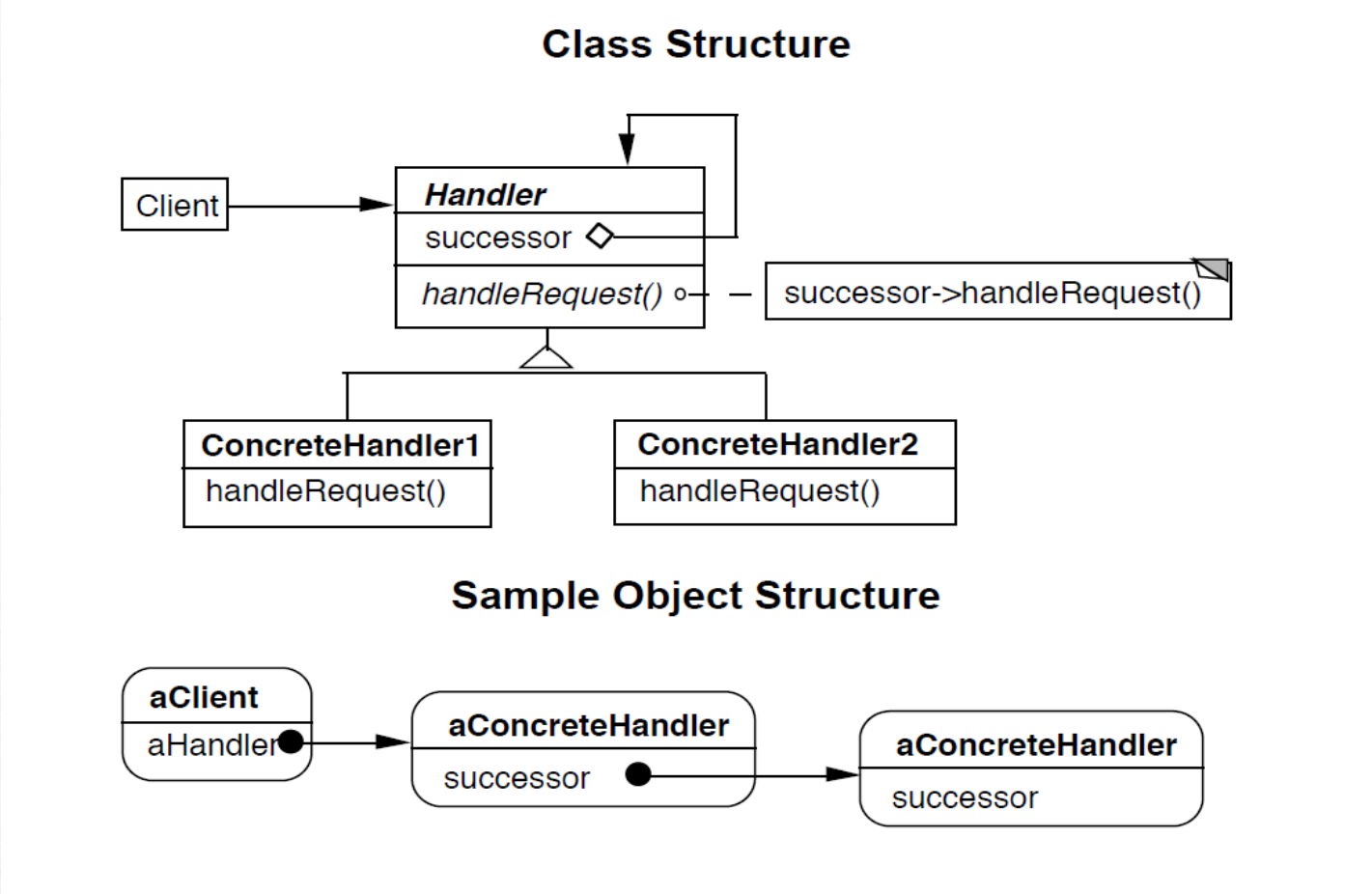
CHAIN OF RESPONSIBILITY

- Intento

  Evitare di accoppiare il mandante di una richiesta con il ricevente, dando così la possibilità a più di un oggetto di gestire la richiesta.

  Concatena gli oggetti riceventi e passa la richiesta tra questi fino a che un oggetto la gestisce.

