Design Patterns V

Ergude Bao

Beijing Jiaotong University

Content

- Behavioral patterns I

Behavioral Patterns I

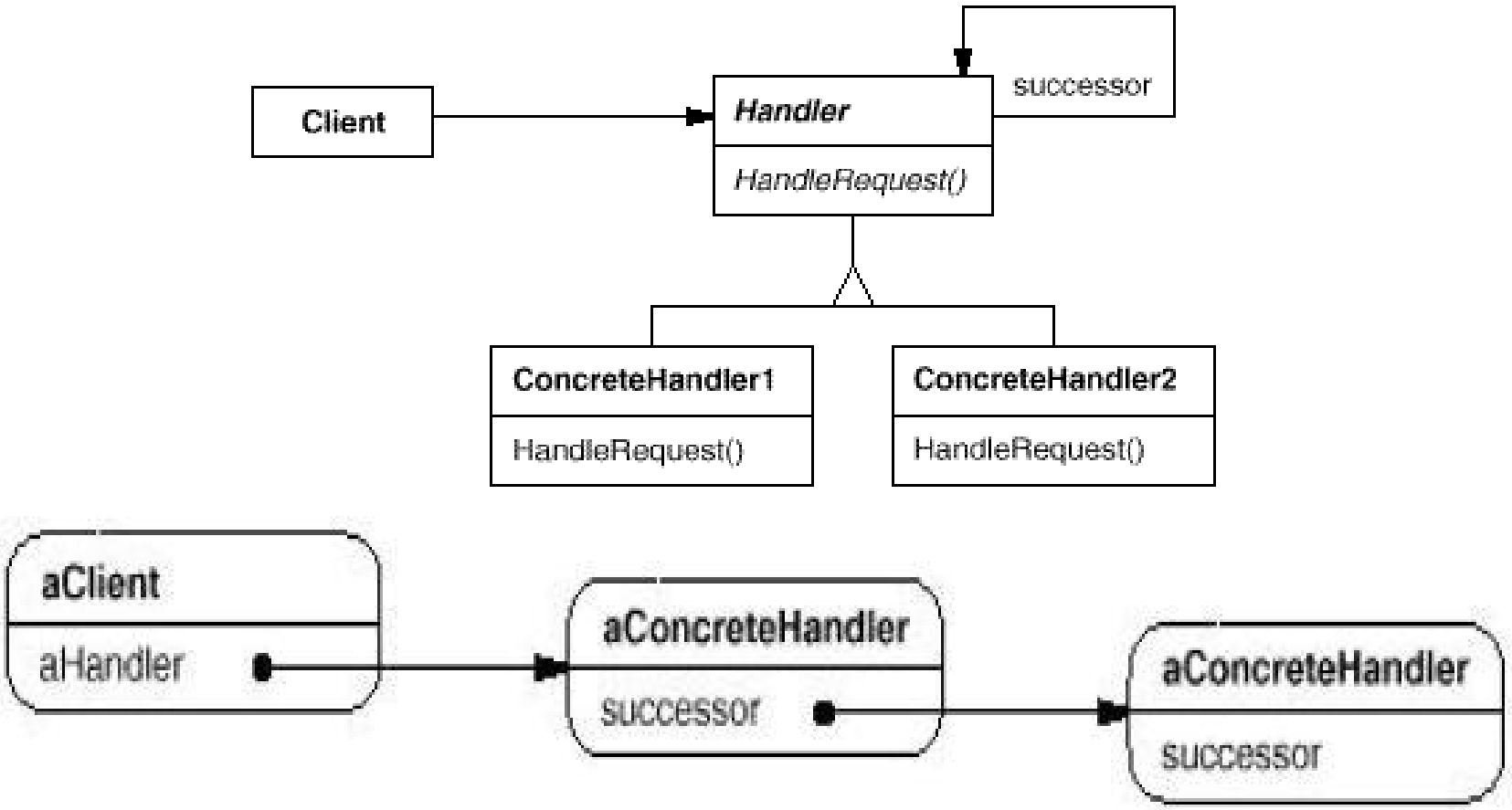
- Chain of Responsibility Pattern
- Command Pattern
- Interpreter Pattern
- Iterator Pattern
- Mediator Pattern
- Memento Pattern

Chain of Responsibility Pattern

Chain of Responsibility Pattern

- Motivation
  - We want to avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request
- Solution
  - Chain the receiving objects and pass the request along the chain until an object handles it

Chain of Responsibility Pattern



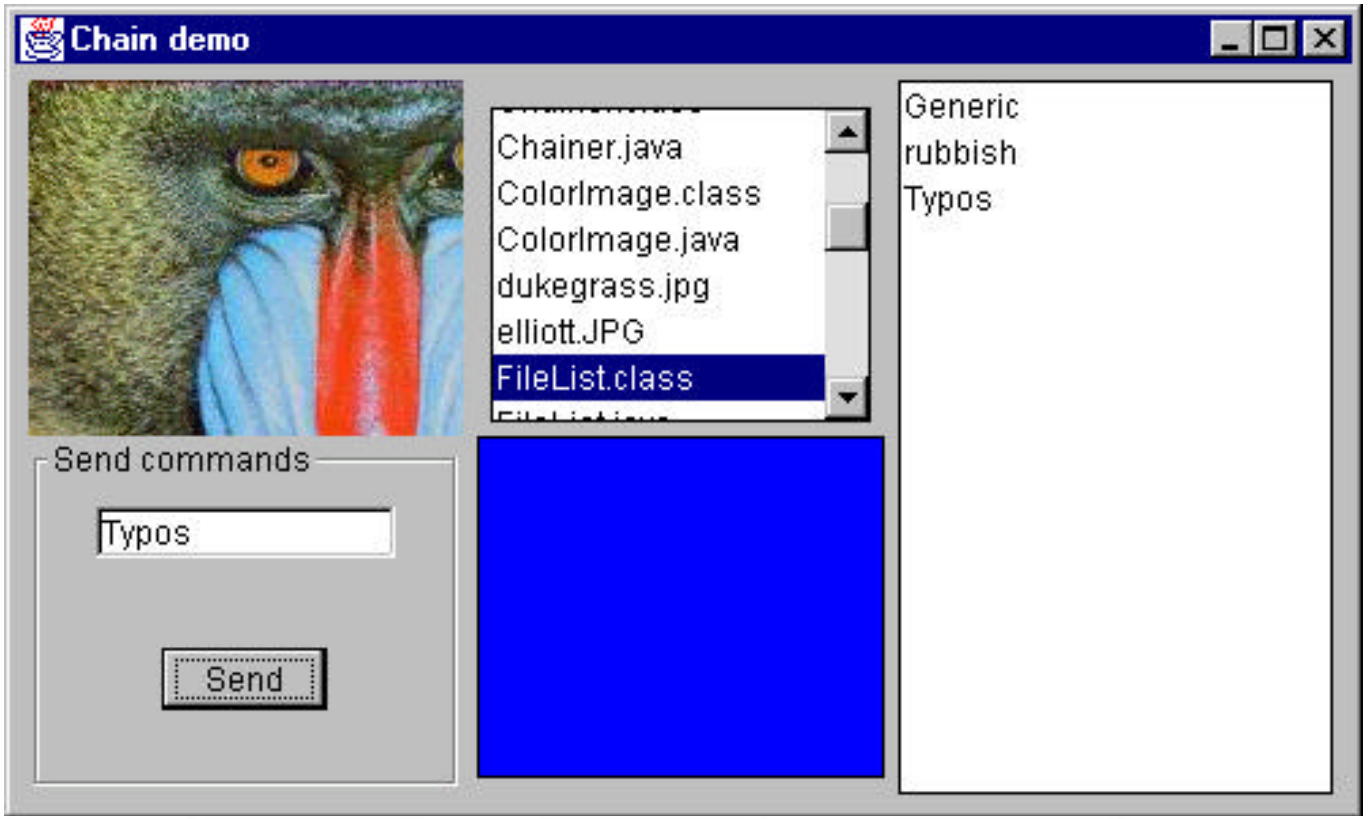
Participants

- Handler
  - Defines an interface for handling the requests
  - Implements the successor link
- ConcreteHandler
  - Handles requests it is responsible for
  - Can access its successor
  - If the ConcreteHandler can handle the request, it does so; otherwise it forwards the request to its successor
- Client
  - Initiates the request to a ConcreteHandler object on the chain

Example I

- Consider a context-sensitive help system for a GUI
  - The object that ultimately provides the help is not known explicitly to the object (e.g., a button) that initiates the help request
- Use a chain of objects to decouple the senders from the receivers. The request gets passed along the chain until one of the objects handles it
  - Each object on the chain shares a common interface for handling requests and for accessing its successor on the chain

Example II


Chain demo
Chainer.java
ColorImage.class
ColorImage.java
dukegrass.jpg
elliott.JPG
FileList.class
Generic
rubbish
Typos
Send commands
Typos
Send


Command
Image file
Color name
File name
General

Discussion

- Notice in the basic structure for the CoR pattern that the Handler class just has one method, handleRequest(). Here it is as a Java interface:

```
public interface Handler {
        public void handleRequest();
}
```

- What if we want to handle different kinds of requests, for example, help, print and format requests?

