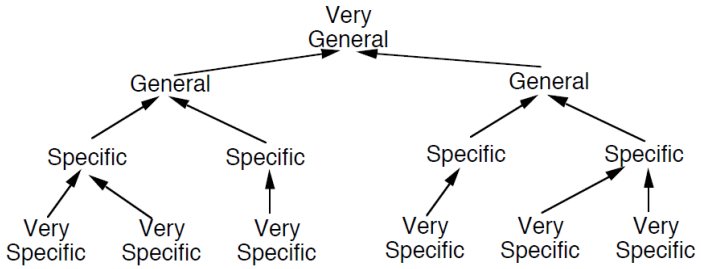
Chain of responsibility

Prof. A. M. Calvagna

Gerarchica di oggetti

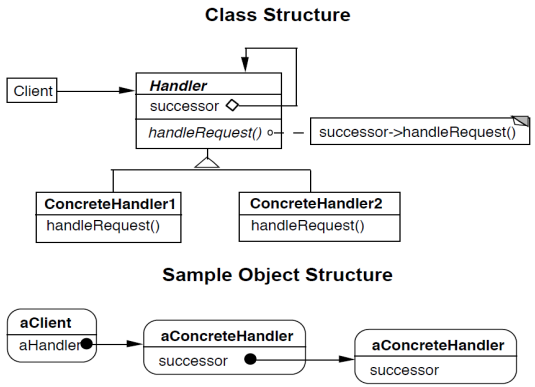


CHAIN OF RESPONSIBILITY

- Intento

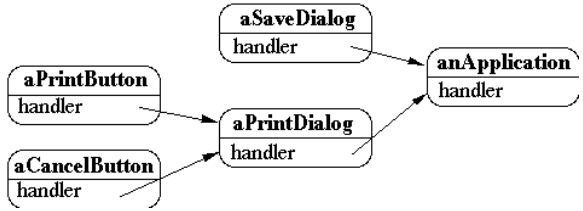
  Dare la possibilità a più di un oggetto di gestire una richiesta.

  Disaccoppiare mandante e ricevente



Motivazione

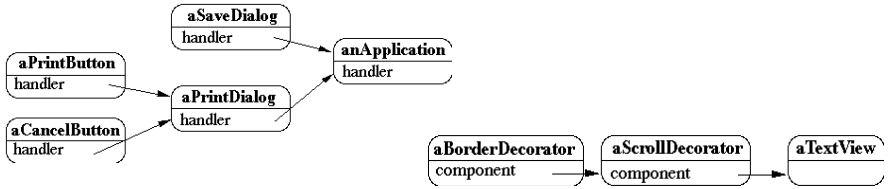
Help contestuale



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 **fuori gerarchia***).
- 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 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 appositi riferimenti aggiuntivi per i successori

Rappresentare le richieste

```
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

```
abstract class SingleHandler {
    private SingleHandler successor;

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

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


Rappresentare le richieste

```
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

```
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
    }
}
```


Esempio:

```
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}
```

Conseguenze

- **Riduce l'accoppiamento**
  - chi fa una richiesta non conosce il ricevente e viceversa
  - Un oggetto (handler) della catena non deve conoscerne la struttura
  - 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

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

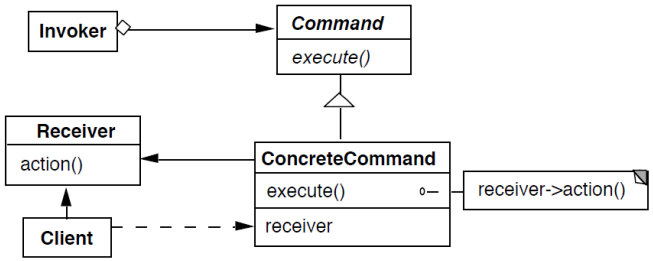
- Esempio: alberi binari di ricerca
  - la ricerca di una chiave è un esempio di ricorsione che seguendo gerarchicamente I riferimenti esistenti giunge al nodo (foglia o radice del sottoalbero) che sa darmi la risposta cercata.
  - La stampa dell'albero è un esempio di ricorsione object oriented che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

15 min. break

Command

Intento:

- Incapsulare la richiesta di una operazione in un oggetto
- disaccoppia del tutto l'oggetto che invoca una operazione da quello che la esegue



Motivazione

Implementare Framework

Es.: Gestire un Menù

- Il framework implementa la classe *menù*
- non fissa le voci del menu né le corrispondenti azioni
- L'applicazione decide le voci e gli abbina le azioni
- È il set-up del framework

Implementazione (invoker)

```
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
}
```


Implementazione (concrete command)

```
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();
        }
    }
}
```

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** e **l'esecuzione** di comandi (vedi anche "command processor")
  - Per **creare comandi complessi** assemblando insieme varie azioni in un oggetto composito
- **Pluggable commands**: definisco una sola classe command, configurabile

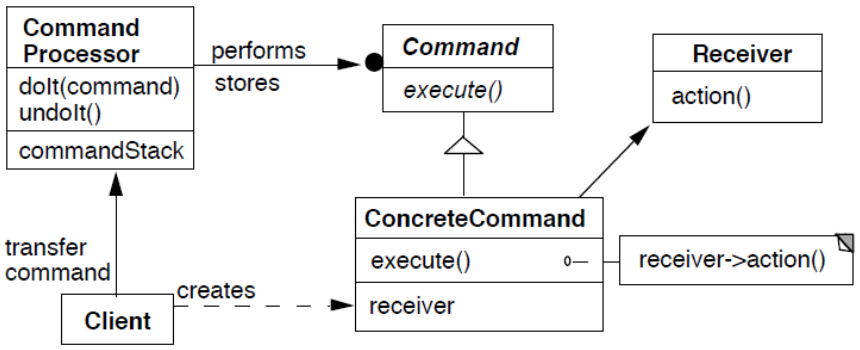
Pluggable commands

```
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));
    }
}
```

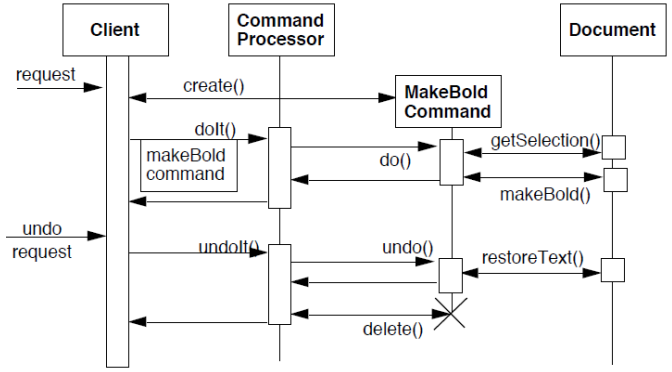

Command Processor



Command Processor

- oggetto che riceve e gestisce richieste di azioni diverse come **parametri**
  - può **schedularne** l'esecuzione con una sua politica
  - Può fare il **log** della sequenza d'esecuzione (per test/debug)
  - Può **conservare** le istanze di **tutti** i diversi comandi richiesti, per eventuale **undo**
  - Può implementare il supporto alle **macro**
  - Può implementare un sistema **transazionale** (tipico nei DB)
  - Uso il pattern singleton per imporre un **gestore unico**

Dinamica dell'undo



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();
        }
    }
}
```


Conseguenze

- **Flessibilità nell'invocazione** delle richieste
  - Gli stessi comandi possono essere generati da elementi diversi (ad es. 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 pluggable command
- Aumento della complessità
  - Non è chiaro nell'invoker l'effetto dell'invocazione