  Disaccoppiare mandante e ricevente



Partecipanti

Handler

- Definisce l'interfaccia per gestire le richieste.
- può avere un riferimento ad un successore

ConcreteHandler

- gestore delle richieste
- può accedere ad un suo successore
- gestisce la richiesta se può farlo, altrimenti la inoltra al successore

Client

- Origina la richiesta inviandola ad un concreteHandler

Conseguenze

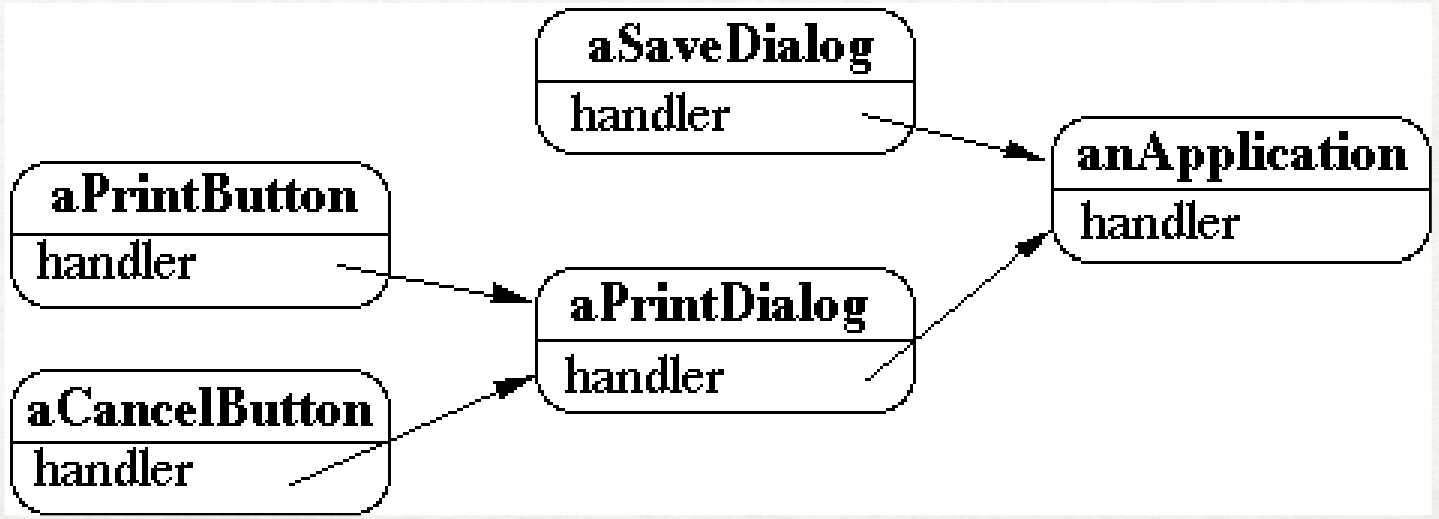
- Riduce l'accoppiamento
    - chi fa una richiesta non conosce il ricevente e viceversa
    - Un oggetto della catena non deve conoscere la struttura della catena
    - Gli oggetti handler anziché mantenere i riferimenti a tutti i candidati riceventi mantengono solo il riferimento al successore
- Si aggiunge flessibilità nella distribuzione di responsabilità agli oggetti
    - Si può cambiare o aggiungere responsabilità nella gestione di una richiesta cambiando la catena a runtime
- Non c'è garanzia che una richiesta venga gestita, poiché non c'è un ricevente esplicito, la richiesta potrebbe arrivare alla fine della catena senza essere gestita

Motivazione

- Sistemi di Help Contestuale

Consideriamo una interfaccia utente dove l'utente può richiedere aiuto su parti dell'interfaccia. Per es. un bottone può fornire informazioni di aiuto. Se non esiste un'informazione specifica allora il sistema dovrebbe fornire il messaggio d'aiuto del contesto più vicino (ad es. la finestra)

l'oggetto che fornirà il messaggio d'aiuto non è conosciuto dall'oggetto (es. PrintButton) che inizia la richiesta



Applicabilità

- Usa Il patter chain quando….
- Quando più di un oggetto potrebbe gestire una certa richiesta e non è noto a priori chi dovrà gestirla (polimorfismo).
- Quando vuoi inoltrare una richiesta a un oggetto presente in un set di vari oggetti, senza indicare il destinatario esplicitamente (accoppiamento lasco)
- Quando il set di oggetti che può gestire una (o più) richieste deve essere specificabile/modificabile dinamicamente