Discussion

- Solution 1: Change our Handler interface to support multiple request types as follows:

```
public interface Handler {
    public void handleHelp();
    public void handlePrint();
    public void handleFormat();
 }
```

Discussion

- Now any concrete handler would have to implement all of the methods of this Handler interface
  - An example of a concrete handler for this new Handler interface:

```
public class ConcreteHandler implements Handler {
        private Handler successor;
        public ConcreteHandler(Handler successor) {
                this.successor = successor;
        }
        public void handleHelp() {
                //we handle help ourselves, so help code is here
        }
        public void handlePrint() {
                successor.handlePrint();
        }
        public void handleFormat() {
                successor.handleFormat();
        }
}
```

- If we add a new kind of request we need to change the interface which means that all concrete handlers need to be modified!

Discussion

- Solution 2: Have separate handler interfaces for each type of request

```
public interface HelpHandler {
        public void handleHelp();
}
public interface PrintHandler {
        public void handlePrint();
}
public interface FormatHandler {
        public void handleFormat();
}
```

- Now a concrete handler can implement one (or more) of these interfaces. The concrete handler must have successor references to each type of request that it deals with, in case it needs to pass the request on to its successor

Discussion

- Here's a concrete handler which deals with all three request types:

```
public class ConcreteHandler implements HelpHandler, PrintHandler, FormatHandler {
        private HelpHandler helpSuccessor;
        private PrintHandler printSuccessor;
        private FormatHandler formatSuccessor;

        public ConcreteHandler(HelpHandler  helpSuccessor, PrintHandler printSuccessor,
FormatHandler formatSuccessor) {
                this.helpSuccessor = helpSuccessor;
                this.printSuccessor = printSuccessor;
                this.formatSuccessor = formatSuccessor;
        }

        public void handleHelp() {  //handle help,  so help code is here }
        public void handlePrint() {printSuccessor.handlePrint();}
        public void handleFormat() {formatSuccessor.handleFormat();}
  }
```

- If we add a new kind of request we need to add a new interface which still means that all concrete handlers need to be modified!

Discussion

- Solution 3: Have a single method in the Handler interface which takes an argument describing the type of request

```
public interface Handler {
    public void handleRequest(String request);
}
```

Discussion

- Concrete handler looks like:

```
public class ConcreteHandler implements Handler {
        private Handler successor;
        public ConcreteHandler(Handler successor) {
                this.successor = successor;
        }
        public void handleRequest(String request) {
                if (request.equals("Help")) {
                        // help code is here.
                } else      // Pass it on!
                        successor.handle(request);
        }
}
```

- If we add a new kind of request we do not need to add/change the interface which means existing codes are not affected!

When to Use

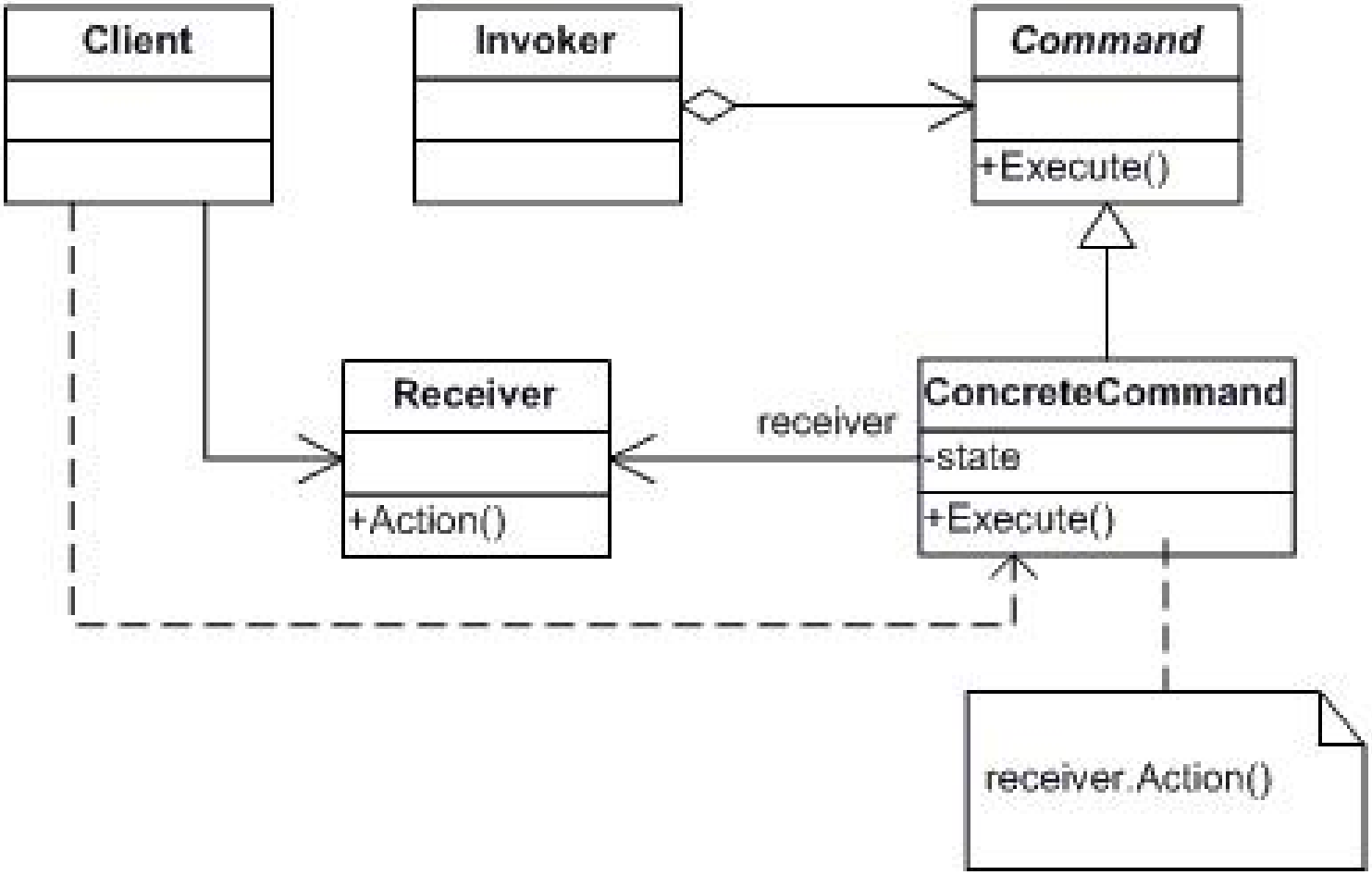
- When more than one object may handle a request and the actual handler is not known in advance
- When requests follow a “handle or forward” model - that is, some requests can be handled where they are generated while others must be forwarded to another object to be handled
- When you want to be able to modify the set of objects dynamically that can handle requests

Command Pattern

Command Pattern

- Motivation
  - We want to allow the client to make requests without knowing anything about the actual action that will be performed, and allow to change that action without affecting the client program in any way
- Solution
  - Encloses a request for a specific action as an object and gives it a known public interface

Command Pattern



Participants

- Command
  - Declares an interface for executing an operation
- ConcreteCommand
  - Defines a binding between a Receiver object and an action
  - Implements Execute by invoking the corresponding operation(s) on Receiver
- Client
  - Creates a ConcreteCommand object and sets its receiver
- Invoker
  - Asks the command to carry out the request
- Receiver
  - Knows how to perform the operations associated with carrying out the request

Example I

- A GUI system has several buttons that perform various actions. We want to have menu items that perform the same action as its corresponding button

Example I

- Solution 1: Have one action listener for all buttons and menu items. As seen earlier, this is not a good solution as the resulting actionPerformed() method violates the Open-Closed Principle

