Object Orientation with Design Patterns

Behavioural Patterns

Command, Flyweight, Proxy Pattern

Behavioural Patterns

- **Behaviour patterns** are concerned with **algorithms** and the **assignment of responsibility** between objects

- Not just concerned with classes and objects but also **communications between them**

- Behavioural patterns **characterize complex control** flow that's difficult to follow at run-time

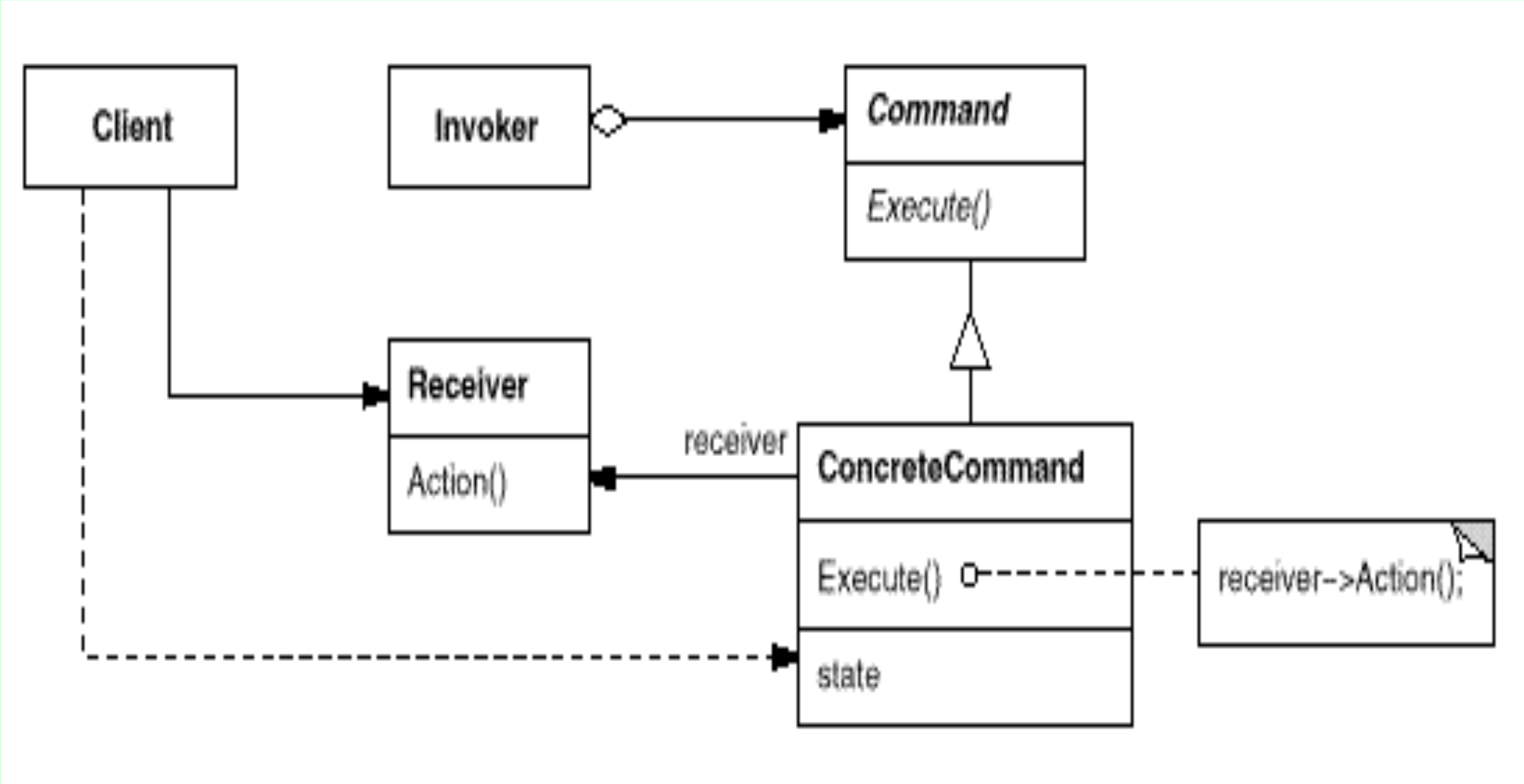
- **Organizing control** within a system can yield **great benefits** in both **efficiency** and **maintainability**

The Command Pattern

- **Intent:**
  Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and **support undoable** operations.

- **AKA:** Action, Transaction

The Command Pattern Structure



The Command Pattern Participants

- **Command**
  - **declares an interface for executing an operation**. This therefore defines the method for the invoker to use.

- **ConcreteCommand**
  - **defines a binding between a Receiver object and an action**.
  - implements the Command interface by invoking the corresponding operation(s) on Receiver.

- **Client**
  - **creates a ConcreteCommand object and sets its receiver**.

The Command Pattern Participants

- **Invoker**
  - **asks the command to carry out the request** e.g. a button or menuItem in a GUI

- **Receiver**
  - **knows how to perform the operations associated with carrying out a request.**
    Any class may serve as a Receiver.
    This is the target of the Command, it represents the object which fulfils the request

Command Pattern Applicability

- **parameterize objects by an action to perform**. Since a request is encapsulated within an object, the request can then be passed as a parameter and manipulated just like any other object, e.g. parameterize a menu with the method calls that correspond to menu labels.

- **specify, queue, and execute requests at different times**. A Command object can have a lifetime independent of the original request. If the receiver of a request can be represented in an address space-independent way, then you can transfer a command object for the request to a different process and fulfill the request there.

- **support undo**. The Command's execute operation can store state for reversing its effects in the command itself.

Command Pattern Applicability

- **support logging changes so that they can be reapplied in case of a system crash**. By augmenting the Command interface with load and store operations, you can keep a persistent log of changes.

- **structure a system around high-level operations built on primitives operations**, i.e. build command which are composed of sequences of command. Such a structure is common in information systems that support transactions.

The Command Pattern - Example

- When you build a Java user interface, you provide controls – menu items, buttons, check boxes, and so on – to **allow the user to interact with the application.**

- When the user selects one of these controls, the application **receives an ActionEvent** which it must trap by implementing the **actionPerformed method** of the ActionListener interface.

- Suppose that we build a very simple program that allows us to select the menu items File | Open and File | Exit and click on a button labelled Red that turns the background of the window red.

The Command Pattern

The Command Pattern

- The program consists of the File Menu object with the **mnuOpen** and **mnuExit MenuItems** added to it. It also contains one button called **btnRed**.

- Clicking any of these causes an **ActionEvent**, which **generates a call to the *actionPerformed* method** which might look as follows:

```
public void actionPerformed(ActionEvent e)
{
    Object ob = e.getSource();
    if (obj == mnuOpen)
        fileOpen();
    else if (obj == mnuExit)
        exitClicked();
    else if (obj = btnRed)
        redClicked();
}
```

The Command Pattern

- Below are the three methods that are called from the *actionPeformed* method.

```
private void exitClicked(){
    System.exit(0);
}

private void fileOpen(){

    FileDialog fDlg = new FileDialog(this,
                         "Open a file", FileDialog.LOAD);
    fDlg.show();
}

private void redClicked(){
    p.setBackground(Color.red);
}
```

The Command Pattern

- As long as there are only a few menu items and buttons, this approach works fine, but when there are **dozens of menu items** and several buttons, the ***actionPerformed*** **method code can get pretty unweildly.**

- This also seems a little inelegant, since, when using an OO language such as Java it is best to **avoid a long series of if statements** to identify the selected object.

