

Object Orientation with Design Patterns



Lecture 8:
Behavioural Patterns
The Command 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**

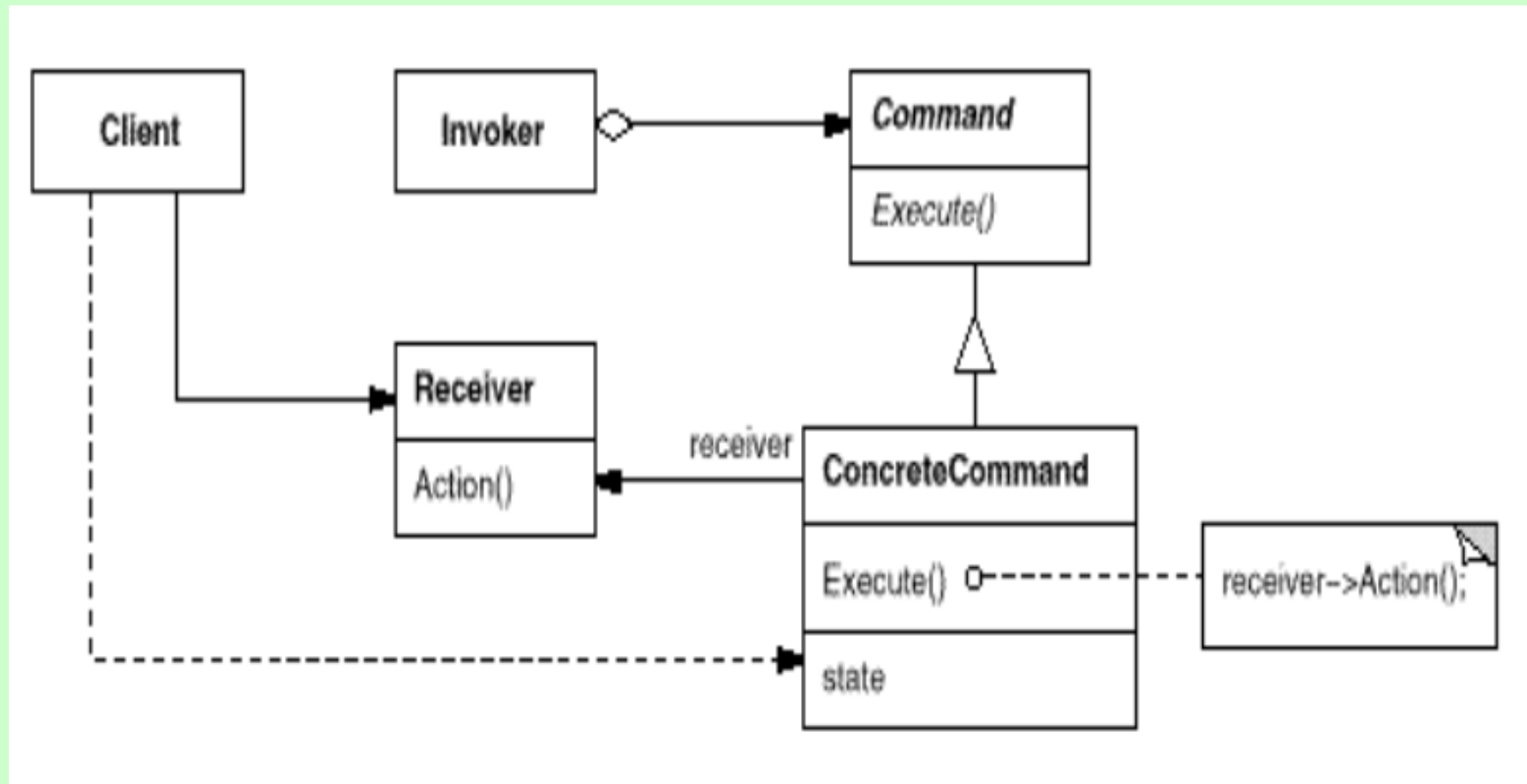
The Command Pattern

- **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.