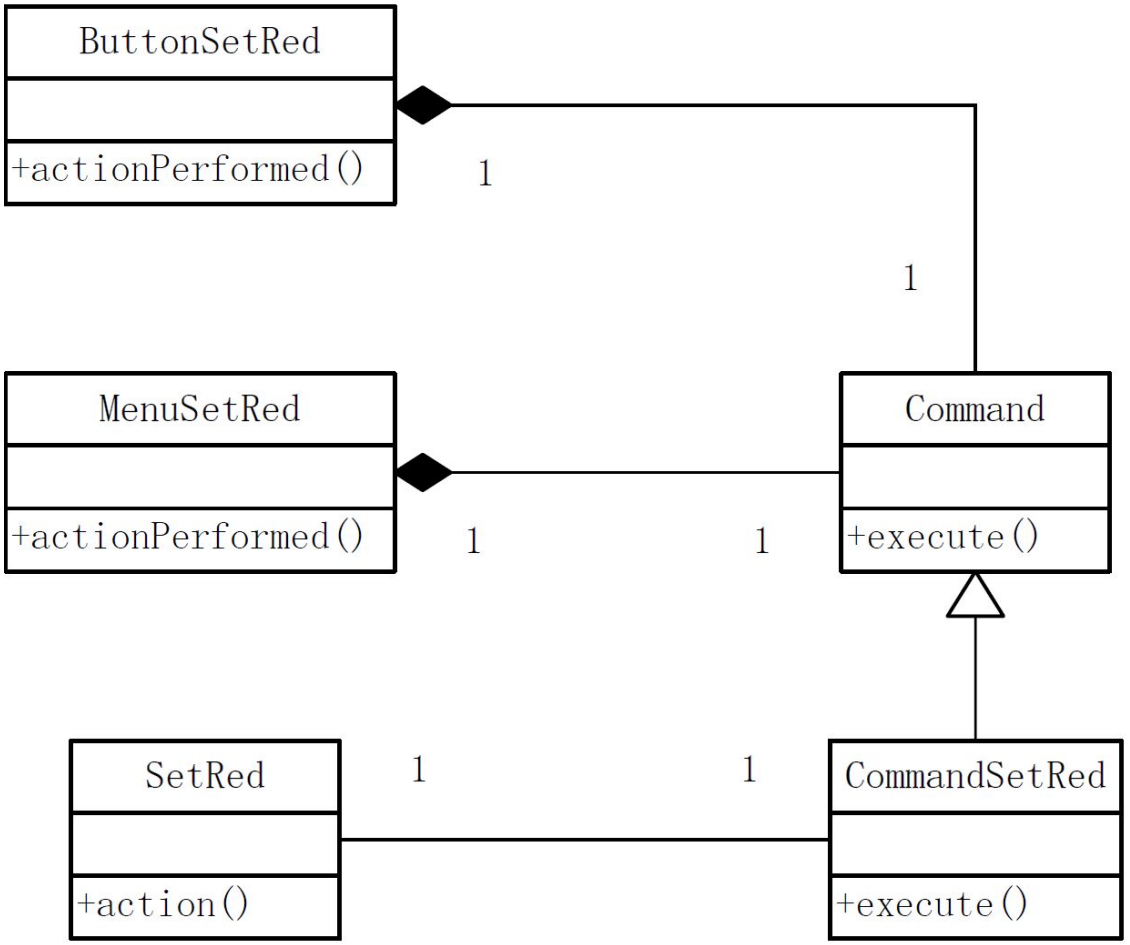
```
public void actionPerformed(ActionEvent e) {
        Object obj = e.getSource();
        if(obj == mnuOpen)
                fileOpen(); //open file
        if (obj == mnuExit)
                exitClicked(); //exit from program
        if (obj == btnRed)
                redClicked(); //turn red
}
```

Example I

- Solution 2: Have an action listener for each paired button and menu item. Keep the required actions in the actionPerformed() method of this one action listener
  - This solution is essentially the Command pattern with simple ConcreteCommand classes that perform the actions themselves

```
public interface Command {
        public void Execute();
}
public void actionPerformed(ActionEvent e) {
        ConcreteCommand cmd = (ConcreteCommand)e.getSource();
        cmd.Execute();
}
```

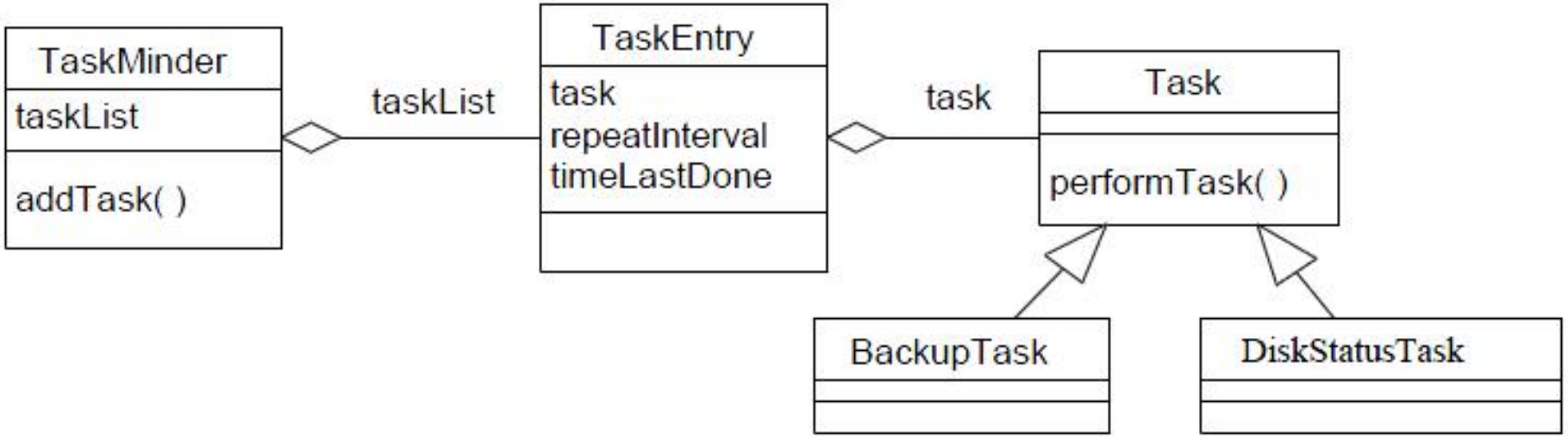
Example I


ButtonSetRed
+actionPerformed()
1
1
MenuSetRed
+actionPerformed()
1
1
Command
+execute()
SetRed
1
1
CommandSetRed
+action()
+execute()

Example II

- We want to write a class that can periodically execute one or more methods of various objects.
  - For example, we want to run a backup operation every hour and a disk status operation every ten minutes.
- We do not want the class to know the details of these operations or the objects that provide them. In other words, we want to decouple the class that schedules the execution of these methods with the classes that actually provide the behavior we want to execute

Example II


TaskMinder
taskList
addTask( )
taskList
TaskEntry
task
repeatInterval
timeLastDone
task
Task
performTask( )
BackupTask
DiskStatusTask

Advantages

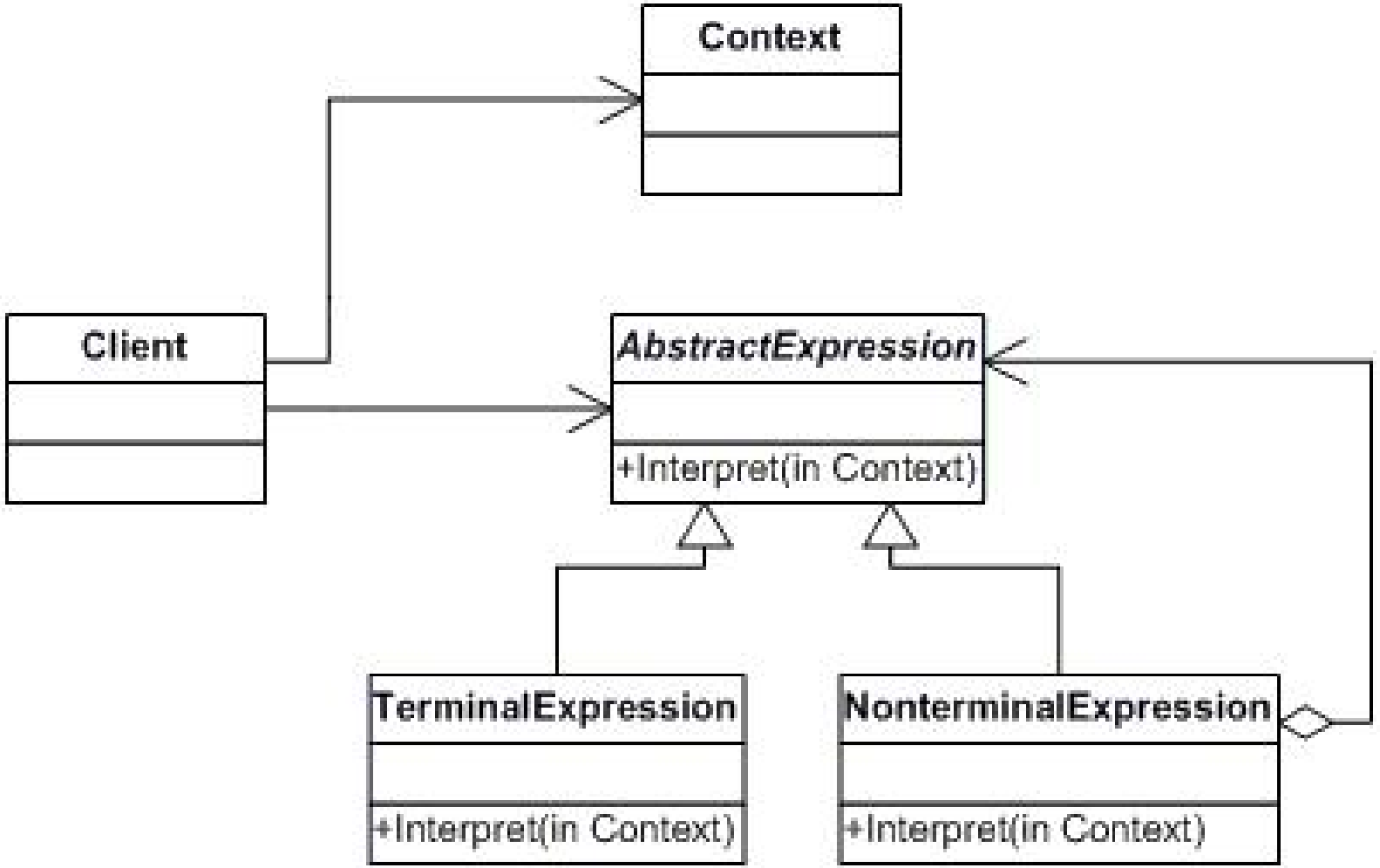
- Command decouples the object that invokes the operation from the one that knows how to perform it
- Command can be manipulated and extended like any other object
- Command can be made into a composite command