- Instead of multiple if statements it's possible to use **polymorphism to call the appropriate action/command** based on the subtype of the source of the ActionEvent

- But **polymorphism requires the implementation of a common interface among subclasses**...this is where the **abstract Command** participant of the **Command Pattern comes into play**

Command Objects

- The **abstract Command class (or Java interface structure) declares the interface for executing operations**. The Command pattern thus fixes the signature of an operation and lets classes vary.
- To apply the Command pattern each **concrete command implements the Command interface**
- A **Command object** always has an ***execute* method** that is called when an action occurs on the object. In it's simplest form a Command object implements at least the following interface:

```
public interface Command {
  public void execute();
}
```

Command Objects

- Using this interface it's possible to reduce the *actionPerformed* code to the following:

```
public void actionPerformed(ActionEvent e) {
    Command obj = (Command)e.getSource();
    obj.execute();
}
```

Details of command hidden from client, dynamic binding ensures correct execute command called

- Each **subclass** of the abstract Command **implements it's own version of the same execute method**. The client no longer needs to contain the details of how to handle each object type.

Building Command Objects

- One way to build a command object is to **derive/extend new classes from the MenuItem and Button classes** and implement the Command interface in each. The following are examples:

```java
class btnRedCommand extends Button implements Command
{
   public btnRedCommand(String caption)
   {
      super(caption);
   }
   public void execute()
   {
      p.setBackground(Color.red);
   }
}
```

Implement the Command i/f and are therefore Commands

Building Command Objects

```
class fileOpenCommand extends MenuItem implements Command
{
   public fileOpenCommand(String caption)
   {
      super(caption);
   }
   public void execute()
   {
      FileDialog fDlg=new FileDialog(fr,"Open file");
      fDlg.show();
   }
}

class fileExitCommand extends MenuItem implements Command
{
   public fileExitCommand(String caption)
   {
      super(caption);
   }
   public void execute()
   {
      System.exit(0);
   }
}
```

Command Pattern

- Simply replace our normal MenuItems and Buttons with the new Command Objects and the correct execute method will then be invoked.

```
mnuOpen = new fileOpenCommand("Open...");
mnuFile.add(mnuOpen);
mnuExit = new fileExitCommand("Exit");
mnuFile.add(mnuExit);

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

btnRed = new btnRedCommand("Red");
p = new Panel();
add(p);
p.add(btnRed);

btnRed.addActionListener(this);
setBounds(100,100,200,100);
setVisible(true);
```

Command Objects

- Using this simple approach has certainly made the **actionPerformed method a lot less complicated**

- However a **lot of new classes are created**!!! In fact a new subclass of JMenuItem would exist for every command, each with an execute method

- The element that causes the action (e.g. MenuItem and Button, the invoker) and the Command objects are **tightly coupled**

- The Command pattern should separate the invoker from the concrete command object, **this reduces coupling**

- This can be achieved by making the **invokers *containers*** for the concrete command objects, i.e. recall "favor object composition over inheritance"

Command Objects

- It's obvious that improvements can be made to our current system

- Suppose the following interface was created, the aim of this interface is to decouple the invoker and command objects:

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

- The UI components can now implement this interface so that a **command can be associated with the particular UI component using the *setCommand(Command)* method**

- The command **associated with the UI component can be retrieved using the *getCommand()* method**.

Command Objects

```
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;
  }
}
```

Invoker now contains a command

Command Objects

- Now the flexibility of our program has been increased....instead of a new component subclass for every type of command, it’s possible to set the command associated with a component at runtime...e.g.

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

- **Note: The *this* reference is passed so that it’s possible to manipulate other components contained in the client** e.g. change background color etc. (thus the receiver is the client itself in this example)

Command Objects

- Separate command objects are still created but now they are seperated from the UI components (invokers)

```
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();
  }
}
```

Command Objects

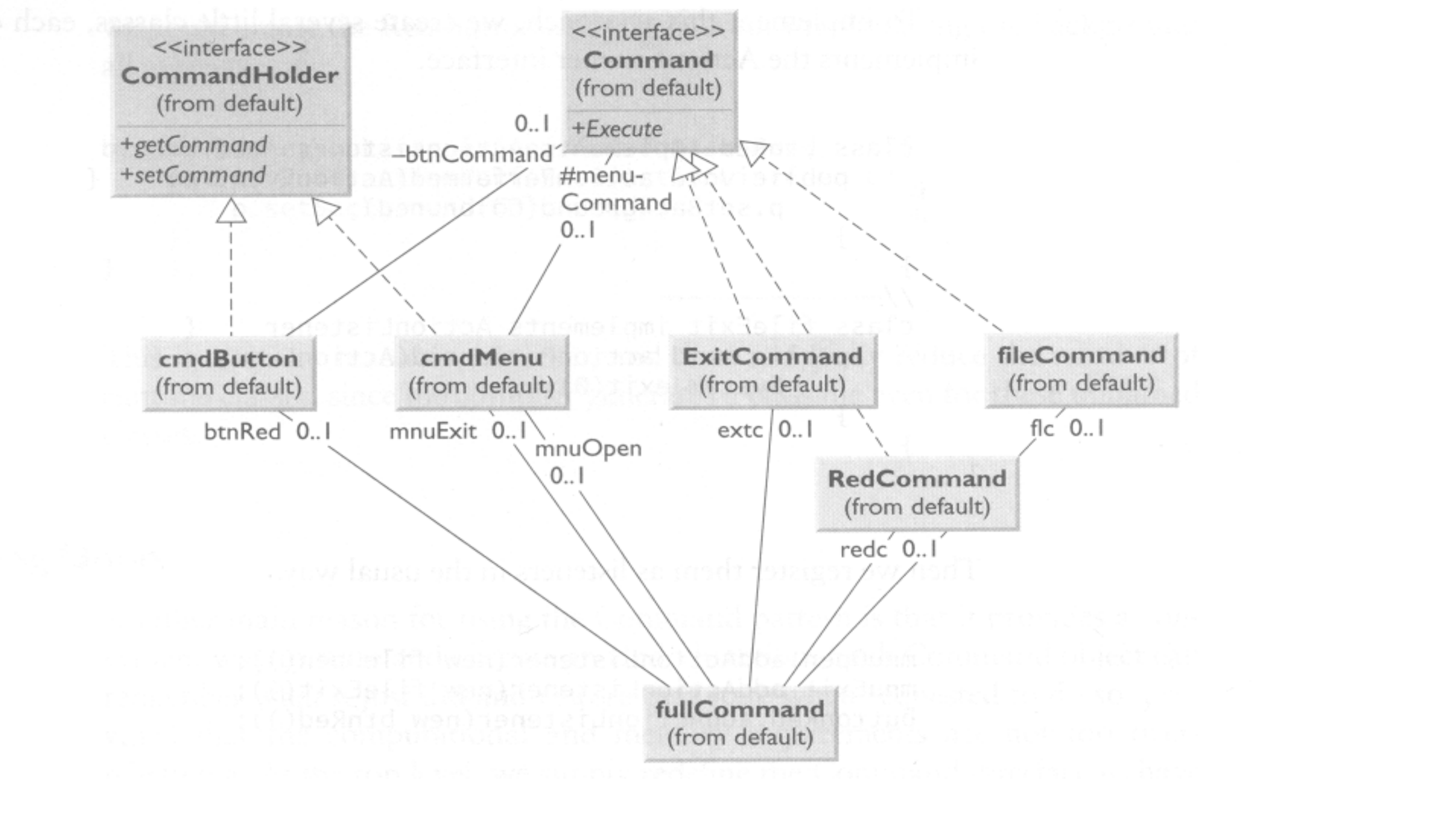
- Now the actionPerformed method has one extra process to carry out, i.e. retrieve the command associated with the source of the ActionEvent

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

- **The cost of this one extra processing step is offset by the flexibility afforded by decoupling the invoker from the command**....why???