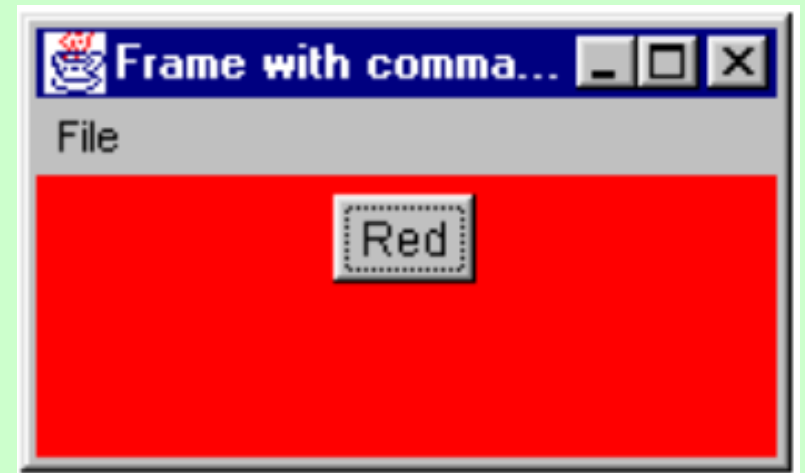
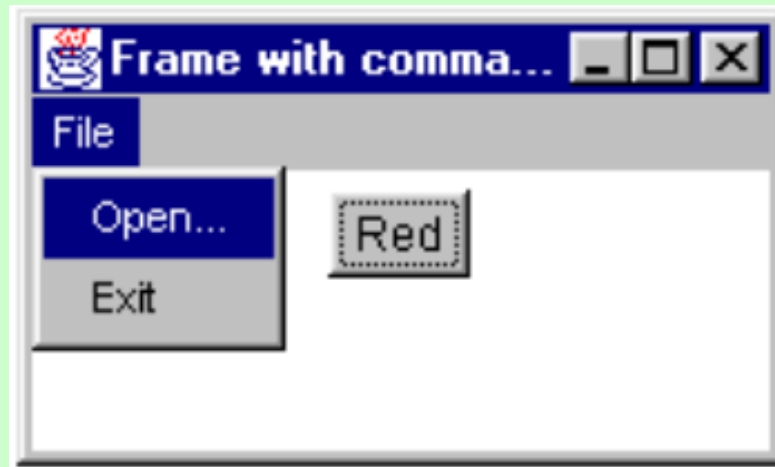
Command Pattern Applicability

- **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 commands which are composed of sequences of commands. 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 MenuItem**s 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 *actionPerformed* 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

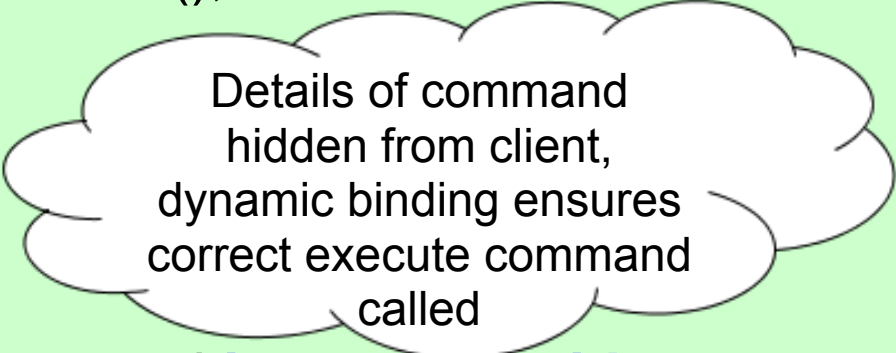
- The **abstract Command class (or Java interface structure)** **declares the interface for executing operations**. The Command pattern thus fixes the signature of an operation and lets classes vary.
- To apply the Command pattern each **concrete command implements the Command interface**
- A **Command 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();  
}
```



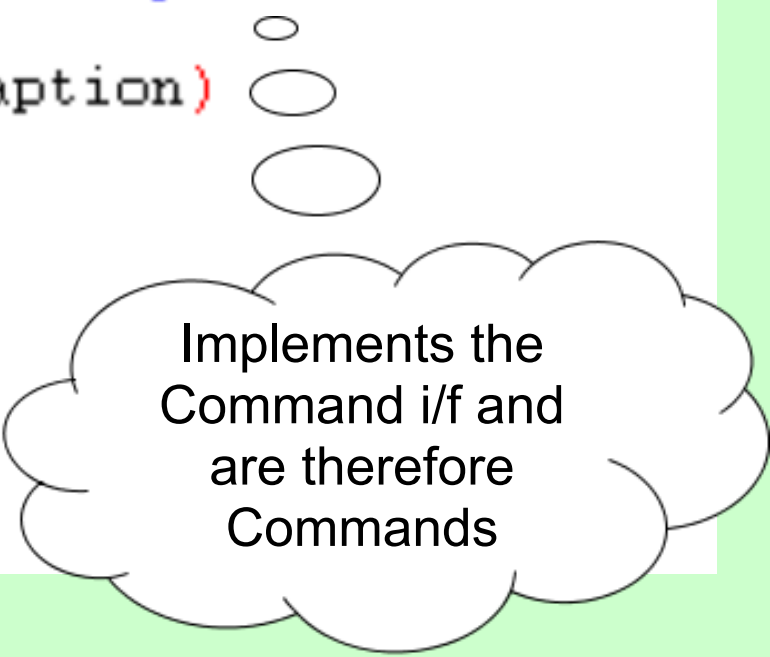
Details of command
hidden from client,
dynamic binding ensures
correct execute command
called

- 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:

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



Implements 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. JMenuItem 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. “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 cmd);  
    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 cmd) {  
        menuCommand = cmd;  
    }  
  
    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 separated 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();  
    }  
}
```