Chain vs Decorator

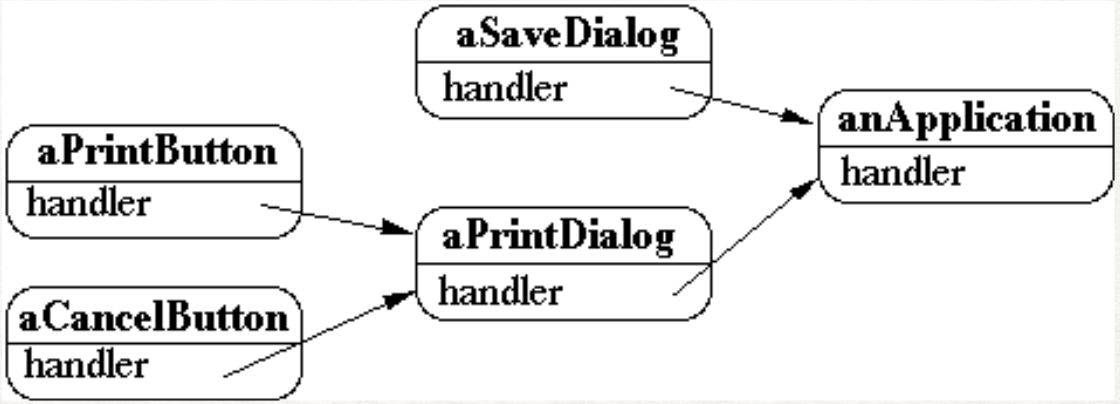
La catena di oggetti non è lineare ma più genericamente ad albero

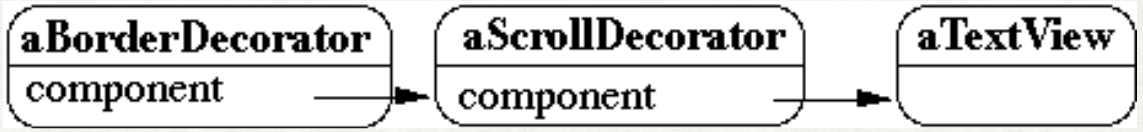
Un solo oggetto gestisce la richiesta, tutti gli altri la inoltrano

Il **primo** oggetto attraversato che implementa il metodo richiesto, lo esegue e interrompe l'inoltro

Nel decorator, **tutti** contribuiscono in proprio alla esecuzione della richiesta

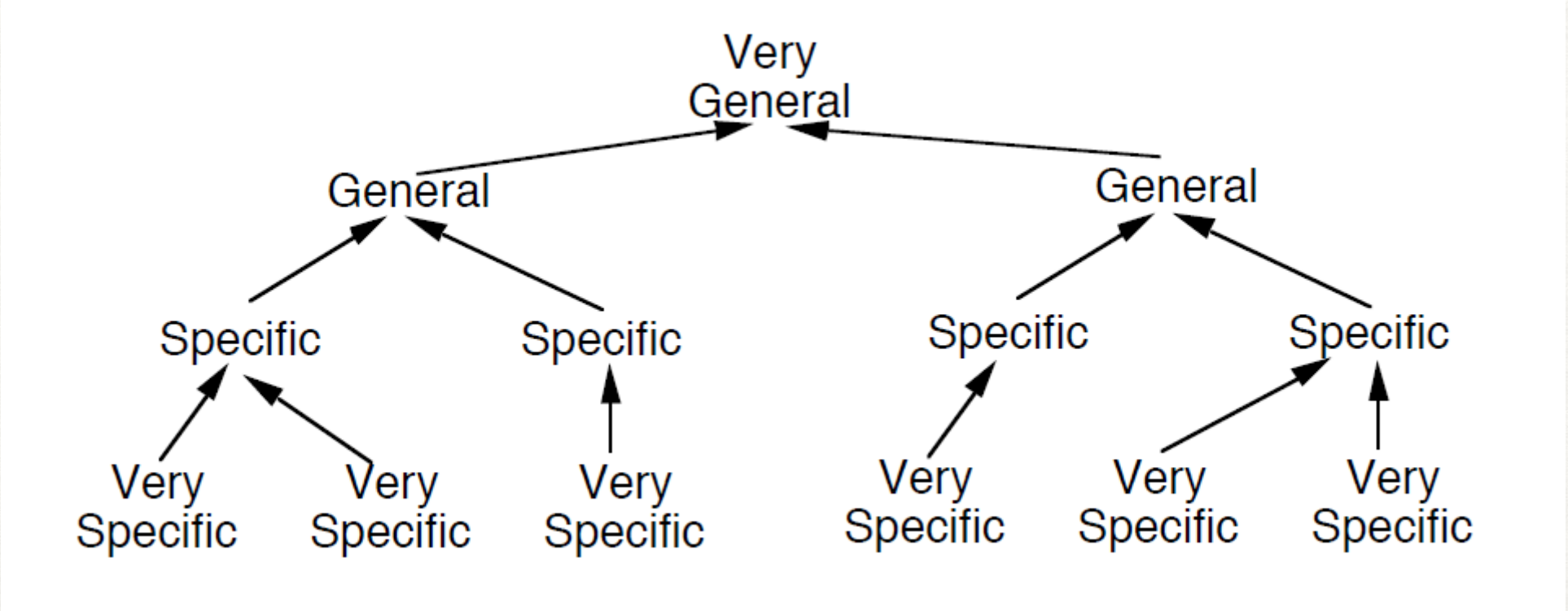
Non c'è un ordinamento gerarchico tra gli oggetti, anzi volgio proprio poterli combinare liberamente in varie sequenze