- Consider adding a new command to this version of the GUI (new command subclass) and compare that to the first version of our GUI (new component subclass implementing Command inter/face).

Command Objects



Command Implementation

- Given that the **command pattern** can **encapsulate** a command as an object successfully...how useful is this???
- One of the most obvious applications of this pattern is the provision of **undo operations**
- Let’s have a look at **an example** of undo using the command pattern.....

Undo Commands

- Since the Commands are objects they are **capable of storing state information**

- This information can be **stored prior** to executing a command

- To undo the command the **receiver object** is **restored** to it's **previous state**

- To implement this an undo method is added to the command interface

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

Undo Command Example

- In this example the GUI allows the user to **draw successive blue and red lines into a paintPanel object** (subclass of JPanel)

- The **last draw command** can be **undone** by hitting the undo button, hitting the undo command repeatedly will undo multiple command

- To keep track of the commands executed, and the order the commands were executed, **a list of executed commands is maintained (implemented as a Vector)**

- Each time a button is clicked the **corresponding command is added to the vector**

Undo Command Example

```
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
  }
```

In the GUI it makes no sense to undo the undo since it would be impossible to determine what action we need to reverse

Undo Command Example

```
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
   }
 }

 public void unDo() {    //does nothing just to keep the example simple
 }
}
```

When the undo is executed the last command is retrieved and the undo of the command object is invoked

Undo Command Example

- The *actionPerformed* method must now retrieve the command associated with the invoker but also store the command into the undoCommand object

```
public void actionPerformed(ActionEvent e) {
    CommandHolder cmdh = (CommandHolder)e.getSource ();
    Command cmd = cmdh.getCommand ();
    u_cmd.add (cmd);  //add to list
    cmd.execute();   //and execute
}
```

Undo Command Example

- Since the command classes in this example share very similar functionality **a base class called drawCommand** will generalise a drawing command

- The **drawCommand** class can be **extended for specific drawing specialisations** e.g. blueCommand or redCommand

- The **drawCommand** class **implements the command interface**, i.e. the general execute and undo methods

- The specialized classes redCommand and blueCommand differ in color and the side of the panel they begin drawing (red draws left to right and blue draws right to left)

Undo Command Example

- The execute method of the drawCommand stores data associated with each draw command in a vector, this is the data that will be used to undo a particular draw command

- The x and y location of the drawing is incremented so that the lines move across the screen (negative dx for blue)

```
public void execute() {
    drawList.add(new drawData(x, y, dx, dy));
    x += dx;    //increment to next position
    y += dy;
    p.repaint(); //paintPanel calls draw for red and blue
}
```

Undo Command Example

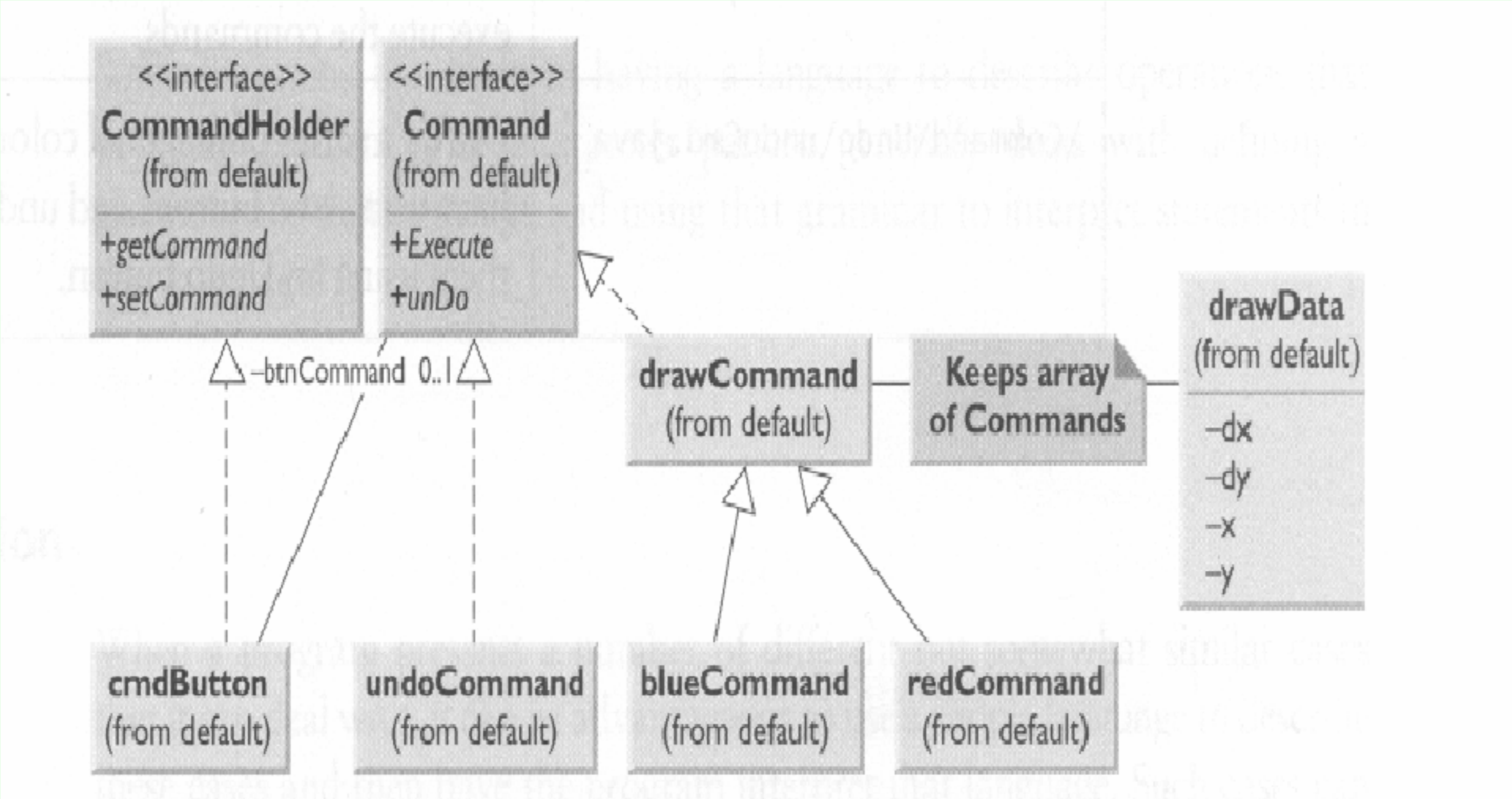
- The *unDo* method of the drawCommand removes the last stored line from it’s list and resets the x and y values

