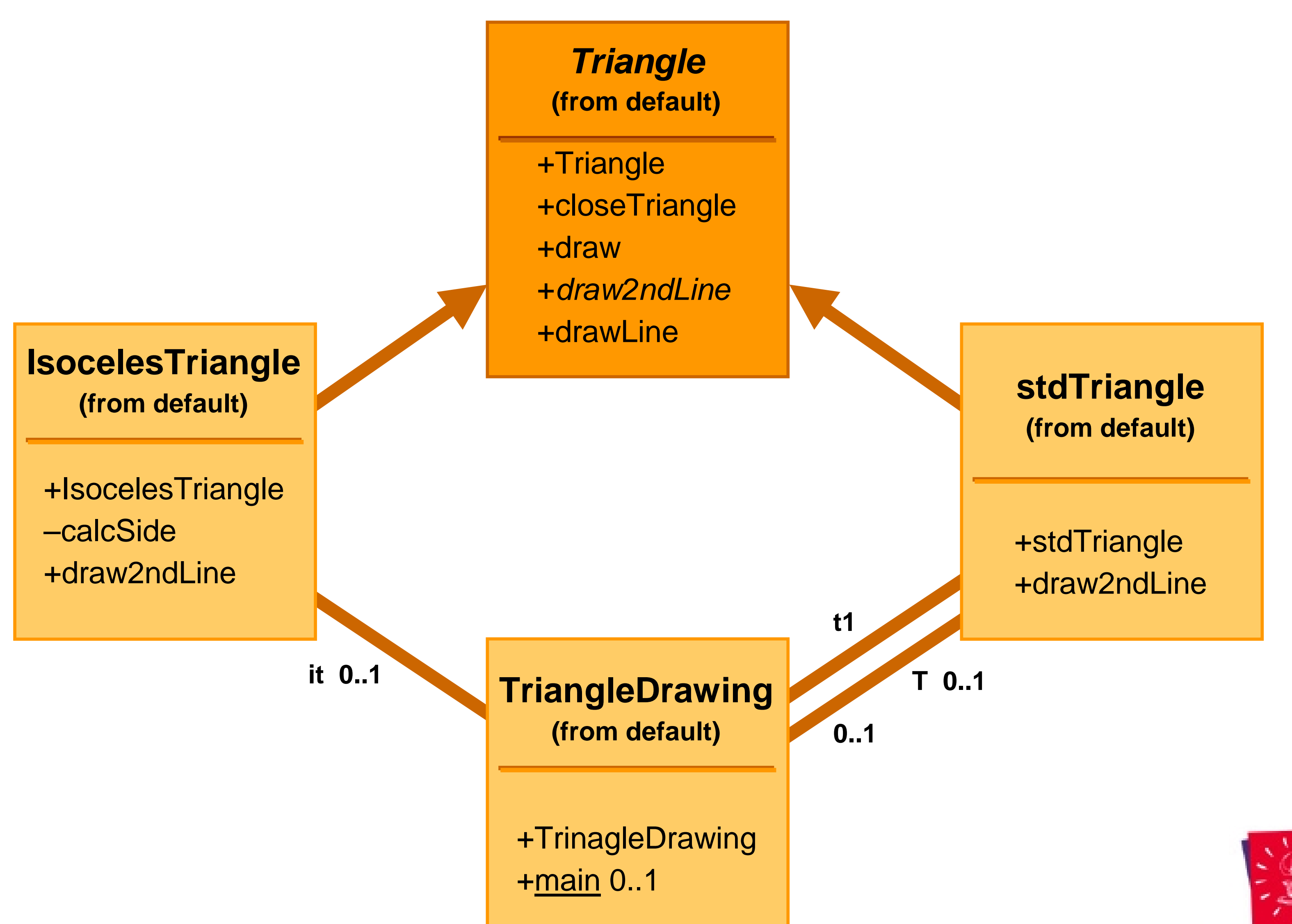
```
public class IsocelesTriangle extends Triangle
{
   Point newc;
   int newcx, newcy;
   int incr;

   public IsocelesTriangle(Point a, Point b, Point c)
   {
      super(a, b, c);
      double side1 = calcSide(dx1, dy1);
      double side2 = calcSide(dx2, dy2);

      if (side2 < side1) incr = -1; else incr = 1;

      double slope = dy2 / dx2;
      double intercept = c.y - slope* c.x;

      //move point c so that this is an isoceles triangle
      newcx = c.x; newcy = c.y;
     //....
      newc = new Point(newcx, newcy);
   }
   //---------------------------------------
   //calculate length of side
   private double calcSide(double dx, double dy)
   {
      return Math.sqrt(dx*dx + dy*dy);
   }
   //---------------------------------------
   //draws 2nd line using saved new point
   public Point draw2ndLine(Graphics g, Point b, Point c)
   {
    g.drawLine(b.x, b.y, newc.x, newc.y);
    return newc;
   }
}
```