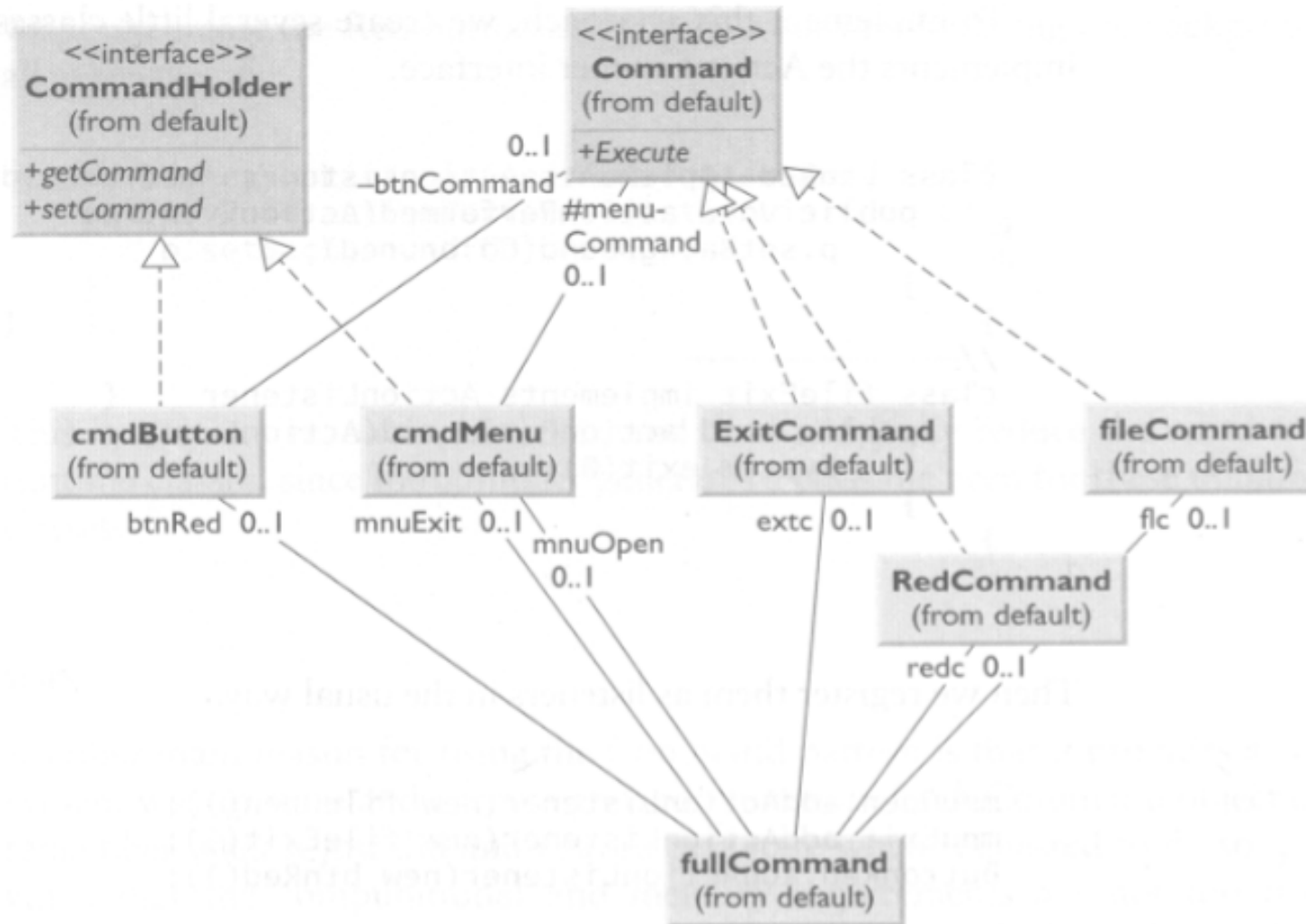
Command Objects

- 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 interface).

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
```

```