Gerarchica di oggetti

- Come nelle organizzazioni complesse: una qualsiasi richiesta, originata dal basso, sale la catena di comando fino a trovare chi ha l'autorità di rispondervi. Più salgo la gerarchia, più generiche sono le capacità.



La catena di responsabilità

- Posso usare riferimenti già esistenti tra gli oggetti
  - se ci sono già e corrispondono ad una gerarchia
  - se tale gerarchia coincide con una responsabilità crescente,
  - Tipico delle strutture dati composite (vedi il pattern *composite*)
- Altrimenti definisco riferimenti aggiuntivi per i successori

Rappresentare le richieste

Richieste *cablate*:

Ogni possible richiesta ha una segnatura specifico ben precisa e diversa dalle altre.

Tutte le possibili richieste gestibili devono essere raccolte nell'interfaccia comune.

Il set di richieste è noto a priori e fissato

```
abstract class HardCodedHandler {
    private HardCodedHandler successor;

    public HardCodedHandler(HardCodedHandler aSuccessor) {
        successor = aSuccessor;
    }

    public void handleOpen() {
        successor.handleOpen();
    }

    public void handleClose() {
        successor.handleClose();
    }

    public void handleNew(String fileName) {
        successor.handleClose(fileName);
    }
}
```

Rappresentare le richieste

Singolo metodo *generico*:

Handler ha un solo metodo, con un parametro che codifica le differenti richieste

Il set di richieste non è fissato

Richiedente e riceventi prendono accordi sulla codifica delle richieste.

```
abstract class SingleHandler {
    private SingleHandler successor;

    public SingleHandler(SingleHandler aSuccessor) {
        successor = aSuccessor;
    }

    public void handle(String request) {
        successor.handle(request);
    }
}
```

Rappresentare le richieste

Singolo metodo *generico:*

Handler ha un solo metodo, con un parametro che codifica le differenti richieste

Nuovi oggetti possono aggiungere supporto per codici addizionali corrispondenti a nuove richieste

Più controlli a runtime

```
class ConcreteOpenHandler extends SingleHandler {
    public void handle(String request) {
        switch (request) {
            case "Open": // do the right thing;
            case "Close": // more right things;
            case "New": // even more right things;
            default:
                successor.handle(request);
        }
    }
}
```

Rappresentare le richieste

Singolo metodo *con parametro oggetto:*

Handler ha un solo metodo, con un parametro oggetto che incapsula la richiesta

Il set di richieste non è fissato

L'oggetto richiesta contiene le info che specificano la richiesta

La esamino a run time per capire il tipo di richiesta e, se sono in grado, processarla

possibile sottoclassare e usare la reflection

Sempre controlli a runtime

```
abstract class SingleHandler {
    private SingleHandler successor;

    public SingleHandler(SingleHandler aSuccessor) {
        successor = aSuccessor;
    }

    public void handle(Request data) {
        successor.handle(data);
    }
}

class ConcreteOpenHandler extends SingleHandler {
    public void handle(Open data) {
        // handle the open here
    }
}
```