Interpreter Pattern

Interpreter Pattern

- Motivation
  - Given a language, define a representation for its grammar along with an interpreter that uses the representation to interpret sentences in the language

Interpreter Pattern



Participants

- AbstractExpression
  - Declares an interface for executing an operation
- TerminalExpression
  - Implements an Interpret operation associated with terminal symbols in the grammar
- NonterminalExpression
  - Implements an Interpret operation for nonterminal symbols in the grammar
- Context
  - Contains information that is global to the interpreter
- Client
  - Builds (or is given) an abstract syntax tree representing a particular sentence in the language that the grammar defines. The abstract syntax tree is assembled from instances of the NonterminalExpression and TerminalExpression classes
  - Invokes the Interpret operation

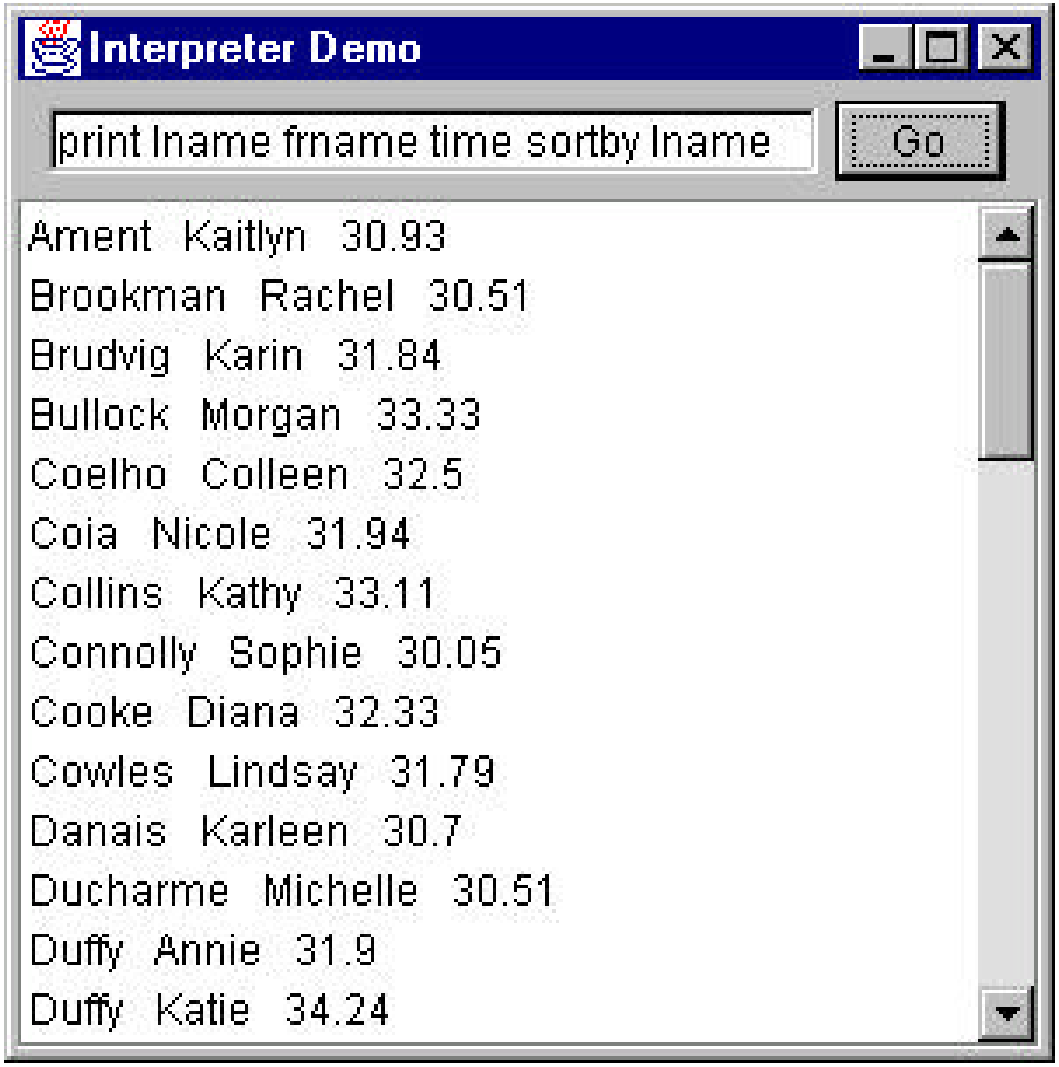
Example

- Suppose we have the following sort of results from a swimming competition, where the 5 columns are frname, lname, age, club and time :

Amanda McCarthy	12	WCA	29.28
Jamie Falco	12	HNHS	29.80
Meaghan O'Donnell	12	EDST	30.00
Greer Gibbs	12	CDEV	30.04
Rhiannon Jeffrey	11	WYW	30.04
Sophie Connolly	12	WAC	30.05
Dana Helyer	12	ARAC	30.18

- A simple report generator to operate on the data and return various reports should be designed
- A language to run the generator to obtain different reports would be very useful

Example


Interpreter Demo
print lname frname time sortby lname
Go
Ament Kaitlyn 30.93
Brookman Rachel 30.51
Brudvig Karin 31.84
Bullock Morgan 33.33
Coelho Colleen 32.5
Coia Nicole 31.94
Collins Kathy 33.11
Connolly Sophie 30.05
Cooke Diana 32.33
Cowles Lindsay 31.79
Danais Karleen 30.7
Ducharme Michelle 30.51
Duffy Annie 31.9
Duffy Katie 34.24

Example

- Define a very simple non-recursive grammar of the sort

  Print lname frname club time Sortby club Thenby time
- 3 verbs: Print, Sortby, and Thenby, plus 5 column names: frname, lname, age, club, and time
- Simple grammar of this language is punctuation free

  Print var[var] [Sortby var [Thenby var]]

Example

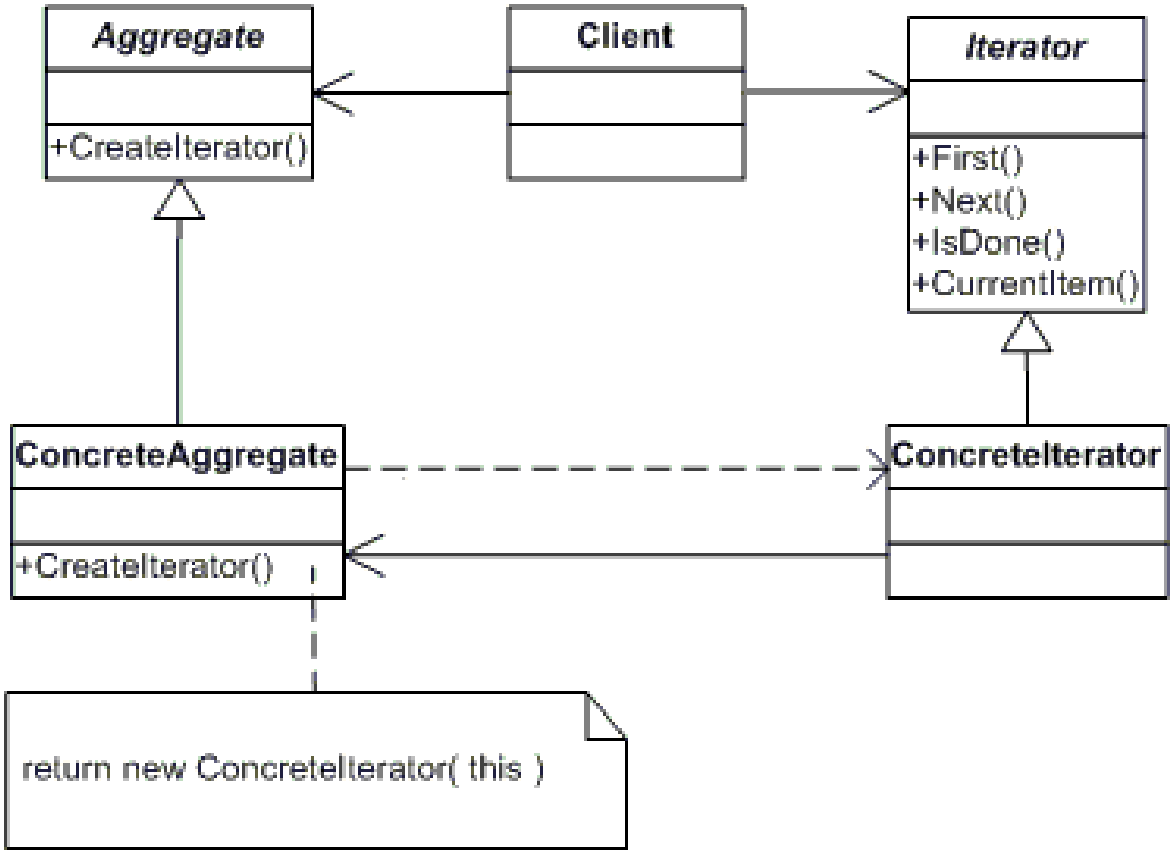
- Interpreting the language takes place in three steps
  - Parsing the language symbols into tokens
  - Reducing the tokens into actions
  - Executing the actions
- Parsers push each parsed token onto a stack. The stack can be implemented using a Vector with push, pop, top and nextTop methods to examine and manipulate the stack contents