Derived Classes of Triangle

- All implement draw2ndLine differently
  - Standard triangle
  - Isoceles triangle
  - Right triangle



Template Class Diagram



JavaOne

Consider the Problem of Indexing Web Pages

- HTML writers are devilishly clever and diverse
- You never know what you are going to find in the HTML source code
- Suppose we want to index a set of HTML documents and save their titles in a table

From the IBM Patent Server

```
<HTML><HEAD>
<META NAME="TITLE" content="Software version management system">
<META NAME="PUBDATE" content="12/10/1985">
<META NAME="ABSTRACT" content="A software version management
system, also called system modeller, provides for automatically collecting
and recompiling updated versions of component software objects
comprising a software program for operation on a plurality of personal
computers c">

<TITLE>Patent Server: 4558413 Detailed View </TITLE>
<META NAME="owner" CONTENT="patserv@almaden.ibm.com">
</HEAD>
```

From the JDK™ 1.2 Release

```
<html>
<head>
<title>Untitled Document</title>
<meta http-equiv="Content-Type" content="text/html; charset=iso-8859-1">
</head>

<body bgcolor="#FFFFFF">
<p><font color="#0000FF"><a href="../servlet/CookieExample">
<img src="../images/execute.gif" align="right" border="0"></a>
<a href="index.html"><img src="../images/return.gif" width="24"
height="24" align="right" border="0"></a></font></p>
<h3>Source Code for Session Example<font color="#0000FF"><br>
  </font> </h3>
```

From the JavaServer Pages™ (JSP) Development Kit

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0//EN">
<!--NewPage-->
<HTML>
<HEAD>
<!-- Generated by javadoc on Tue Apr 20 23:19:54 PDT 1999 -->
<TITLE>
Java Servlet API Documentation: Class  Cookie
</TITLE>
<LINK REL ="stylesheet" TYPE="text/css" HREF="../../../stylesheet.css"
TITLE="Style">
</HEAD>
<BODY BGCOLOR="white">
```

Document Titles Vary in Their Location

- Locations we have just seen
  - Inside a <META tag
  - Inside a <TITLE> tag
  - Inside an <H3> tag
- How to write an indexing program to handle all these cases
- First we write a base class

A Base HTMLDoc Class

```
public abstract class HTMLDoc {
    InputFile fl;      //local file variable

 //constructor
 public HTMLDoc(String filename) {
   fl = new InputFile(filename);
 }

 //read next line from file
 public String getNextLine() {
    return fl.readLine();
 }

 //get title in various ways
 public abstract String getTitle();
}
```

Parsing a Patent Server Document

```
public class PatentDoc extends HTMLDoc {
    //documents from IBM Patent server
 public PatentDoc(String filename) {
   super(filename);
 }
 //In Patent server documents,
 //the title is in the first META tag
 public String getTitle() {
     String s="";
     do {
       s = getNextLine();
     } while (s.indexOf ("<META") < 0  );
     int index = s.indexOf ("content=") + 9;
     s = s.substring (index);
     int last = s.indexOf ("\"");
     s = s.substring (0, last);
     return s;
 }
}
```

A Calling Program