Rappresentare le richieste

Singolo metodo *con parametro oggetto che Incapsula la richiesta:*

Esempio di richieste

```
class Request {
    private int size;
    private String name;

    public Request(int mySize, String myName) {
        size = mySize;
        name = myName;
    }

    public int size() {
        return size;
    }

    public String name() {
        return name;
    }
}

class Open extends Request {// add Open specific stuff here }
class Close extends Request { // add Close specific stuff here}
```

Delega ricorsiva

- Il pattern implementa di fatto una ***ricorsione*** orientata agli oggetti
  - La chiamata di un metodo corrisponde all'inoltro (polimorfico) della stessa chiamata al prossimo (diverso) ricevente
  - Alla fine una di queste chiamate corrisponderà all'esecuzione effettiva del metodo
  - La ricorsione quindi termina e si riavvolge all'indietro fino a tornare al chiamante originale.

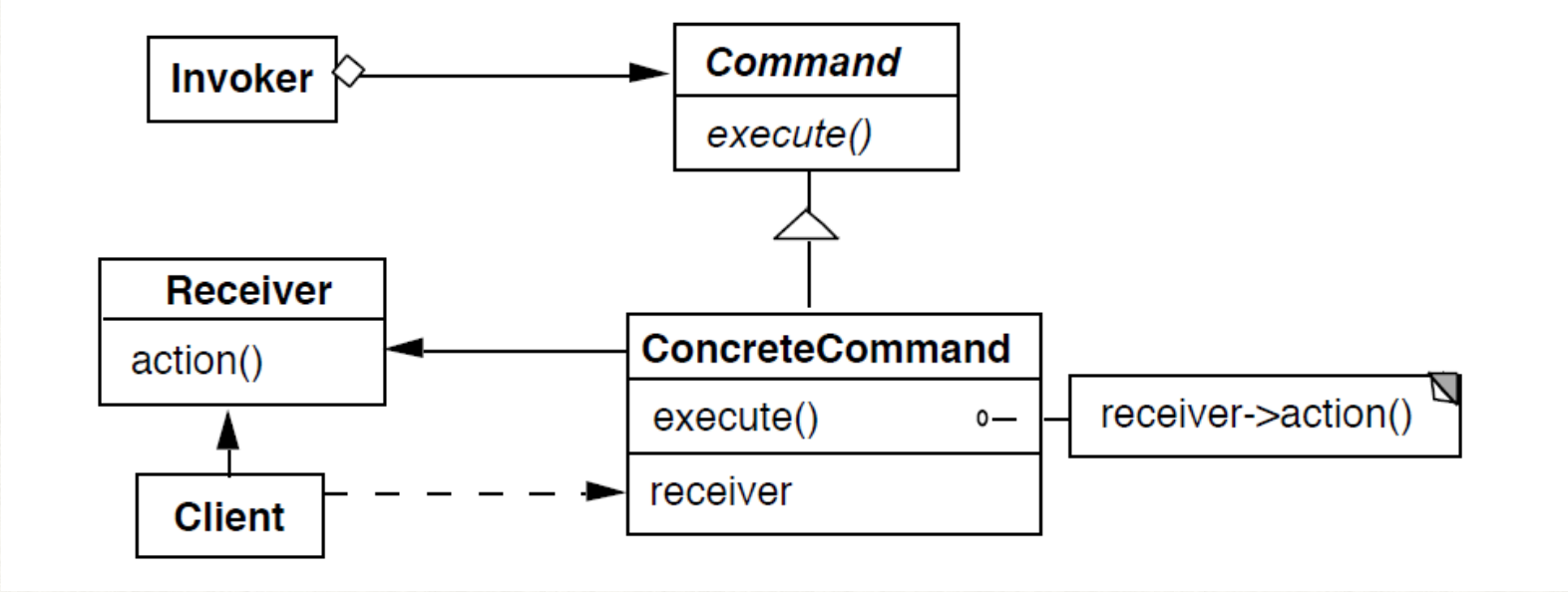
esempio

- Esempio di ricorsione: alberi binari
  - la ricerca di una chiave è un esempio di come seguendo gerarchicamente I riferimenti esistenti giungo al nodo (foglia o radice del sottoalbero) che sa darmi la risposta cercata.
  - La stampa dell'albero è invece un esempio di ricorsione object oriented che effettua la visita completa.
- Esempio codice in applicazione BinaryTree