- This *unDo* method is called when the undo button is pushed and the undoCommand *execute* method is called

```
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);
      x = d.getX ();  //x value set to before the line was drawn
      y = d.getY (); //y value set to before the line was drawn
    }
    p.repaint(); //repaint the panel less the undone line
}
```

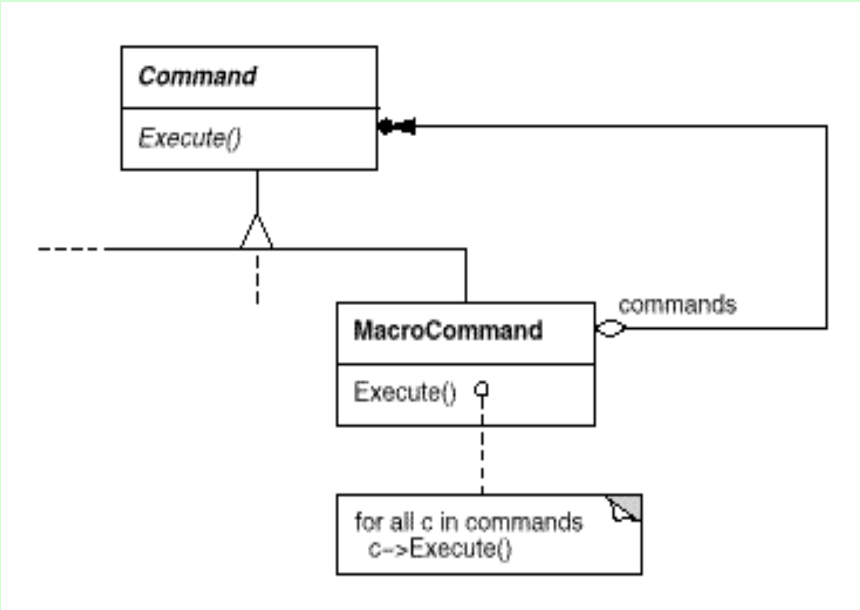
The drawData class makes storage of drawing data cleaner

Undo Command Example



Sequential Commands

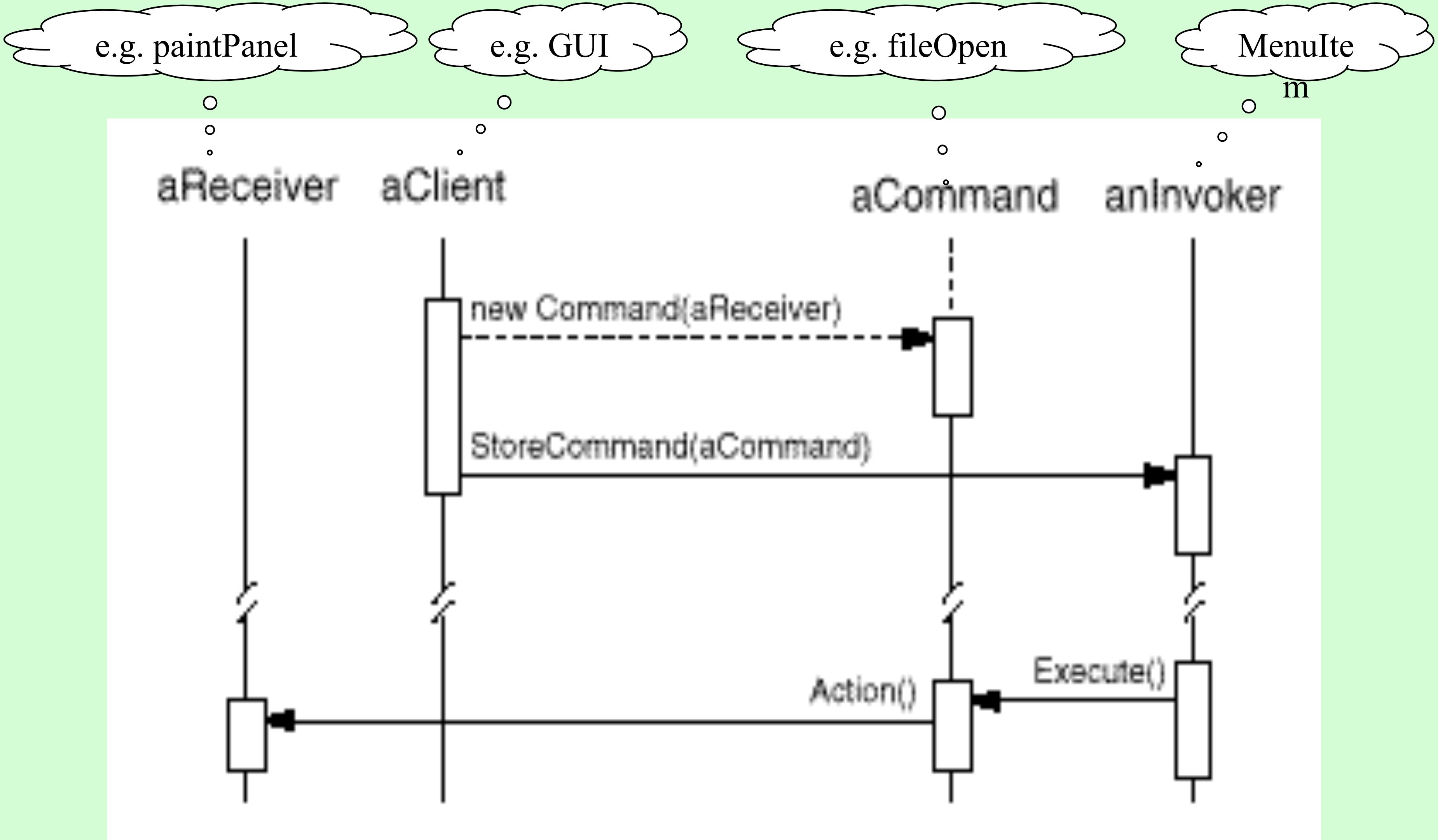
- Often a **command** may be composed of ***a sequence* of commands**
- For example the Command class might have a subclass called MacroCommand; MacroCommand is a concrete Command subclass that simply executes a sequence of Commands.
- The composite pattern can be applied



Command Collaborations

- The client **creates a ConcreteCommand object** and specifies its receiver.
- An Invoker object stores the **ConcreteCommand object.**
- The **invoker** issues a request by **calling Execute** on the command. When commands are undoable, ConcreteCommand stores state for undoing the command prior to invoking Execute.
- The ConcreteCommand object **invokes operation(s)** on its receiver to **carry out the request.**

Command Collaborations



Command Consequences

- Command **decouples** the object that **invokes the operation from** the one that **knows how to perform it.**
- **Commands are first-class objects**. They can be manipulated and extended like any other object.
- You can assemble commands into a **composite command**. An example is the MacroCommand class described earlier.
- It's easy to add **new Commands**, because you don't have to change existing classes.