```
public class Titler {
    public Titler() {
        HTMLDoc ht; //instance of base class
    //print out titles from 3 different HTML doc types
        ht = new PatentDoc("4558413.html");
        System.out.println(ht.getTitle());

        ht = new JavaDoc("Cookie.html");
        System.out.println(ht.getTitle());

        ht = new HowTo("sessions.html");
        System.out.println(ht.getTitle());
    }
    //------------------------------------
    static public void main (String[] argv) {
        new Titler();
    }
}
```

These Illustrate a Template Design Pattern

- Concrete methods are complete in the base class
  - used by all the derived classes (readLine)
- Abstract methods are not implemented at all in the base class
  - Each derived class must provide and implementation
- Hook methods have a default implementation in the base class but are expected to be overridden

Example of Concrete Method

```
//read next line from file
 //This is a concrete method
 public String getNextLine() {
     return fl.readLine();
 }
```

Template Pattern (Cont.)

- Template methods
  - Call a combination of
    - Concrete methods
    - Abstract methods
    - Hook methods

Example of a Template Method

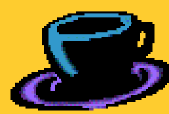
- Remove extra spaces between title words

```
//get words in title without extra spaces
//This is a template method
public String getCompactTitle() {
    StringTokenizer tok = new StringTokenizer(getTitle());
    Vector words = new Vector();
    while(tok.hasMoreTokens ()) {
        words.addElement(tok.nextToken ());
    }
    String newTitle ="";
    for(int i=0; i< words.size(); i++)
        newTitle += (String)words.elementAt (i) + " ";
    return newTitle;
}
```

A Template Method

- Calls an abstract method
- The implementation in the derived class is called
  - Apparently from the base class
- This is called the “Hollywood Principle”
  - “Don’t call us, we'll call you”

Design Patterns You Use Every Day



Java™ Technology-based Design Patterns—Part II

Some Design Patterns Are Very Common

- Iterator
  - For moving through an arbitrary collection of objects
- Template
  - Provides an abstract definition of an algorithm
- Interpreter
  - Describes a grammar for a language and interprets statements in that language
- Singleton
  - Assures that there will be only one instance of a class and provides a global point of access to it

The Iterator Pattern

- Simplest, most frequently used pattern
- Use it to move through a collection of data
  - Without knowing the detailed internal representation of that collection
  - Could also do special filtering and only return certain elements

The Iterator Is Just an Interface

- You can implement it any way you'd like
- Design Patterns proposes the following simple interface:

```
public interface Iterator {
  public Object First();
  public Object Next();
  public boolean isDone();
  public Object currentItem();
}
```

The Iterator of Choice Is the Enumeration Interface

- Enumeration is much like the Iterator interface we just suggested:

```
public interface Enumeration {
  public boolean hasMoreElements();
  public Object nextElement();
}
```

- Note that there is no First() method
- However in the Java programming language we usually get a new instance instead

One Disadvantage in the Java Programming Language

- Since the language is strongly typed
  - You have to cast the results of nextElement()
  - String name = (String)v.nextElement();

Enumeration Is Built Into

- Vector class
- Hashtable class
  - The Hashtable has a second Enumeration keys()

```
Vector vector = new Vector();
//...load vector....
Enumeration e = vector.elements()
//print out contents of Vector
while e.hasMoreElements()
    –{
    –String name = (String)e.nextElement();
    –System.out.println(name);
    –}
```

A Simple Example

```
public class KidData {
  private Vector kids;
  public KidData(String filename) {
    kids = new Vector();
     InputFile f = new InputFile(filename);
     String s = f.readLine();
     while (s != null) {
        kid = new Kid(s);
        kids.addElement(kid);
      }
    public Enumeration elements() {      //we just return the
      return kids.elements();         //enumeration from the Vector
    }
```

Filtered Enumeration

- Return only some elements based on some criteria
- Our Kid object contains kid names and the names of their teams
- We could return only those kids on a particular team

A KidData Filtered Enumeration