- Pausa 10 min

Command

- Intento: incapsulare una richiesta in un oggetto
- Struttura



- Esempio:
  - Receiver sia un oggetto che rappresenta un Documento word
  - invoker sia un menù, cui corrisponde un comando astratto *execute()*
  - il Client, ad es. un applicazione di word processing, lo rimpiazza con una istanza di comando specifico *action(),* ad es. document.save(), rivolto ad una istanza di documento (quello aperto)
  - L'azione scatta dal menù, ma **quale e per chi è deciso a run-time** dal client

Gestire un Menù

- Quando scrivo il codice di un framework grafico implemento la classe *menù*
- *Tuttavia* non voglio e non posso fissare le voci del menu e le corrispondenti azioni una volta per tutte, perché il framework serve proprio a supportare tante applicazioni future.
- *Collego* alle voci dei menù dei comandi astratti che, a run-time, saranno sostituiti da oggetti che incapsulano l'azione richiesta ed il suo ricevente.
- È l'applicazione e non il framework ad istanziare le azioni (oggetti command) e a configurare i menu, assegnandole in corrispondenza delle varie voci

Applicabilità

- Uso il pattern command:
  - Invece di definire funzioni callback staticamente: incapsulo l'azione in un parametro *Comando gestibile a run-time*
  - Per avere in tempi diversi la definizione, l'invocazione (metto in coda) e l'esecuzione di comandi
  - Per supportare il log delle richieste eseguite e quindi dei cambiamenti
  - Per supportare l'UNDO, ripercorrendo il log e annullando le modifiche a ritroso
  - Per offrire operazioni di alto livello, da comporre su operazioni primitive
  - Per implementare un sistema transazionale (tipico nei DB)
  - Per implementare il supporto alle macro

conseguenze

- Il pattern disaccoppia l'oggetto che invoca una operazione da quello che sa eseguirla
- È facile cambiare/aggiungere nuovi comandi, perché non devi cambiare le classi esistenti
- Puoi creare comandi complessi assemblando insieme vari comandi in un oggetto composito

Implementazione

```
abstract class Command {
    abstract public void execute();
}

class OpenCommand extends Command {
    private Application opener;

    public OpenCommand(Application theOpener) {
        opener = theOpener;
    }

    public void execute() {
        String documentName = AskUserSomeHow();
        if (name != null) {
            Document toOpen = new Document(documentName);
            opener.add(toOpen);
            opener.open();
        }
    }
}
```

Implementazione

```
class Menu {
    private Hashtable menuActions = new Hashtable();

    public void addMenuItem(String displayString,
            Command itemAction) {
        menuActions.put(displayString, itemAction);
    }

    public void handleEvent(String itemSelected) {
        Command runMe;
        runMe = (Command) menuActions.get(itemSelected);
        runMe.execute();
    }
    // lots of stuff missing
}
```

Macro command

```
class MacroCommand extends Command {
    private Vector commands = new Vector();

    public void add(Command toAdd) {
        commands.addElement(toAdd);
    }

    public void remove(Command toRemove) {
        commands.removeElement(toAdd);
    }

    public void execute() {
        Enumeration commandList = commands.elements();
        while (commandList.hasMoreElements()) {
            Command nextCommand;
            nextCommand = (Command) commandList.nextElement();
            nextCommand.execute();
        }
    }
}
```

Pluggable commands

```
import java.lang.reflect.*;

public class Command {
    private Object receiver;
    private Method command;
    private Object[] arguments;

    public Command(Object receiver, Method command, Object[] arguments) {
        this.receiver = receiver;
        this.command = command;
        this.arguments = arguments;
    }

    public void execute() throws InvocationTargetException,IllegalAccessException {
        command.invoke(receiver, arguments);
    }
}
```