Command Consequences

- It's possible to share **Command instances** between several objects

- Allows replacement of Commands and/or Receivers at runtime

- The Command patterns **main disadvantage** is the proliferation of **little classes** that either **clutter up** the **main class** if implemented as inner classes or clutter up the **program namespace** if there are **outer classes**.

Web Site Examples - Command

- http://www.codeproject.com/Articles/9415/Distributed-Command-Pattern-an-extension-of-comman

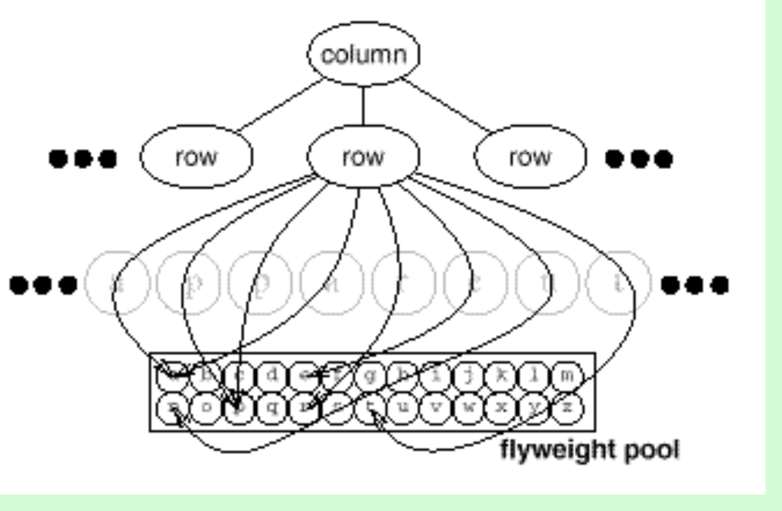
The Flyweight Pattern

- **Intent:**
  Use sharing to support **large numbers** of fine-grained objects efficiently.

Flyweight Motivation

- Flyweights model concepts or entities that are normally too plentiful to represent with objects.

- For example, a document editor can create a flyweight for each letter of the alphabet. Each flyweight stores a **character code**, but its **coordinate position** in the document and its **typographic style** can be determined from the text layout algorithms and formatting commands in effect wherever the character appears. The character code is **intrinsic** state, while the other information is **extrinsic.**



Extrinsic versus Intrinsic state

- A **flyweight** is a **shared object** that can be used in multiple contexts simultaneously.

- The flyweight acts as an **independent object** in each context—it's indistinguishable from an instance of the object that's not shared. Flyweights cannot make assumptions about the context in which they operate.

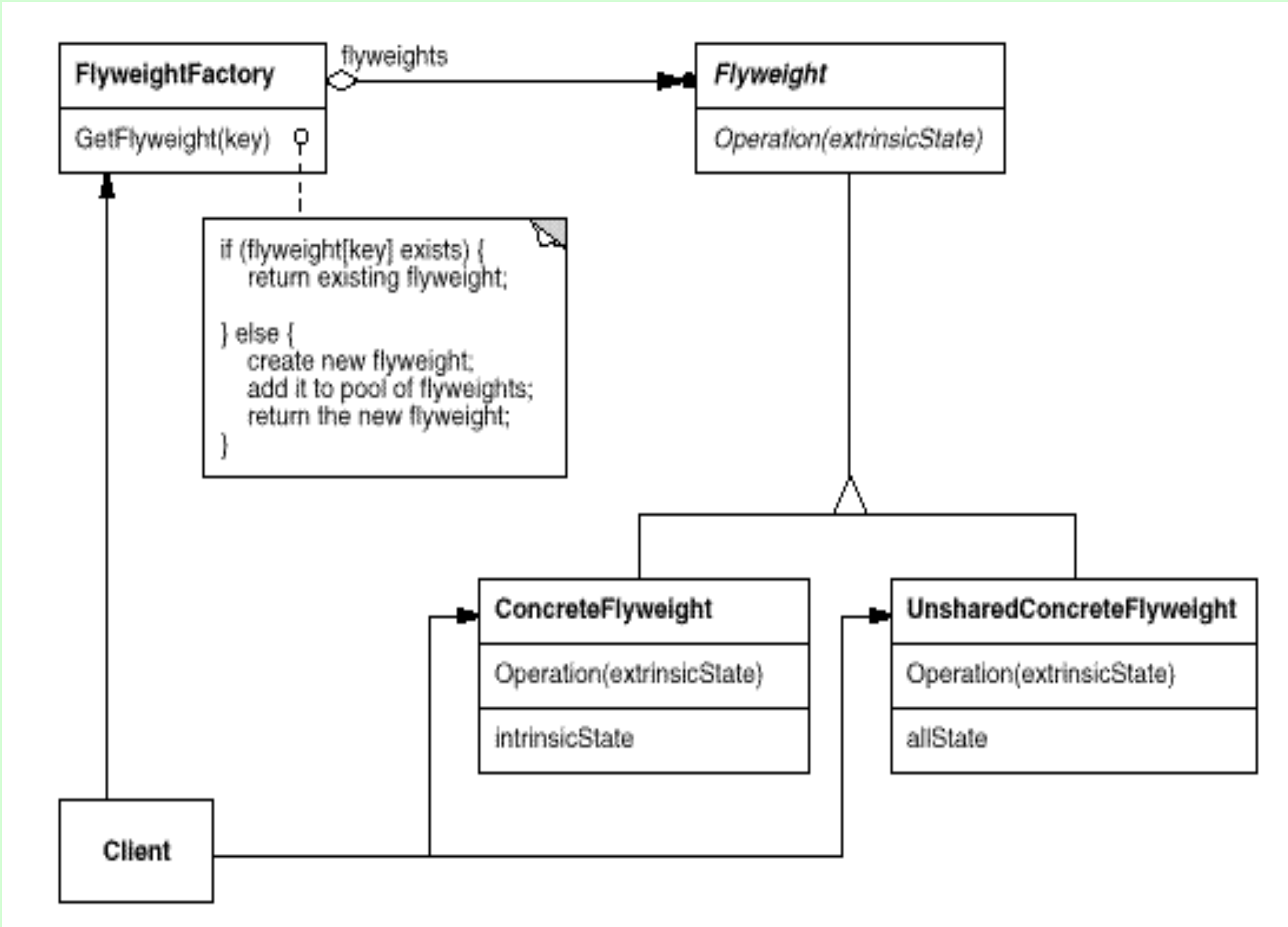
- The key concept here is the distinction between **intrinsic** and **extrinsic** state.

Extrinsic versus Intrinsic state

- **Intrinsic state** is stored in the flyweight; it consists of **information that's independent** of the flyweight's context making it **sharable.**

- **Extrinsic state** depends on and varies with the flyweight's context and therefore **can't be shared**. Client objects are responsible for passing extrinsic state to the flyweight when it needs it.

The Flyweight Structure



The Flyweight Participants

- **Flyweight**
  - declares an interface through which flyweights can receive and act on **extrinsic** state.

- **ConcreteFlyweight (Folder)**
  - implements the Flyweight interface and adds storage for intrinsic state, if any. **A ConcreteFlyweight object must be sharable**. Any state it stores must be **intrinsic**; that is, it must be independent of the ConcreteFlyweight object's context.

- **UnsharedConcreteFlyweight**
  - not all Flyweight subclasses **need to be shared**. The Flyweight interface *enables* sharing; it doesn't enforce it.