```
public class kidClub
   implements Enumeration
{
   String clubMask;      //name of club
   Kid kid;              //next kid to return
   Enumeration ke;       //gets all kids
   KidData kdata;        //class containing kids
//-----------------------------------------
   public kidClub(KidData kd, String club)
   {
      clubMask = club;      //save the club
      kdata = kd;           //copy the class
      kid = null;           //default
      ke = kdata.elements();  //get Enumerator
   }
```

The Methods…

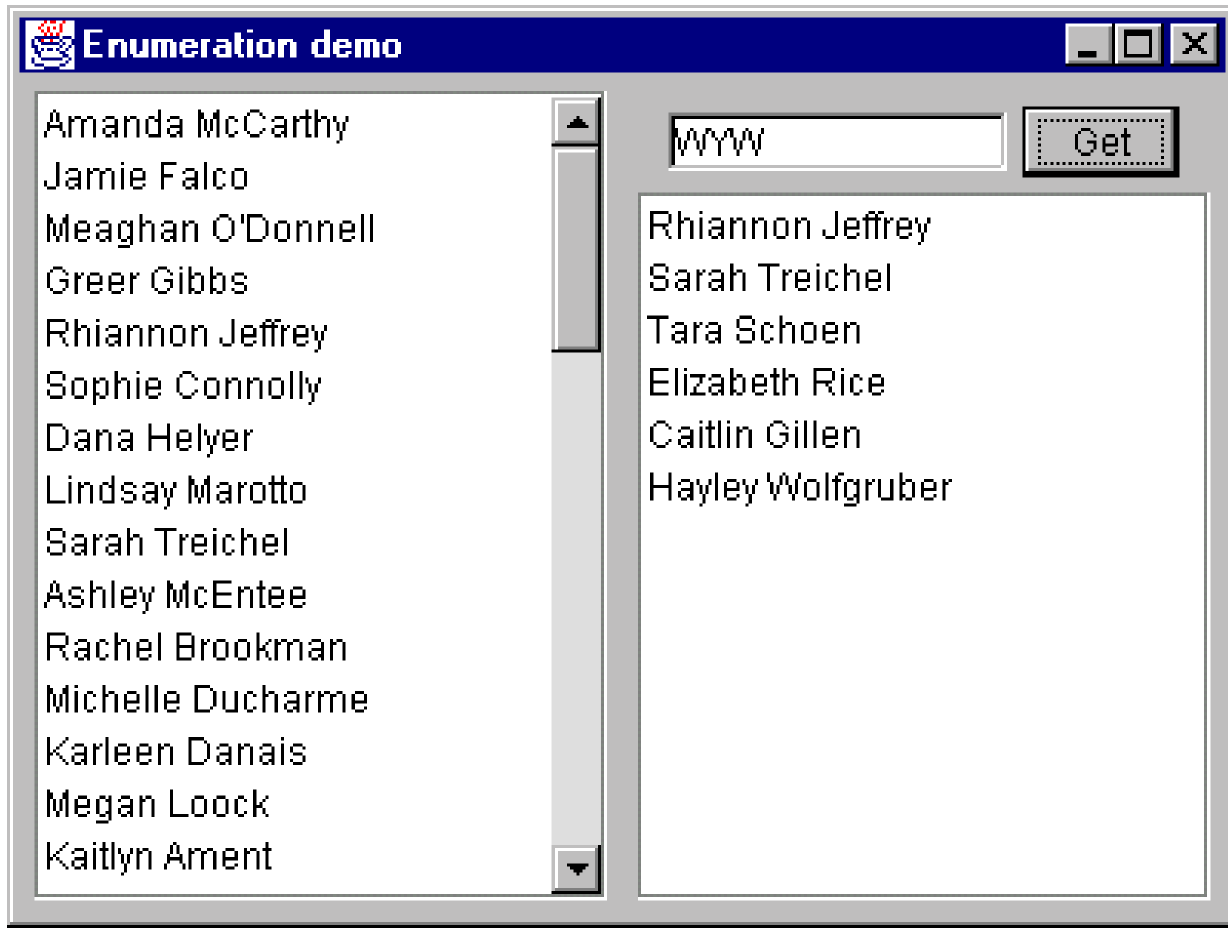
```
 public boolean hasMoreElements()
   {
   //return true if there are any more kids
   //belonging to the specified club
   boolean found = false;
   while(ke.hasMoreElements() && ! found)
      {
      kid = (Kid)ke.nextElement();
      found = kid.getClub().equals(clubMask);
      }
   if(! found)
      kid = null;    //set to null if none left
   return found;
   }
//-------------------------------------
public Object nextElement()
   {
      if(kid != null)
         return kid;
      else
         //throw exception if access past end
         throw new NoSuchElementException();
   }
```

Returns a Filtered Enumeration

- Add to the KidData class