Iterator Pattern

Iterator Pattern

- Motivation
  - We want to provide a way to access the elements of a list or collection of data sequentially without exposing its underlying representation
- Solution
  - Move through an aggregate object using a standard interface without having to know the details of the internal representations of that object

Iterator Pattern



Participants

- Iterator
  - Defines an interface for accessing and traversing elements
- ConcreteIterator
  - Implements the Iterator interface
  - Keeps track of the current position in the traversal
- Aggregate
  - Defines an interface for creating an Iterator object
- ConcreteAggregate
  - Implements the Aggregate interface to return an object of the ConcreteIterator

Iterators in Java

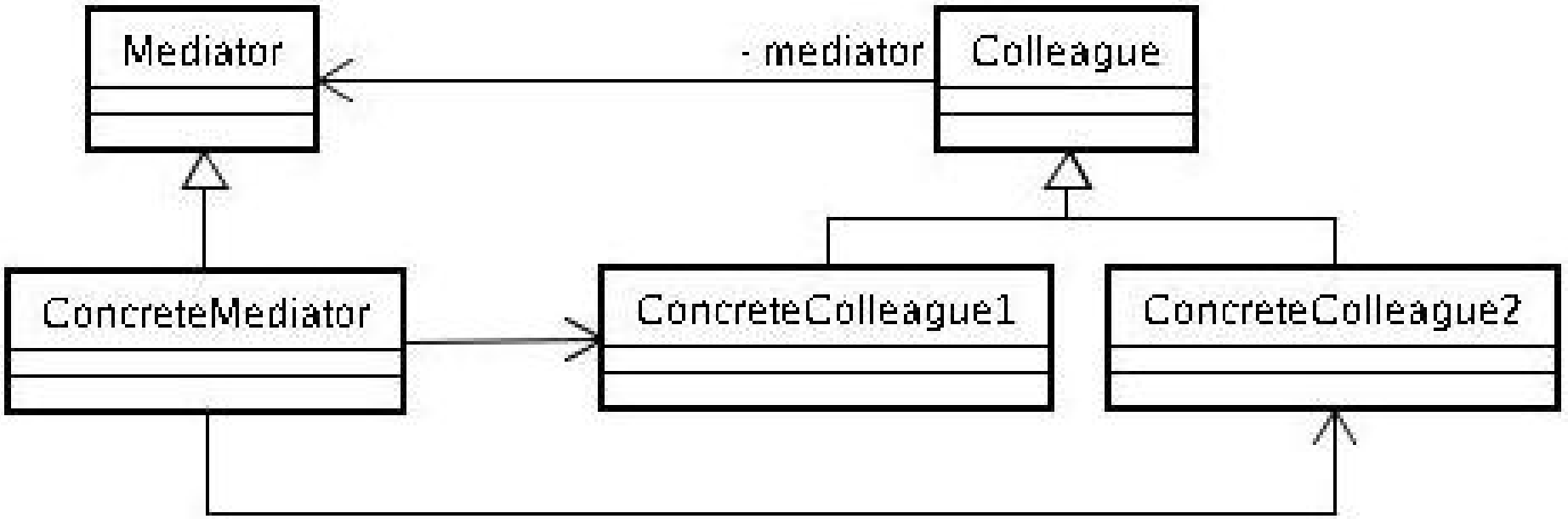
- Java provides built-in support for the Iterator pattern
  - Java provides many aggregate classes, such as sets, lists, maps and vector
  - Each aggregate class provides a method to return an Iterator object
  - The Iterator object allows traversal of objects of the aggregate class

Mediator Pattern

Mediator Pattern

- Motivation
  - With more isolated classes are developed in a program, the communication between the classes become more complex, and the program becomes harder to be read and maintained
- Solution
  - Define an object that encapsulates/manages object-to-object communications, in order to promote loose coupling, and to allow to vary the communications