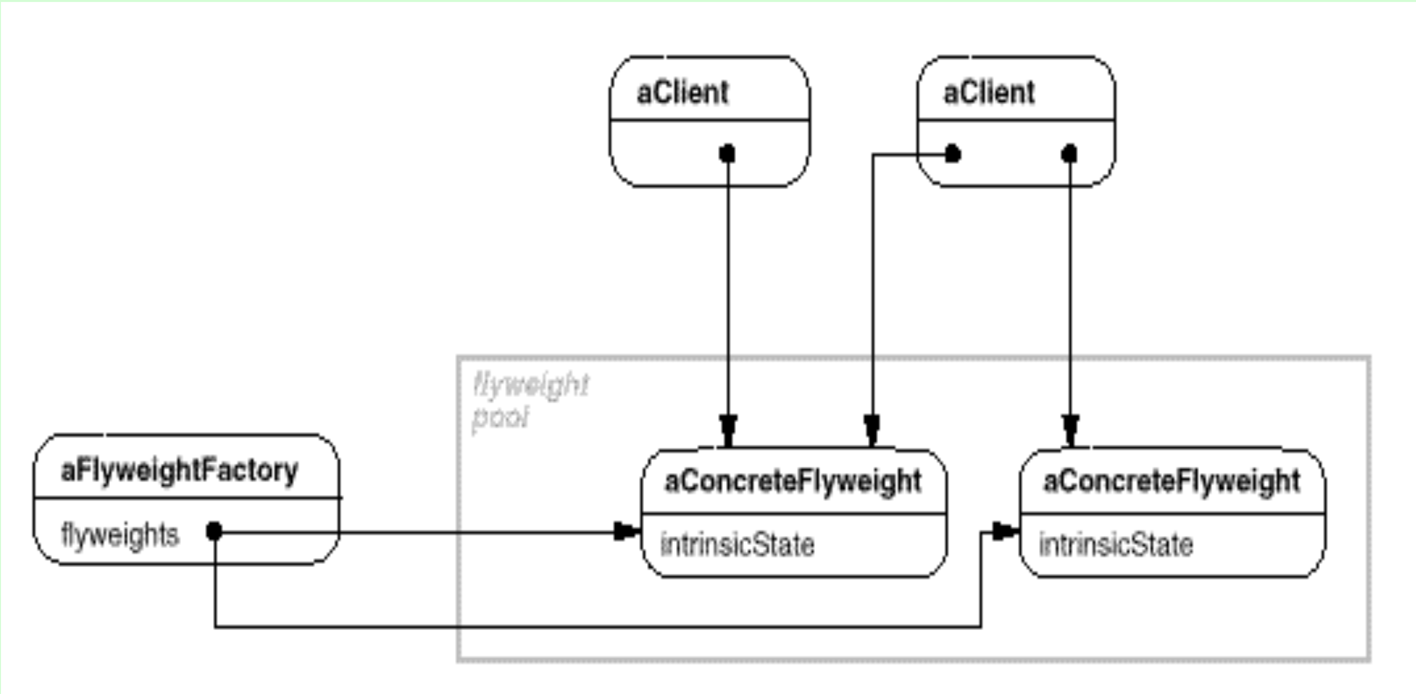
The Flyweight Participants

- **FlyweightFactory (FolderFactory)**
  - creates and manages flyweight objects.
  - ensures that flyweights are shared properly. When a client requests a flyweight, the FlyweightFactory object supplies an existing instance or creates one, if none exists.

- **Client**
  - maintains a reference to flyweight(s).
  - computes or stores the **extrinsic state** of flyweight (s).

The Flyweight Structure

- The following object diagram shows how flyweights are shared:



The Flyweight Pattern Example

- Sometimes it is necessary to create a **very large number of small class instances to represent data.** You can greatly **reduce** the number of different classes that you need to instantiate if you can **determine that the instances are fundamentally the same**, except for a few parameters.

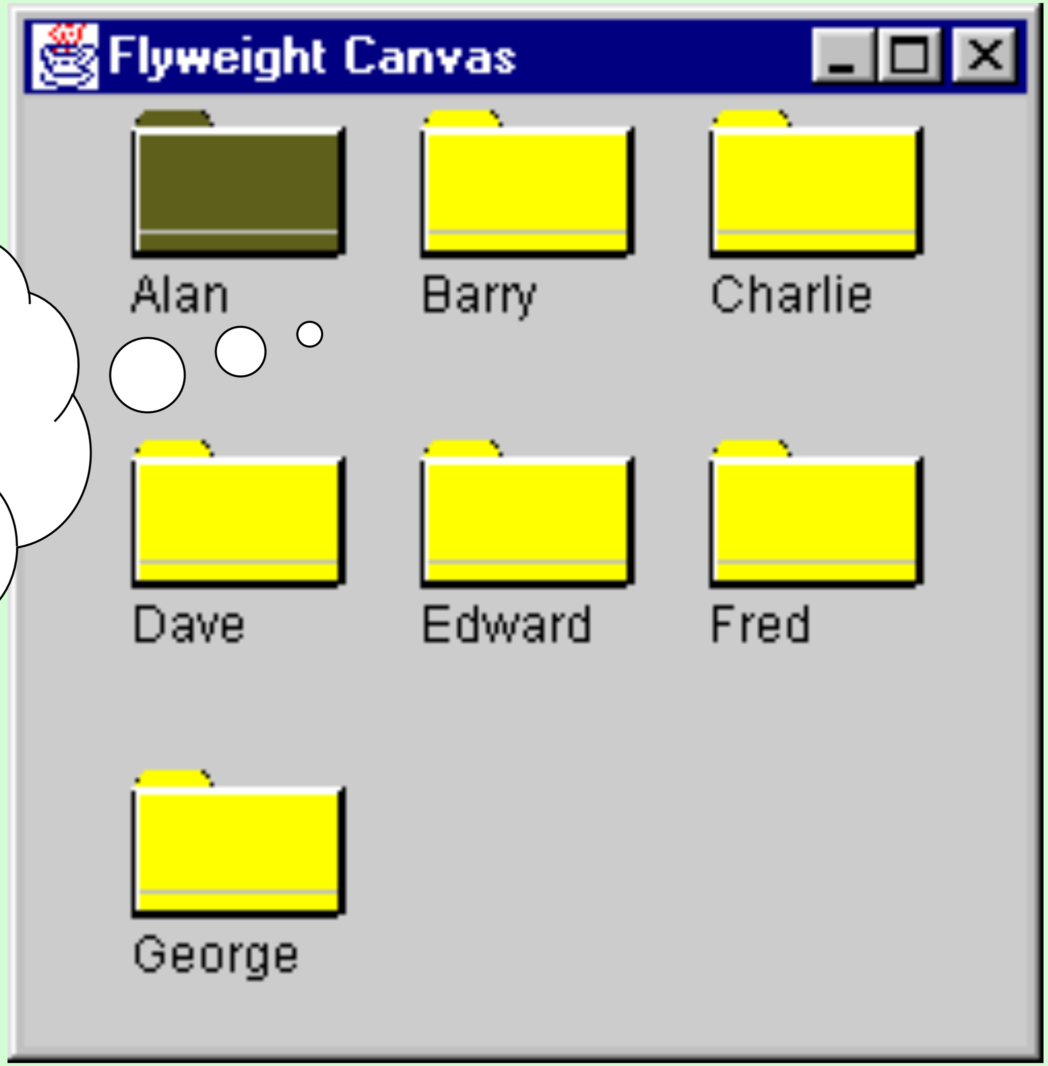
- The Flyweight pattern provides an approach for handling such classes. It refers to the instance's ***intrinsic* data** that makes the **instance unique** and the ***extrinsic* data** that is **passed as arguments.**

- The Flyweight is appropriate for **small , finegrained classes** such as those used for individual characters or icons in the screen.