```
public Enumeration kidsInClub(String club)
    {
        return new kidClub(this, club);
    }
```

Example



Consequences of the Iterator Pattern

- What should happen if another thread changes the data while an iterator is active?
- Enumerators need privileged access to the data structure in a class
  - This usually means make some internal methods public

Now Let's Consider Interpreters

- Some programs can benefit from having a language to describe operations they can perform
- Interpreter defines a grammar for that language and how to use that grammar to execute statements in that language
- Simple cases
  - Macro language
  - Using VBA

Recognizing When an Interpreter May Be Helpful

- When the program must parse an algebraic statement
- When the program must produce varying kinds of output

Let's Consider Some Results From a Swim Meet

- Report results sorted by time
- By club
- By last name
- In various column orders

```
1  Amanda McCarthy          12  WCA    29.28
2  Jamie Falco              12  HNHS   29.80
3  Meaghan O'Donnell        12  EDST   30.00
4  Greer Gibbs              12  CDEV   30.04
5  Rhiannon Jeffrey         11  WYW    30.04
6  Sophie Connolly          12  WAC    30.05
7  Dana Helyer              12  ARAC   30.18
8  Lindsay Marotto          12  OAK    30.23
9  Sarah Treichel           12  WYW    30.35
10 Ashley McEntee           12  RAC    30.47
11 Rachel Brookman          12  CAT    30.51
12 Michelle Ducharme        12  LEHY   30.51
51 Katie Duffy              12  MGAT   34.24
```

Define a Simple Grammar

```
Print lname frname club time sortby club thenby time
```

Verbs		Variables
Print		Frname
Sortby		Lname
Thenby		Age
		Club
		Time

Print var [var] [sortby var [thenby var]]

Note There Is No Other Easy Way to Accomplish This

- The order and number of columns is difficult to specify simply in a UI
- A simple language makes more sense
  - Parse the language symbols into tokens
  - Reduce the groups of tokens to actions
  - Execute the actions

How Interpreting Proceeds

- Store data on stack and reduce stack

```
Time
Thenby
Club
Sortby
Time
Club
Frname
Lname
Print
```

We Put Parseobjects on Stack

```
public class ParseObject
{
   public static final int VERB=1000, VAR = 1010, MULTVAR =
1020;
   protected int value;
   protected int type;