    }
```

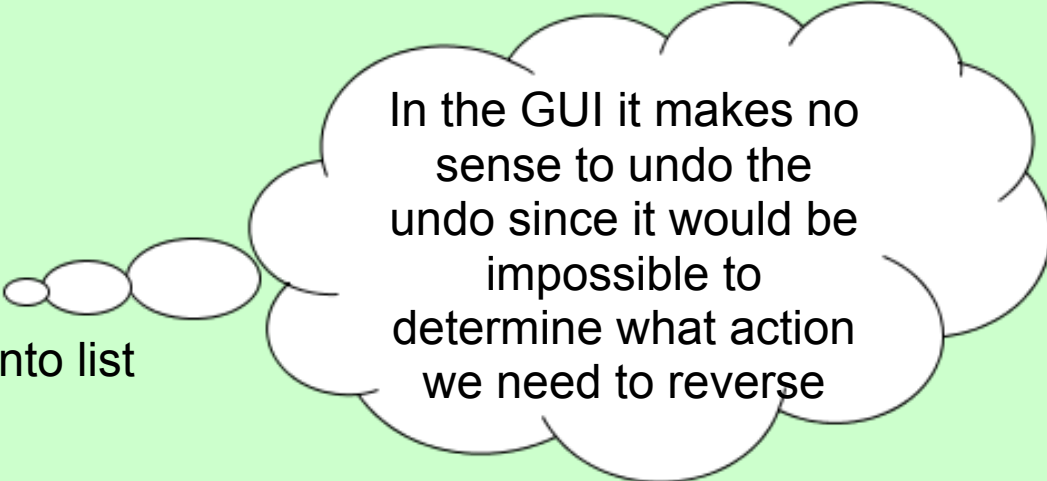
```
    //-----
```

```
    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
```

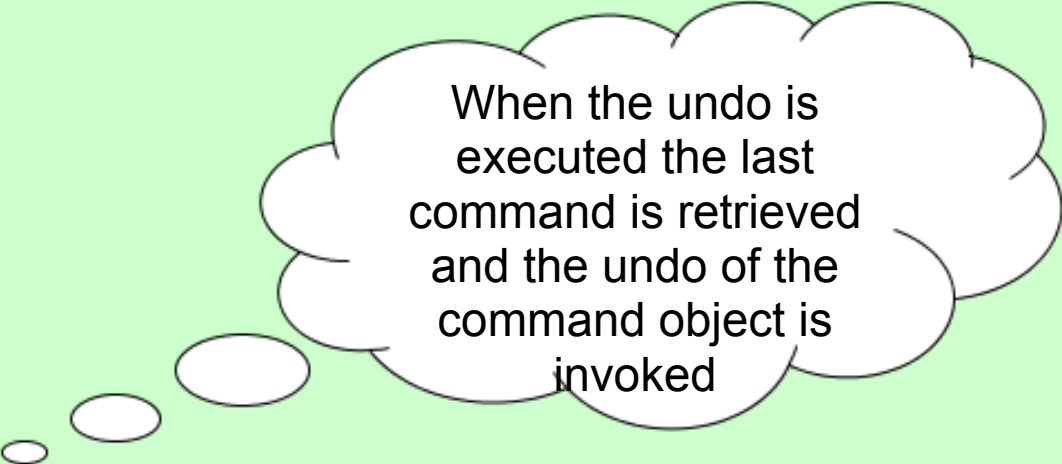
```
    }
```