- For example you might be drawing a series of icons which represent folders as shown on the next slide.

The Flyweight Pattern

Differ only in there label and location, graphics are common



The Flyweight Pattern

- A Flyweight is a **sharable instance** of a class. At first glance, each class might appear to be a **Singleton**. In fact, a small number of instances might exist, such as one for every character or one for every icon type.

- The **number of instances** that are allocated must be **decided as the class instances are needed**; usually a **FlyWeightFactory** class does this.

- The **factory class** is usually **a singleton** since it needs to **keep track of whether a particular instance** has been **generated.** It then **returns** either **a new instance** or a **reference to one that already exists**.

The Flyweight Pattern

- To decide whether some part of a program is **a candidate** for using the **flyweight pattern** consider whether it is **possible to remove some data from the class** and make it *extrinsic* (pass it in as a parameter).

- If this is possible then it **will greatly reduce the number of class instances** your program needs to maintain.

- Suppose you want to draw a small folder icon with a name under it for each person in an organization. If the organization is large , there could be many such icons; however they are actually all the same graphically. Even if we have **two icons**, one for **'Selected'** and one for **'Not Selected'.**

- In the following example having a **separate object** (icon) for each person is a **waste of resources**.

The Flyweight Pattern

- Instead, we will create a FolderFactory that **returns** either the **selected** or the **unselected folder** drawing class but does **not create additional instances** once one of each has been created. Since this is a **simple case** we can create the two instances at the outset and return one or the other.

```
import java.awt.*;

public class FolderFactory {
    Folder unSelected, Selected;
    public FolderFactory() {
        Color brown = new Color(0x5f5f1c);
        Selected =  new Folder(brown);
        unSelected = new Folder(Color.yellow);
    }

    public Folder getFolder(boolean isSelected) {
        if (isSelected)
            return Selected;
        else
            return unSelected;
    }
}
```

Two instances of folder

The Flyweight Pattern

- When **more instances** could exist, **the factory could keep a table** of the ones that it had **already created**.

- The unique thing about the flyweight pattern is that we **pass the coordinates** and the **name to be drawn** under the folder. These coordinates are the ***extrinsic* data** that allow us share the folder objects.

- The complete folder class shown on the next slide creates a **folder instance** with **one background color** or **the other** and has a public *draw* method that draws the folder at the point we specify.

```java
public class Folder extends JPanel {
    private Color color;
    final int W = 50, H = 30;
    final int tableft = 0, tabheight=4, tabwidth=20, tabslant=3;
    public Folder(Color c) {
        color = c;
    }
    public void draw(Graphics g, int tx, int ty, String name) {
        g.setColor(Color.black);                  //outline
        g.drawRect(tx, ty, W, H);
        g.drawString(name, tx, ty + H+15);  //title
        g.setColor(Color.white);
        g.drawLine (tx, ty, tx+W, ty);
        Polygon poly = new Polygon();
        poly.addPoint (tx+tableft,ty);
        poly.addPoint (tx+tableft+tabslant, ty-tabheight);
        poly.addPoint (tx+tabwidth-tabslant, ty-tabheight);
        poly.addPoint (tx+tabwidth, ty);
        g.setColor(Color.black);
        g.drawPolygon (poly);
        g.setColor(color);                        //fill rectangle
        g.fillRect(tx+1, ty+1, W-1, H-1);
        g.fillPolygon (poly);
        g.setColor(Color.white);
        g.drawLine (tx, ty, tx+W, ty);
        g.setColor(Color.lightGray);              //bend line
        g.drawLine(tx+1, ty+H-5, tx+W-1, ty+H-5);
        g.setColor(Color.black);                  //shadow lines
        g.drawLine(tx, ty+H+1, tx+W-1, ty+H+1);
        g.drawLine(tx+W+1, ty, tx+W+1, ty+H);
        g.setColor(Color.white);                  //highlight lines
        g.drawLine(tx+1, ty+1, tx+W-1, ty+1);
        g.drawLine(tx+1, ty+1, tx+1, ty+H-1);
    }
}
```

The Flyweight Pattern

- To use the flyweight class like this our calling **program (FlyCanvas)** must **calculate the position** of each folder as part of its paint routine and then **pass the coordinates to the folder instance.**

- This is an advantage as we need the folders to have a different layout depending on the size of the window. Each folders position is computed dynamically during the paint routine.

- The following slide shows the paint method of the calling program.

The Flyweight Pattern

```
public void paint(Graphics g) {
    Folder f;
    String name;

    int j = 0;          //count number in row
    int row = Top;      //start in upper left
    int x = Left;

    //go through all the names and folders
    for (int i = 0; i< names.size(); i++) {
        name = (String)names.elementAt(i);
        if (name.equals(selectedName))
            f = fact.getFolder(true);
        else
            f = fact.getFolder(false);
        //have that folder draw itself at this spot
        f.draw(g, x, row, name);

        x = x + HSpace;         //change to next posn
        j++;
        if (j >= HCount) { //reset for next row
            j = 0;
            row += VSpace;
            x = Left;
        }
    }
}
```

Selecting a Folder

- Since we have **two folder instances**, **selected** and **unselected**, we would like to be able to select folders by moving the mouse over them. In the previous paint routine, we simply remember the name of the folder that was selected and ask the factory to return a "selected" folder for it.

- Because the folders are not **individual** instances **we cannot listen for mouse motion within each folder instance.**

- Instead we listen for **mouse events** at the **window level**. If the mouse is found to be within a Rectangle we make that **corresponding folder the selected one**.

Selecting a Folder

```
public void mouseMoved(MouseEvent e) {
    int j = 0;        //count number in row
    int row = Top;    //start in upper left
    int x = Left;

    //go through all the names and folders
    for (int i = 0; i< names.size(); i++) {
        //see if this folder contains the mouse
        Rectangle r = new Rectangle(x,row,W,H);
        if (r.contains(e.getX(), e.getY())) {
            selectedName=(String)names.elementAt(i);
            repaint();
        }
        x = x + HSpace;         //change to next posn
        j++;
        if (j >= HCount) {      //reset for next row
            j = 0;
            row += VSpace;
            x = Left;
        }
    }
}
```

Flyweights in Java

- Flyweights are **not often used at application level in Java**. They are more of a **system resource management technique** that is used at lower levels then Java.

- Some objects within the Java language could be implemented as flyweights. For example if two instances of the class **String** are created with the same literal values they could refer to the same storage location.

- To **prove the absence of flyweight classes** we could use the following code.

```
String fred1 = new String("Fred");
String fred2 = new String("Fred");

System.out.println(fred1 == fred2);
```

Flyweights in Java

- The **output** of such a test **will be false** because the two reference variables fred1 and fred2 are **referencing different objects** (different memory locations).