   public int getValue() {return value;}
   public int getType() {return type;}
}
```

Parsevar and Parseverb Classes

ParseObject

ParseVerb

ParseVar

Print

Sort

Reducing the Parsed Stack

Verb
Time

Var
Club

MultVar

Verb
SortBy

Verb

Var
Time

Var
Club

MultVar

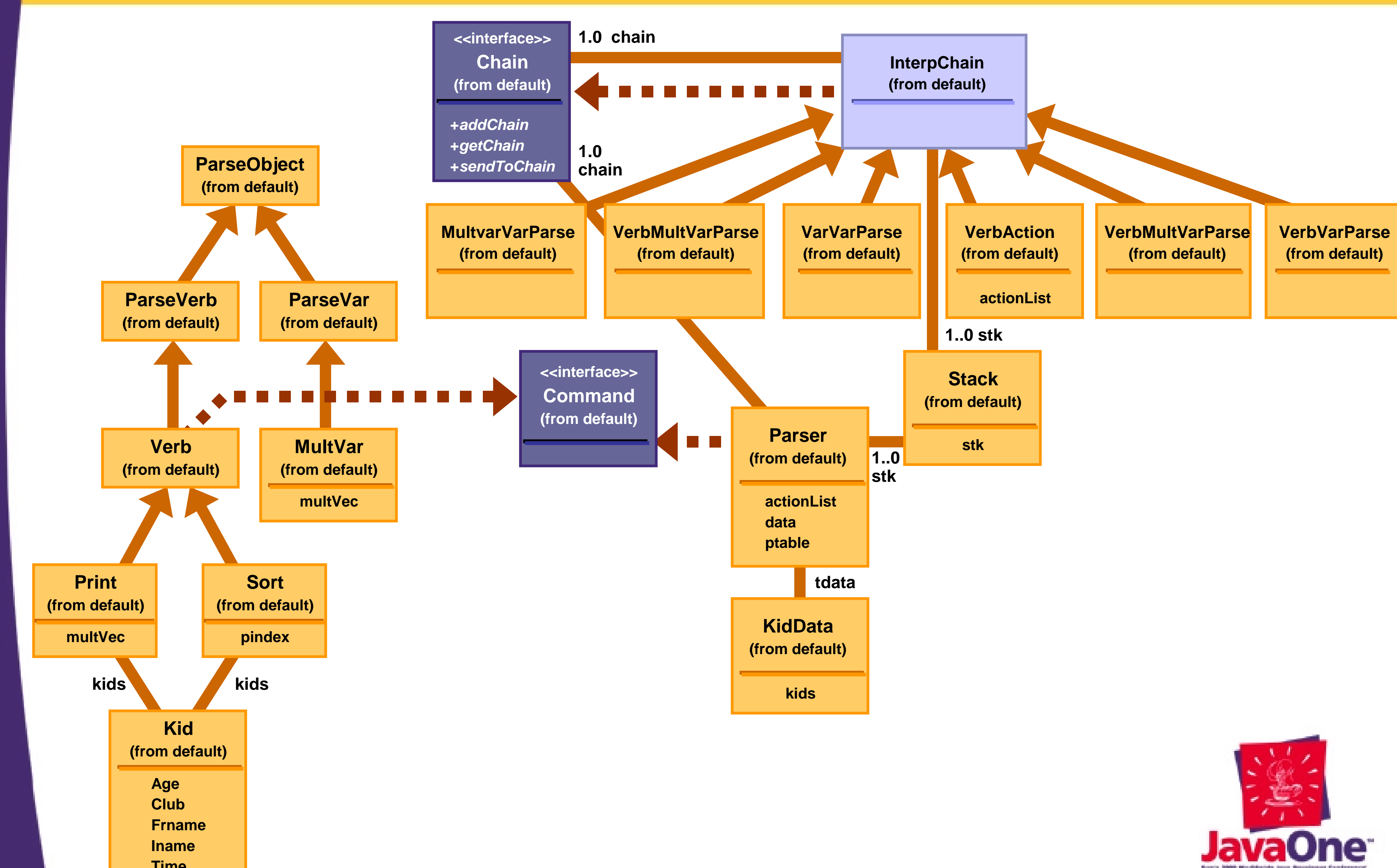
Var
Frname

MultVar

Var
Lname

Verb

The Interpreter Class Diagram



An Interpreter in Action

Interpreter Demo

print lname club time sortby time   Go

McCarthy	WCA	29.28
Falco	HNHS	29.8
O'Donnell	EDST	30.0
Gibbs	CDEV	30.04
Jeffrey	WYW	30.04
Connolly	WAC	30.05
Helyer	ARAC	30.18
Marotto	OAK	30.23
Treichel	WYW	30.35
McEntee	RAC	30.47
Brookman	CAT	30.51
Ducharme	LEHY	30.51
Danais	NES	30.7
Loock	WAC	30.9
Ament	HNHS	30.93
Schoen	WYW	31.01
Olshefski	NCY	31.01

Consequences of the Interpreter Pattern

- You must provide a simple way to enter and edit the language statements
- Extensive error checking can be required
- May want to consider generating a language automatically from UI component states
- Languages are easily extended
- But can eventually become hard to maintain

The Singleton Pattern

- Assures that there is only one instance of a class
- Provides a global point of access to it
  - Window manager
  - Print spooler
  - Database engine
  - Serial ports

Singleton Using Static Method

- Constructor is private
  - Static Instance method only allows one instance