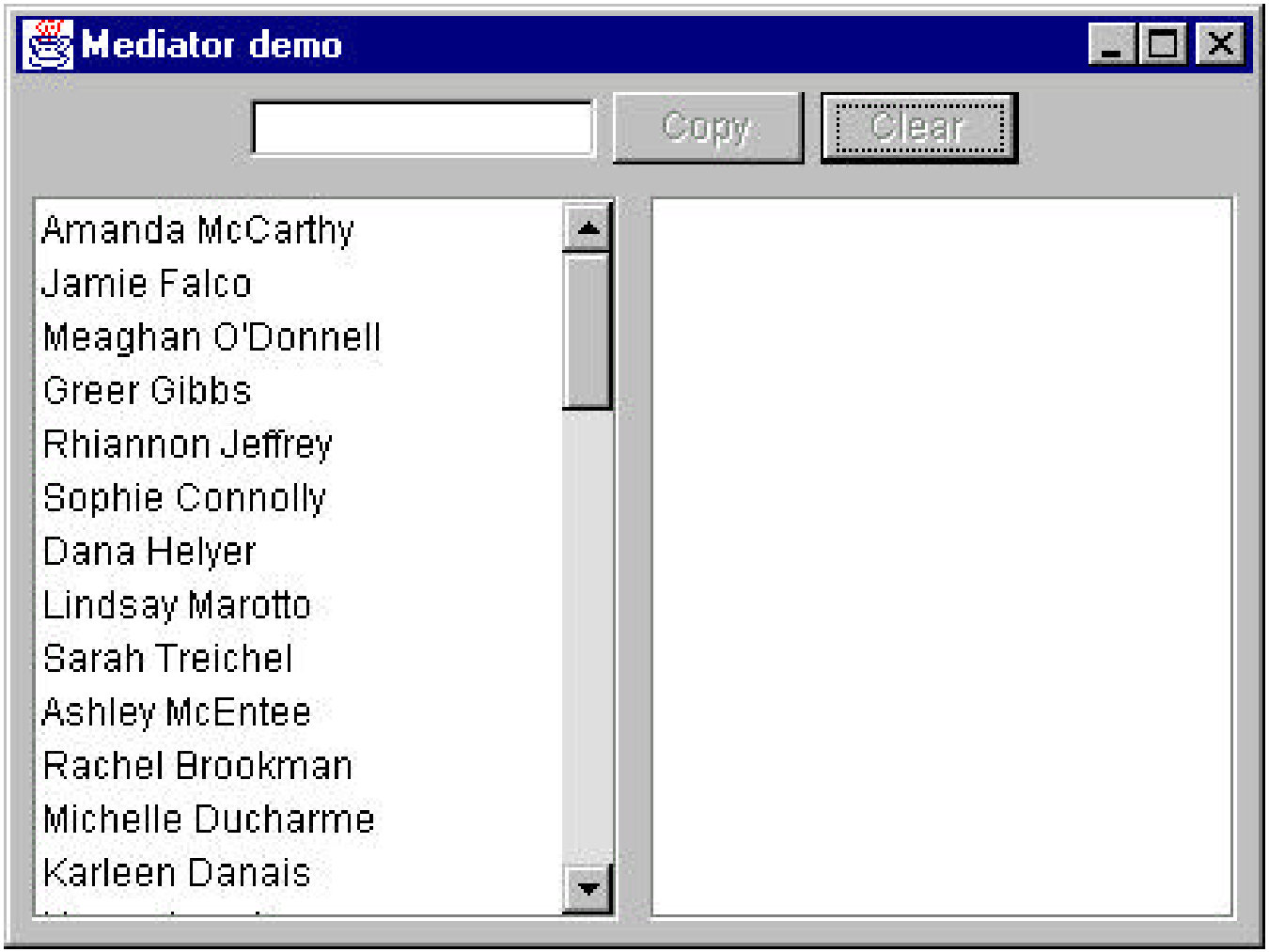
Mediator Pattern



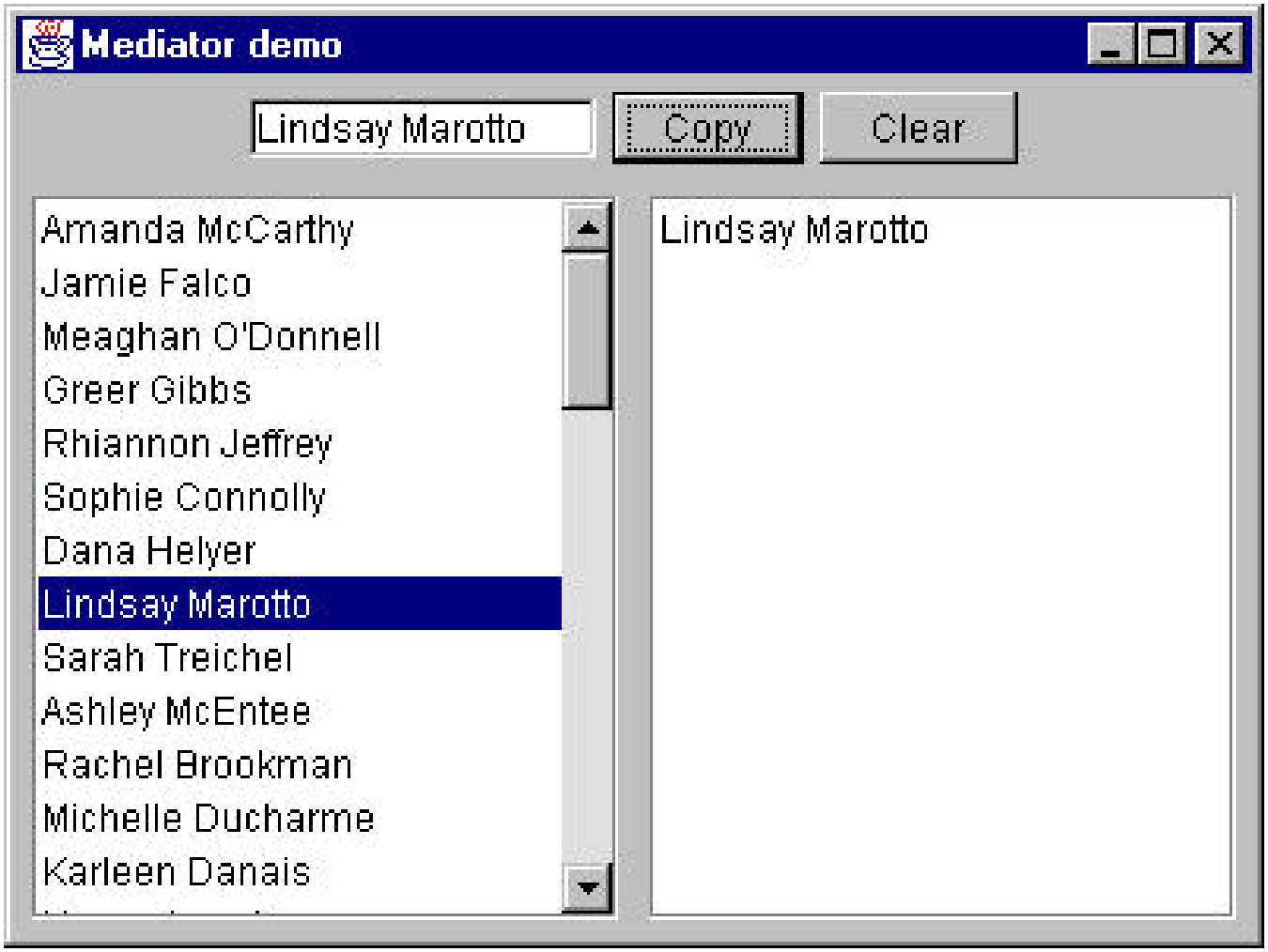
Participants

- Mediator
  - Defines an interface for communicating with Colleague classes
- Concrete Mediator
  - Knows and maintains its colleagues
  - Implements cooperative behavior by coordinating colleague objects
- Colleague
  - Defines an interface for communicating with the Mediator class
- Concrete Colleague classes
  - Knows its mediator
  - Communicates with its mediator whenever it needs to communicate with another colleague

Example

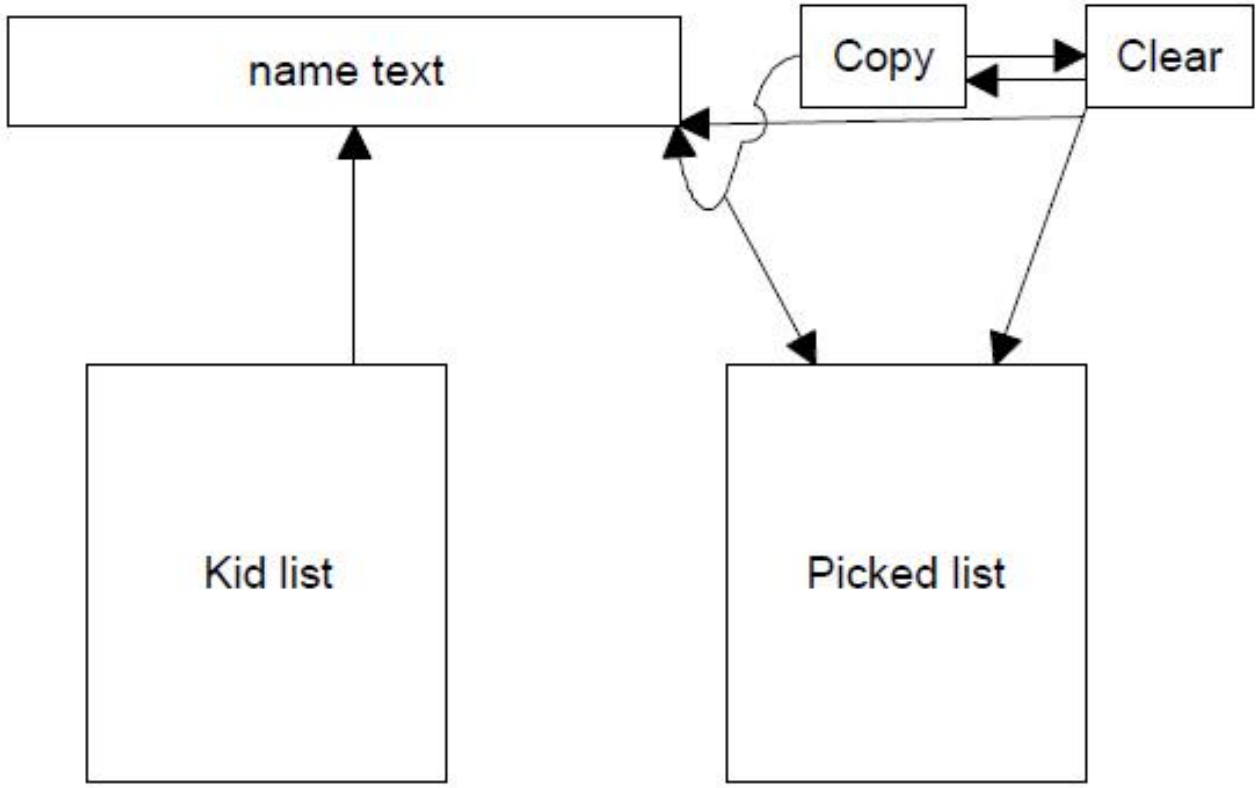

Mediator demo
Copy
Clear
Amanda McCarthy
Jamie Falco
Meaghan O'Donnell
Greer Gibbs
Rhiannon Jeffrey
Sophie Connolly
Dana Helyer
Lindsay Marotto
Sarah Treichel
Ashley McEntee
Rachel Brookman
Michelle Ducharme
Karleen Danais

Example


Mediator demo
Lindsay Marotto
Copy
Clear
Amanda McCarthy
Jamie Falco
Meaghan O'Donnell
Greer Gibbs
Rhiannon Jeffrey
Sophie Connolly
Dana Helyer
Lindsay Marotto
Sarah Treichel
Ashley McEntee
Rachel Brookman
Michelle Ducharme
Karleen Danais
Lindsay Marotto