- Remember the == operator **compares actual object references** rather then the = which checks for equality of value.

- **Layout managers** in Java are flyweights since the only **difference** between one gridlayout, for example, and another is the **list of components** it contains and some attribute values. When the layout functionality is required the components and attributes are passed to the single shared instance (i.e. you feed specific context to the shared instance..the client is responsible for context specific information)

Flyweights in Java

- Example:
  – http://www.codeproject.com/Articles/186002/Flyweight-Design-Pattern.aspx

Flyweight Applicability

Apply the Flyweight pattern when *all* of the following are true:

- An application uses a **large number of objects** (identical or nearly identical)
- **Storage costs are high** because of the sheer quantity of objects.
- Most object state can be made **extrinsic** (non-identical parts)
- The application **doesn't depend on object identity**. Since flyweight objects may be shared, identity tests will return true for conceptually distinct objects.

Flyweight Consequences

- Flyweights may introduce **run-time costs** associated with transferring, finding, and/or computing extrinsic state, especially if it was formerly stored as intrinsic state

- However, **such costs** are **offset by space savings**, which increases as more flyweights are shared.

- Storage savings are a function of several factors:
  - the **reduction** in the **total number of instances** that comes from sharing
  - the **amount of intrinsic state** per object
  - whether **extrinsic state** is **computed or stored**.

Proxy Pattern

Proxy Pattern

- **Intent:**
  Provide a **surrogate** or placeholder for another object **to control access to it.**

- **AKA:** Surrogate

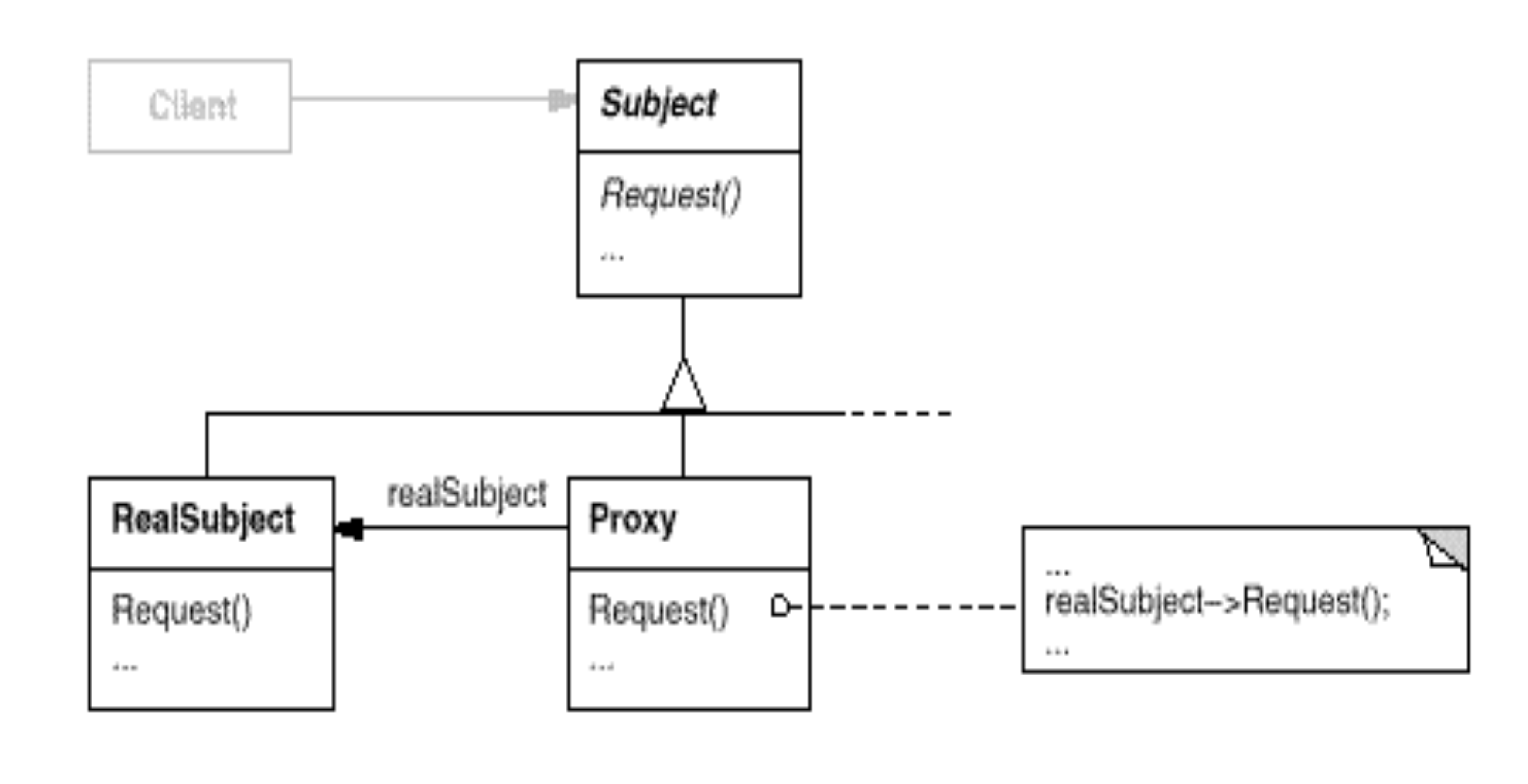
- The **proxy pattern** is used to **represent** with **a simpler object** an object that is **complex or time-consuming to create**.
  If creating an object is expensive in time or computer-resources, a proxy **allows you to postpone** this creation until you actually **need the object**.
- http://sourcemaking.com/design_patterns/proxy

Proxy Pattern

- A **proxy** usually has **the same methods** as the **full object** that it represents. Once that full object is loaded, the Proxy passes on the method calls to the full object.

- There are several cases where a proxy can be useful
  - If an object such as a **large image** takes a long time to load. When the program starts, some indication that an image is to be displayed is needed so that the screen is set out correctly. However the actual image display can be postponed until the actual image is loaded.
  - If the object is on a **remote machine** and **loading it over a network might be slow,** especially during peak network load periods.
  - If the **object** has **limited access rights**. The proxy can then validate the access permissions for that user.
  - **Example:** http://www.codeproject.com/Articles/186001/Proxy-Design-Pattern

Proxy Structure



Proxy Pattern Participants

- **Proxy**
  - maintains a **reference** that lets the proxy access the real subject.
  - provides **an interface** identical to Subject's so that a **proxy** can be **substituted** for the **real subject**.
  - **controls access** to the **real subject** and may be responsible for **creating** and **deleting it.**

Proxy Pattern Participants

- **Subject**
  - defines the **common interface for RealSubject** and Proxy so that a Proxy can be used anywhere a RealSubject is expected.
- **RealSubject**
  - **defines the real object that the proxy represents**.

Proxy Pattern Applicability

- Proxy is **applicable** whenever there is a **need** for a more versatile or **sophisticated reference** to an object than a **simple pointer**. Here are several common situations in which the Proxy pattern is applicable:

  - A **remote proxy** provides a local representative for an object in a different address space.

  - A **virtual proxy** creates expensive objects on demand.

  - A **protection proxy** controls access to the original object. Protection proxies are useful when objects should have different access rights.