Pluggable commands

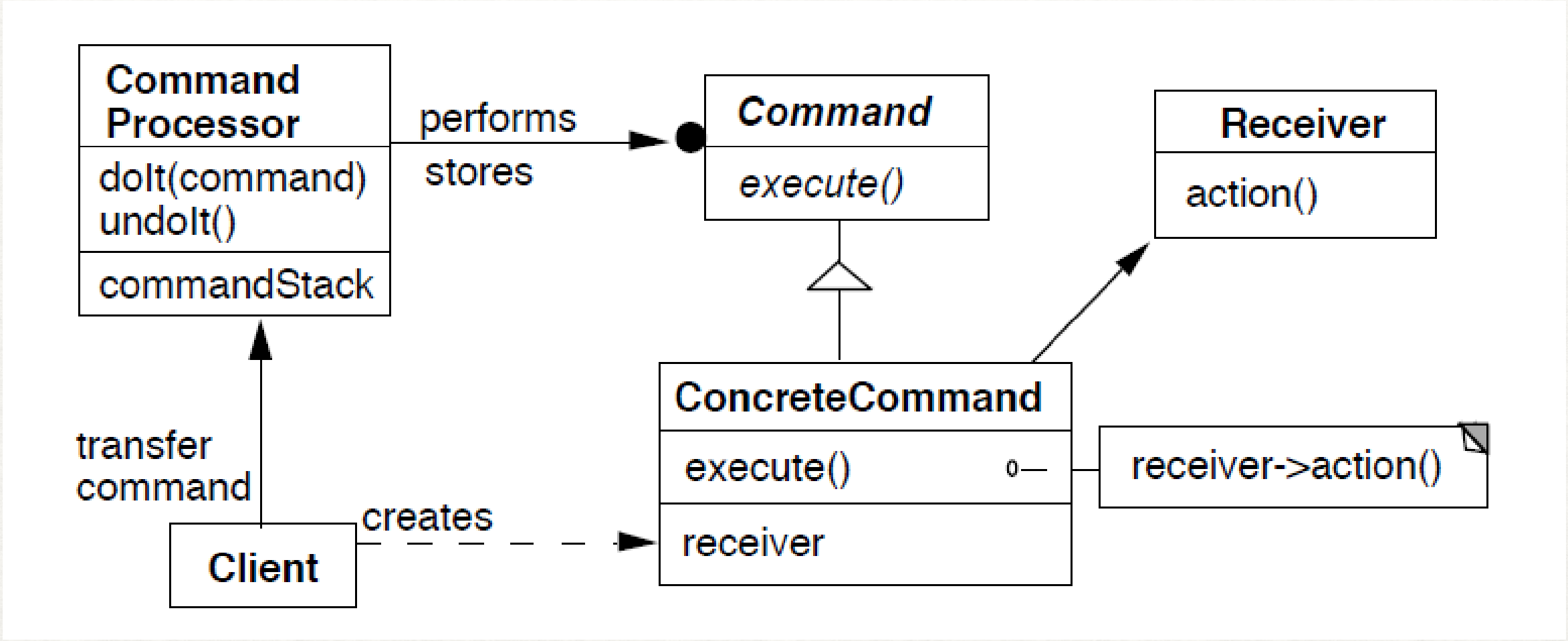
```
import java.util.*;
import java.lang.reflect.*;

public class Test {
    public static void main(String[] args) throws Exception {
        Vector sample = new Vector();
        Class[] argumentTypes = { Object.class };
        Method add = Vector.class.getMethod("addElement", argumentTypes);
        Object[] arguments = { "cat" };
        Command test = new Command(sample, add, arguments);
        test.execute();
        System.out.println(sample.elementAt(0));
    }
}
```

Grazie alla reflection posso definire una sola classe *Command* invece di una gerarchia e configurarla per incapsulare qualsivoglia invocazione di metodo su qualsiasi oggetto (vedi demo)