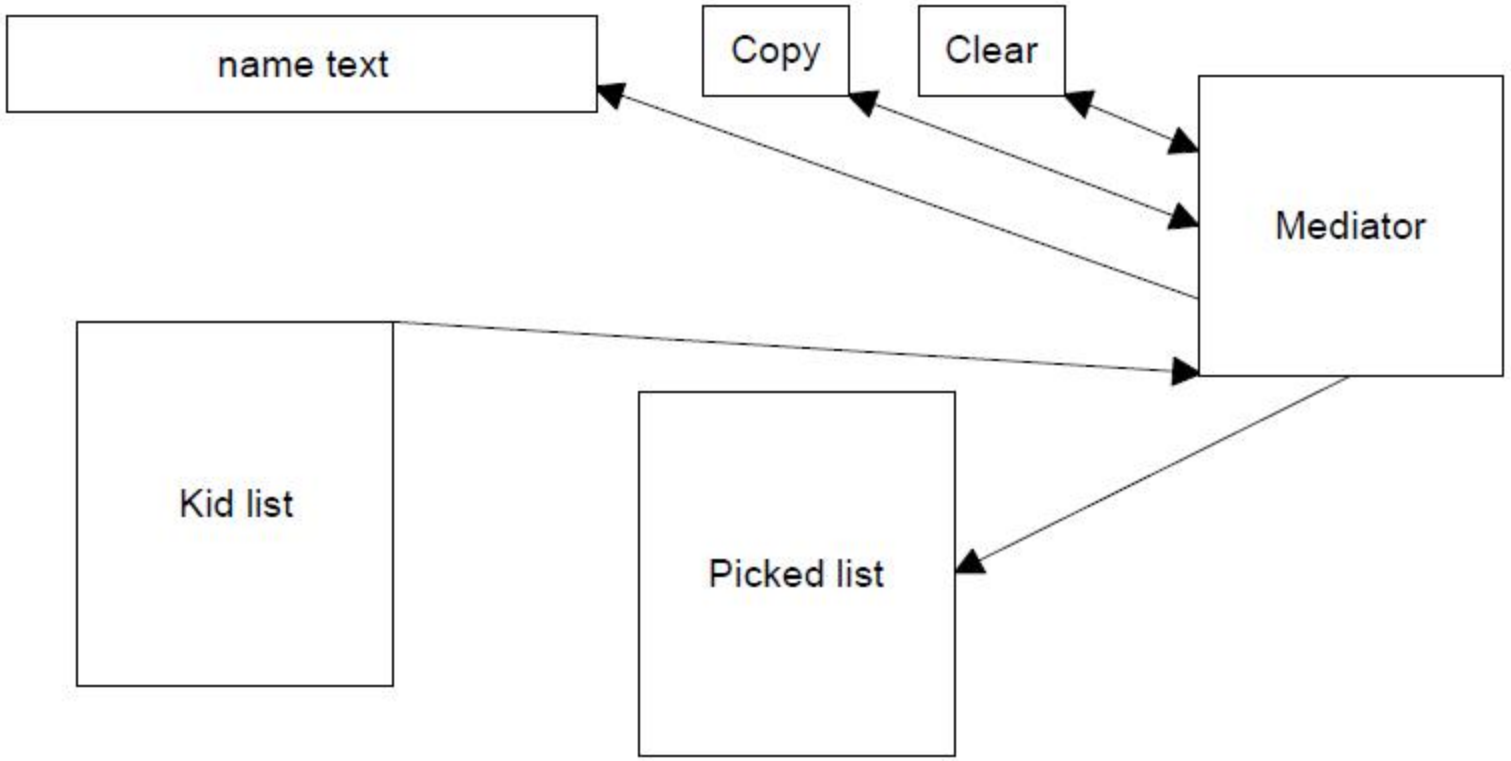
Example

- Heuristic design



Example

- Design with the mediator pattern



Implementation

- Start by creating an instance of the Mediator and then pass the instance of the Mediator to each class in its constructor

  Mediator med = new Mediator();

  kidList = new KidList(med);

  tx = new KTextField(med);

  Move = new MoveButton(this, med);

  Clear = new ClearButton(this, med);

  med.init();
- Another approach is to have a single interface to your Mediator, and pass to the Mediator various constants or objects which tell the Mediator which operations to perform

Advantages

- Comprehension: the mediator encapsulate the logic of mediation between the colleagues, so it is easier to understand this logic since it is kept in only one class
- Decoupled Colleagues: the colleague classes are totally decoupled. Adding a new colleague class is very easy due to this decoupling level
- Simplified object protocols: the colleague objects need to communicate only with the mediator objects. Practically the mediator pattern reduce the required communication channels (protocols) from many to many to one to many and many to one
- Limited Subclassing: the entire communication logic is encapsulated by the mediator class, so when this logic need to be extended only the mediator class need to be extended

Disadvantages

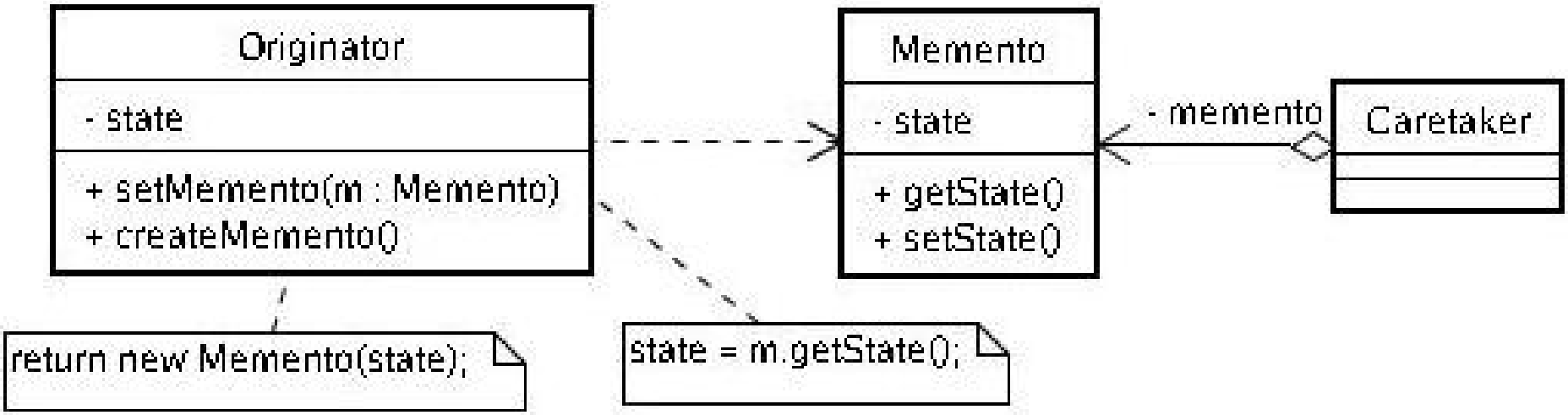
- Mediator can become monolithic in complexity, making it hard to change and maintain
- It is difficult to reuse Mediator code in different projects
  - Each Mediator has only the methods for the corresponding Colleagues only and knows only the Colleagues
- Each class needs to be aware of the existence of the Mediator

Memento Pattern

Memento Pattern

- Motivation
  - We want to restore an object back to its previous state (e.g. “undo” or “rollback” operations)
- Solution
  - Without violating encapsulation, capture and externalize an object’s internal state so that the object can be returned to this state later

Memento Pattern



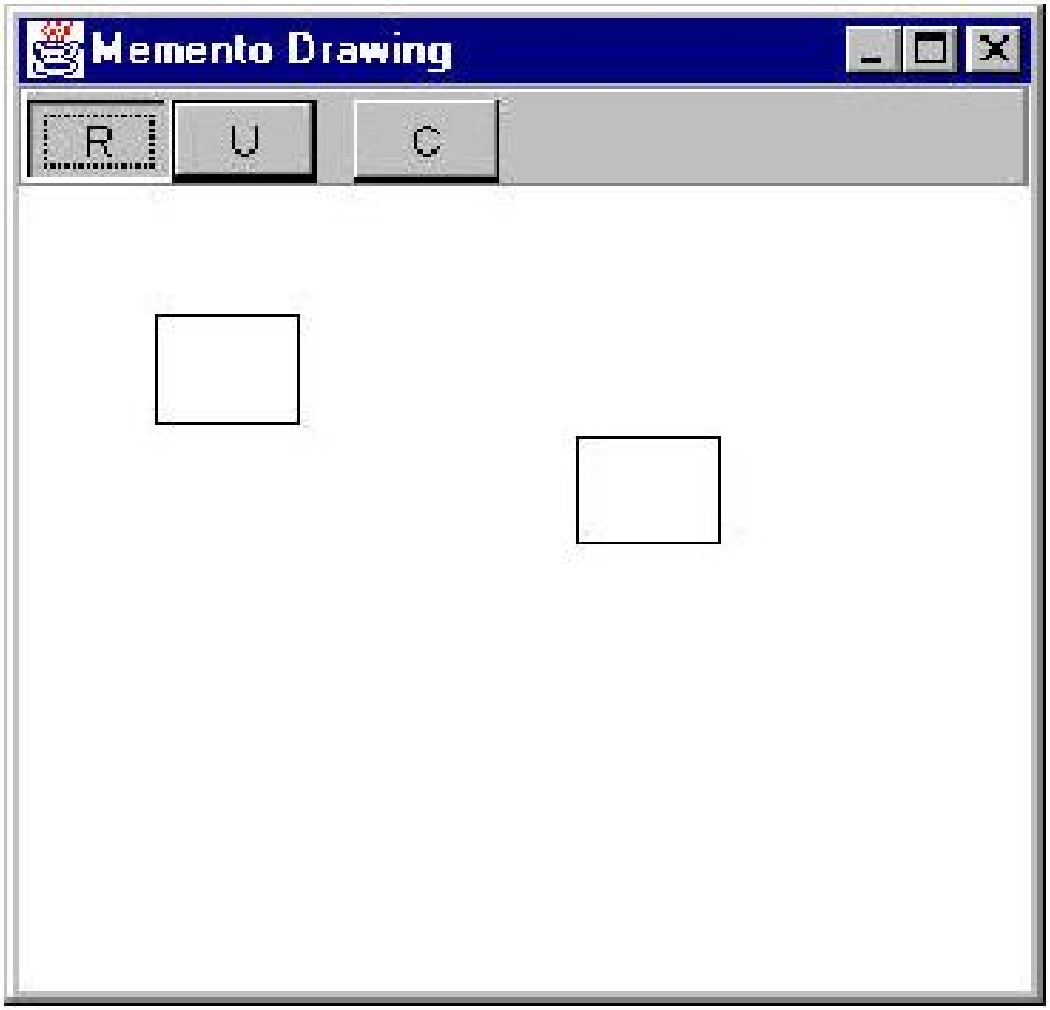
Participants

- Memento
  - Stores internal state of the Originator object
  - Protects against access by objects of other than the originator
- Originator
  - Creates a memento containing a snapshot of its current internal state
  - Uses the memento to restore its internal state
- Caretaker
  - Responsible for the memento's safekeeping
  - Never operate on or examine the contents of a memento