```
}
```

```
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

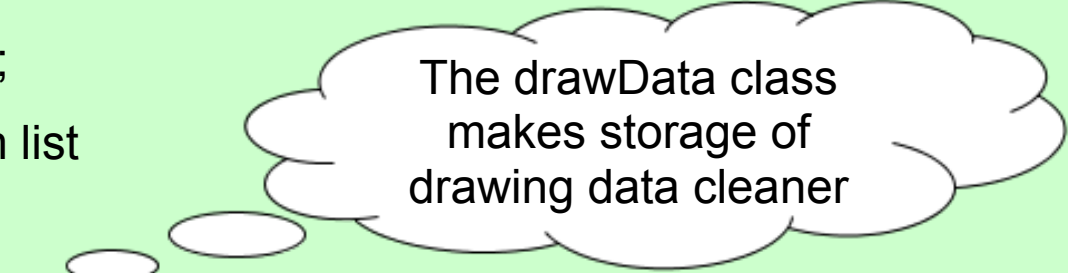
- The execute method of the drawCommand stores data associated with each draw command in a vector, this is the data that will be used to undo a particular draw command
- The x and y location of the drawing is incremented so that the lines move across the screen (negative dx for blue)

```
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

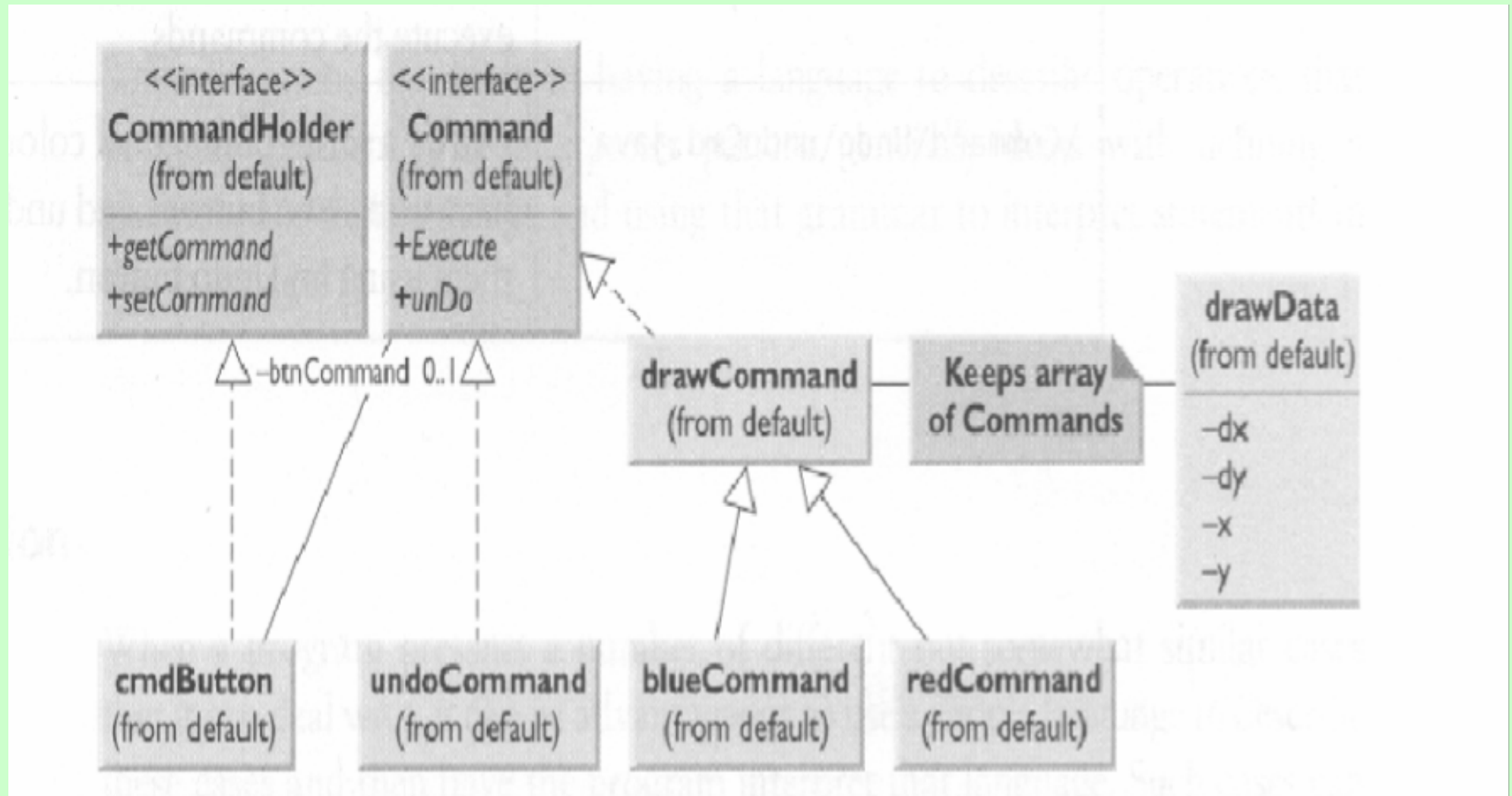
- The *unDo* method of the drawCommand removes the last stored line from it's list and resets the x and y values
- This *unDo* method is called when the undo button is pushed and the undoCommand *execute* method is called