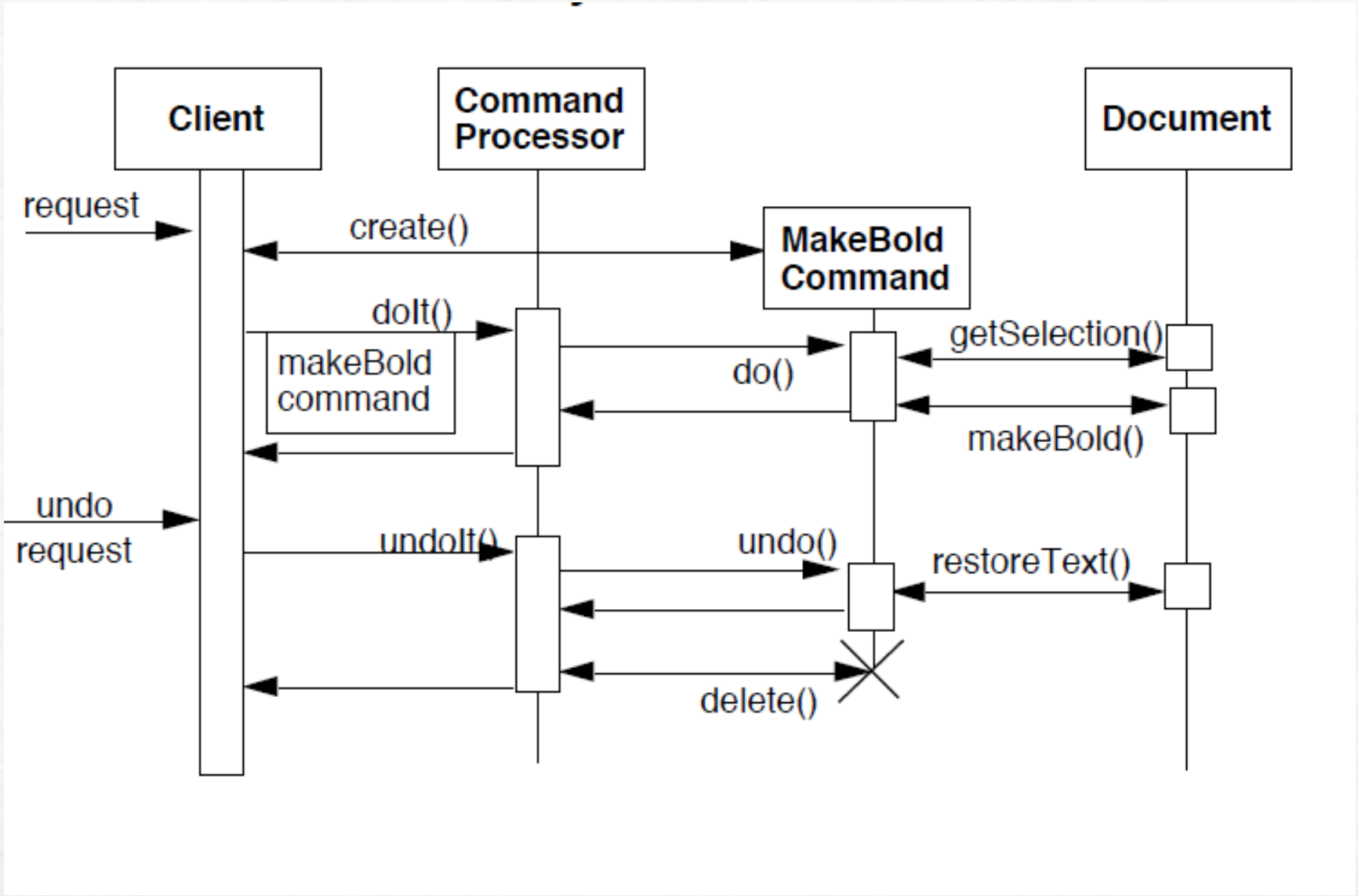
Command Processor

- Simile al pattern command, ma con in più la presenza di un gestore degli oggetti comando
  - Il gestore custodisce tutte le istanze dei comandi
  - Schedula la loro esecuzione
  - Può conservarli anche dopo, per eventuale undo
  - Può fare il log della sequenza d'esecuzione (per test/debug)
  - Uso il pattern singleton per imporre un gestore unico

Struttura


Command Processor
doIt(command)
undoIt()
commandStack
performs
stores
Command
execute()
Receiver
action()
ConcreteCommand
execute()
receiver
0
receiver->action()
transfer command
creates
Client

Dinamica


Client
Command Processor
Document
request
create()
MakeBold Command
doIt()
makeBold command
do()
getSelection()
makeBold()
undo request
undoIt()
undo()
restoreText()
delete()

Conseguenze

- Flessibilità nell'esecuzione delle richieste
  - Gli stessi comandi possono essere generati da elementi diversi dell'interfaccia utente
  - L'utente stesso potrebbe configurare le azioni corrispondenti agli elementi di interfaccia
- Flessibilità nel numero e nella funzionalità delle richieste
  - Aggiungere nuovi comandi o definire macro è facile
- Flessibilità nell'esecuzione delle richieste
  - I comandi possono essere conservati per ripeterli in seguito
  - Possono essere loggati
  - Possono essere annullati
  - Possono essere eseguiti in parallelo su threads separati

Svantaggi

- Leggera perdita di efficienza per l'indirezione
- Potenziale perdita di efficienza per l'aumento delle classi
    - Evitabile usando command singolo con metodo generico e parametrico
- Aumento della complessità