```
public class iSpooler {
   //this is a prototype for a printer-spooler class
   //such that only one instance can ever exist
   static boolean instance_flag = false; //true if 1 instance

   private iSpooler() {}        //private constructor

   static public iSpooler Instance() {
      if (! instance_flag) {  //only allow one instance
         instance_flag = true;
         return new iSpooler();
      }
      else
         return null;          //or return none
   }
```

Advantages of Static Instance Method

- Get a null return if it fails
- Trying to create instances will fail at compile time

Another Approach Is to Throw an Exception

```
public class Printer
   //this is a prototype for a printer-spooler class
   //such that only one instance can ever exist
   static boolean instance_flag=false; //true if 1 instance

   public Printer() throws SingletonException {
      if (instance_flag)
         throw new SingletonException("Only one printer allowed");
      else
         instance_flag=true;     //set flag for 1 instance
      System.out.println("printer opened");
   }
  }
//================================================
class SingletonException extends RuntimeException  {
   //new exception type for singleton classes
   public SingletonException() {
      super();
   }
//------------------------------------------------
   public SingletonException(String s) {
      super(s);
   }
}
```

The javax.comm Package as Singletons

- Controls serial ports for various computers
- Separately downloadable from JDK™ software
- Two possible Singletons
  - One manages collection of ports
  - One manages each port

CommPortIdentifier

- Has static method for returning an enumeration of all the ports in a system