```
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  
}
```



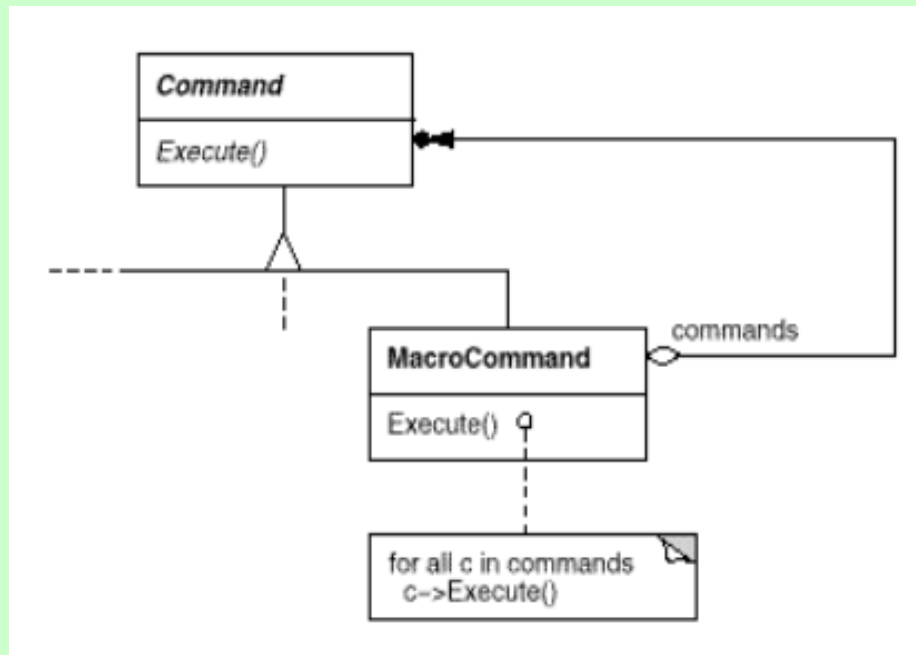
The drawData class
makes storage of
drawing data cleaner

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**.

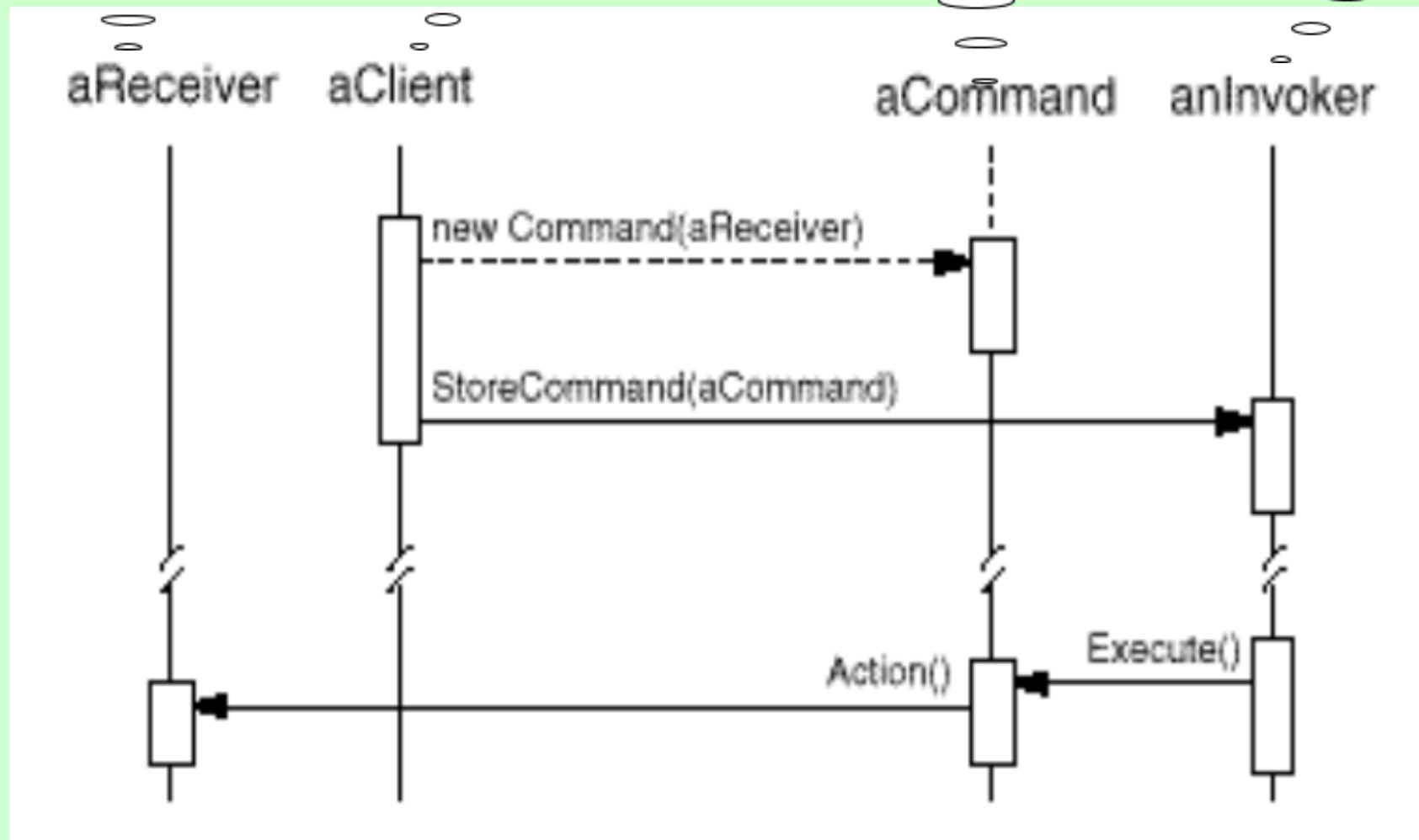
Command Collaborations

e.g. paintPanel

e.g. GUI

e.g. fileOpen

MenuItem



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

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**.

Exercis es

1. Modify the 'tCommand' program (in ActionCommand Folder) so that it includes the following functionality:

- A Yellow button , when clicked the background changes to yellow.
- An option on the file menu called 'display'. When selected a dialog box is displayed with text 'Display menu selected'

2. Modify the 'fullCommand' program (in fullCommand Folder) so that it includes the following functionality:

- A Green button , when clicked the background changes to green.
- An option on the file menu called 'update'. When selected a dialog box is displayed with text 'Update menu selected'