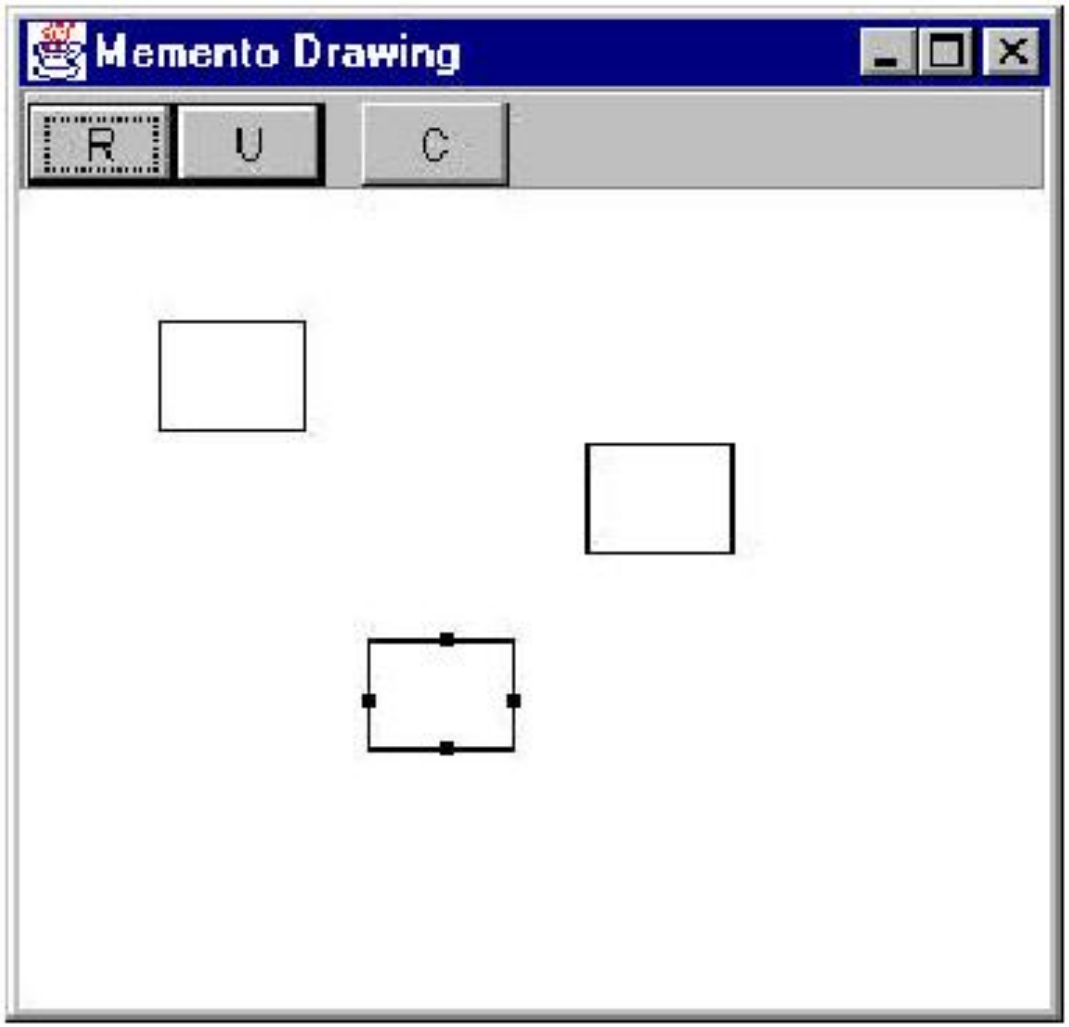
Participants

- Caretaker first asks the Originator for a Memento object
- Caretaker does whatever operation (or sequence of operations) it was going to do
- To roll back to the state before the operations, it returns the Memento object to the Originator

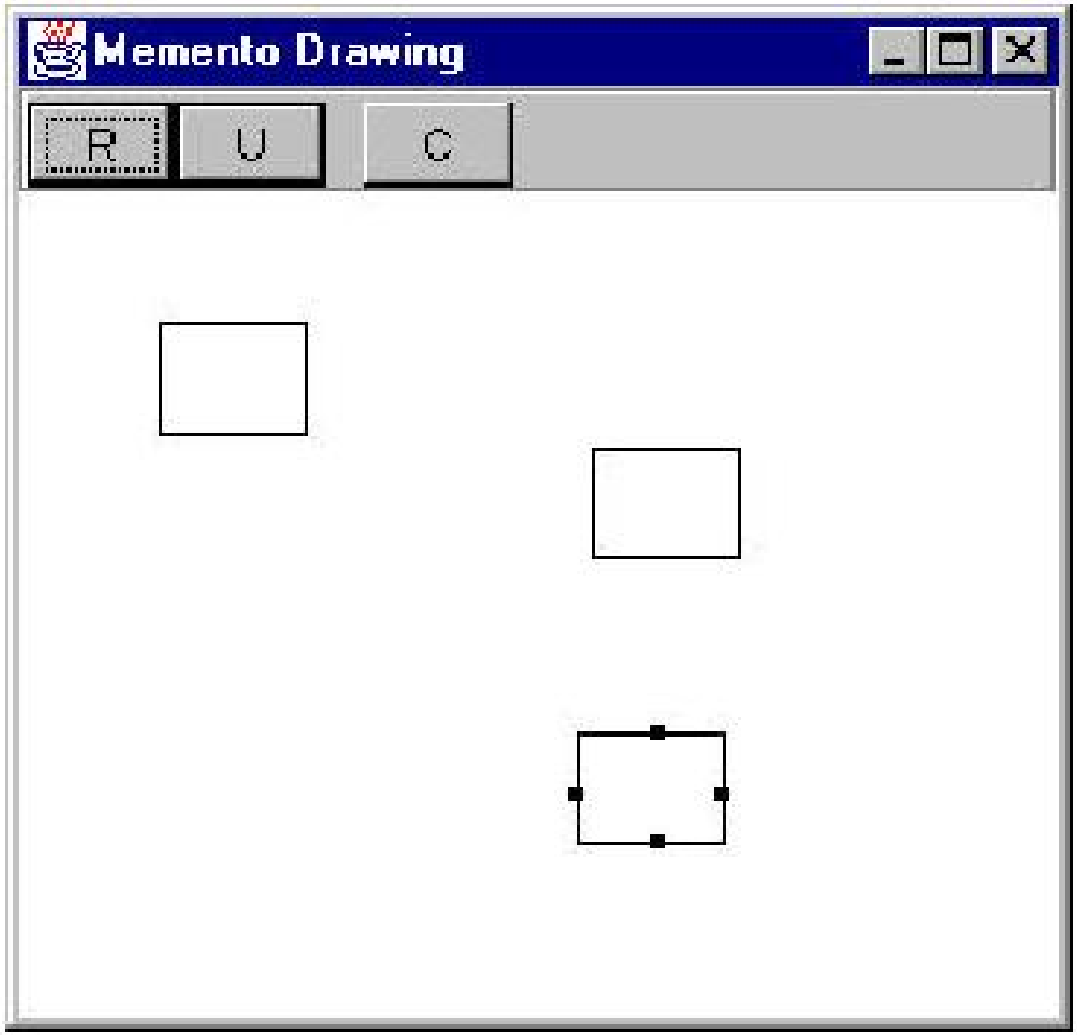
Example



Example



Example



Example

- 5 user actions
  - Rectangle button click
  - Mouse click
  - Mouse drag
  - Undo button click
  - Clear button click
- 2 undo actions
  - Rectangle creation
  - Rectangle position change

Advantages and Disadvantage

- Advantages
  - The Memento provides a way to preserve the state of an object while preserving encapsulation
  - It preserves the simplicity of the Originator class by delegating the saving and restoring of information to the Memento class
- Disadvantage
  - The amount of information that a Memento has to save might be quite large, thus taking up fair amounts of storage