```
Enumeration portEnum = CommPortIdentifier.getPortIdentifiers();
while (portEnum.hasMoreElements()) {
          portId = (CommPortIdentifier) portEnum.nextElement();
          if (portId.getPortType() == CommPortIdentifier.PORT_SERIAL)
             {
             ports.addElement (portId.getName());
             }
      }
```

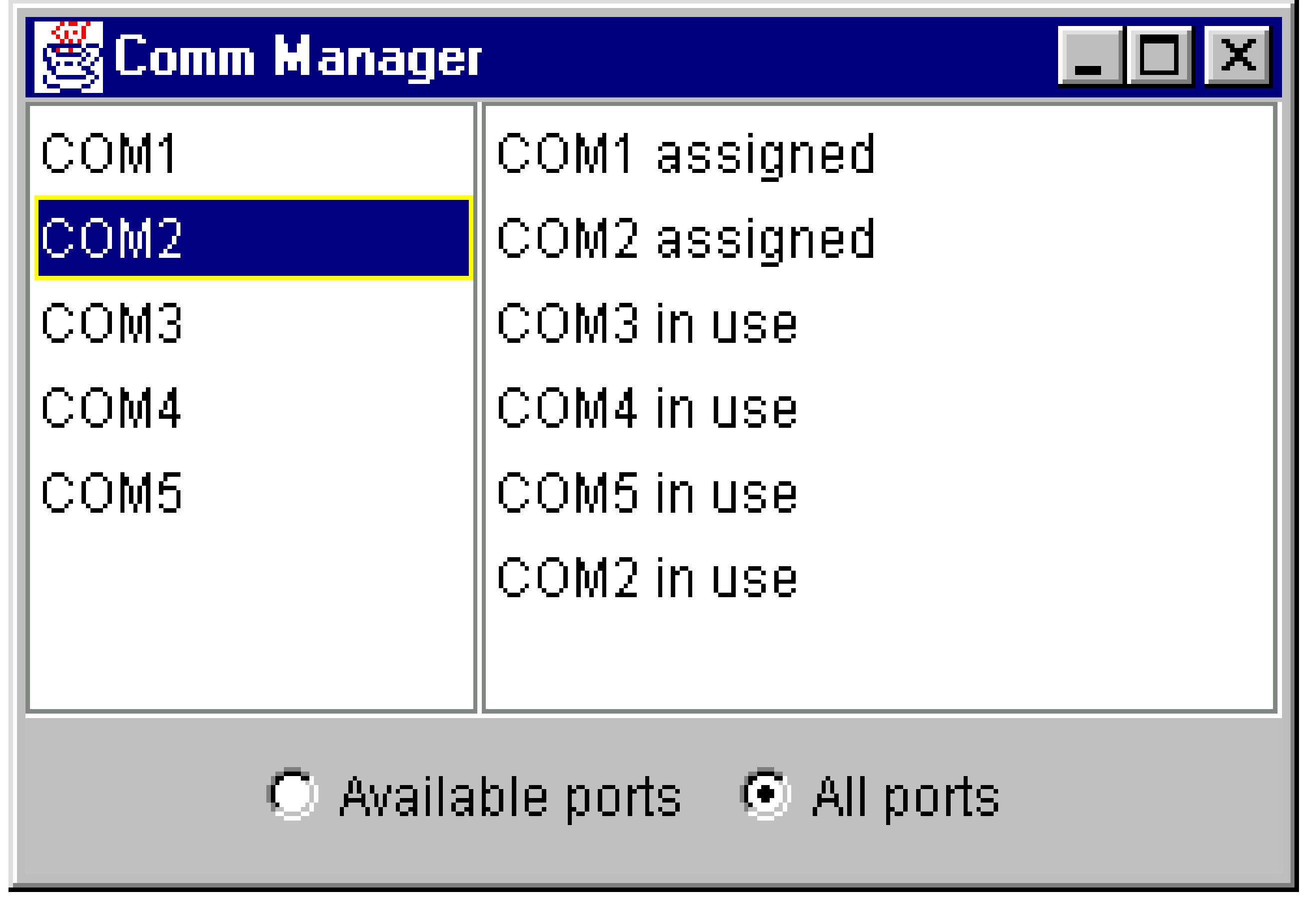
Find Out If Port Is Available

- Attempt to open it
- If it doesn't exist, it throws NoSuchPortException
- If it is in use, it throws PortInUseException

Opening a Port

```
try {
         //try to get port ownership
    portId = CommPortIdentifier.getPortIdentifier(portName);

//if successful, open the port
         CommPort cp = portId.open("SimpleComm",100);

//report success
         portOpenList.addElement(portName);
         cp.close();
         }
      catch(NoSuchPortException e){}
      catch(PortInUseException e){}
```

Getting COM Ports



After This Many Patterns

- You see a general use for this level of abstraction
- You may be listing to port…

Conclusion

- Design Patterns can help you write simpler and more elegant programs
- They provide a level of indirection that keeps classes from having to know about each other’s internal workings
- Many are really ways to communicate between objects effectively
- They have been catalogued and are widely known in the field

References

- Gang of Four
  - Design Patterns, Elements of Reusable Object-Oriented Software
    - Gamma, Helm, Johnson, Vlissides,
    - Addison-Wesley
- Design Patterns Smalltalk Companion
    - Alpert, Brown and Woolf. A-W, 1998.
- Principles of Object Oriented Programming in Java™ 1.1
    - J W Cooper - Coriolis/Ventana
- Java™ Design Patterns: a Tutorial,
    - J W Cooper (A-W), 2000
- Thank you

JavaOne SM
Sun's 2000 Worldwide Java Developer Conference