CORSO DESIGN PATTERN

I pattern Behavioral

Dott. Romina Fiorenza
fiorenza.romina@gmail.com

Pattern Behavioral

- **Scopo dei pattern comportamentali**
  - Affrontare problemi che riguardano il **modo**
    - con cui un oggetto **svolge** la sua funzione

    oppure

    - un insieme di oggetti **collaborano** tra loro per raggiungere un dato obiettivo

Elenco Pattern

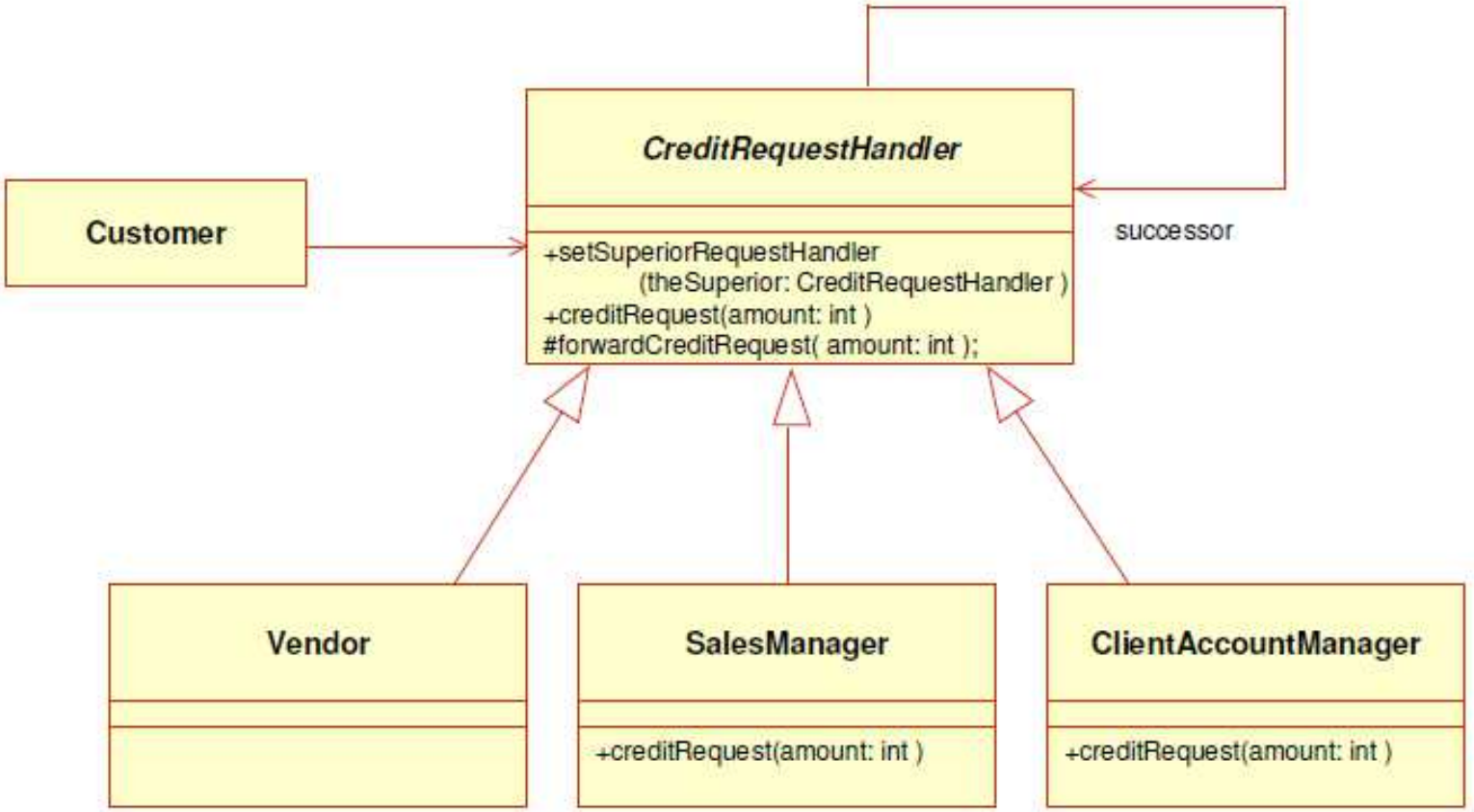
1. Chain of Responsability
2. Iterator
3. Command
4. Observer
5. Mediator
6. State
7. Strategy
8. Memento
9. Template method
10. Visitor
11. Interpreter

1) Chain of Responsibility

- **Esempio:**
  - Si vuole descrivere un team di venditori di un'azienda.
  - Il team prevede una struttura gerarchica: ogni venditore ha un suo superiore
  - E' inoltre fissata la regola per cui

    Al crescere dell'ammontare della vendita, bisogna passare la richiesta al collega che sta più in alto nella scala gerarchica

1) Chain of Responsibility



1) Chain of Responsability

Partecipanti

- Handler (CreditRequestHandler)
  - Definisce l'interfaccia per trattare la richiesta
  - Ha un riferimento al successor → superiore *generico*
  - Prevede il metodo protetto forward, per inoltrare la richiesta

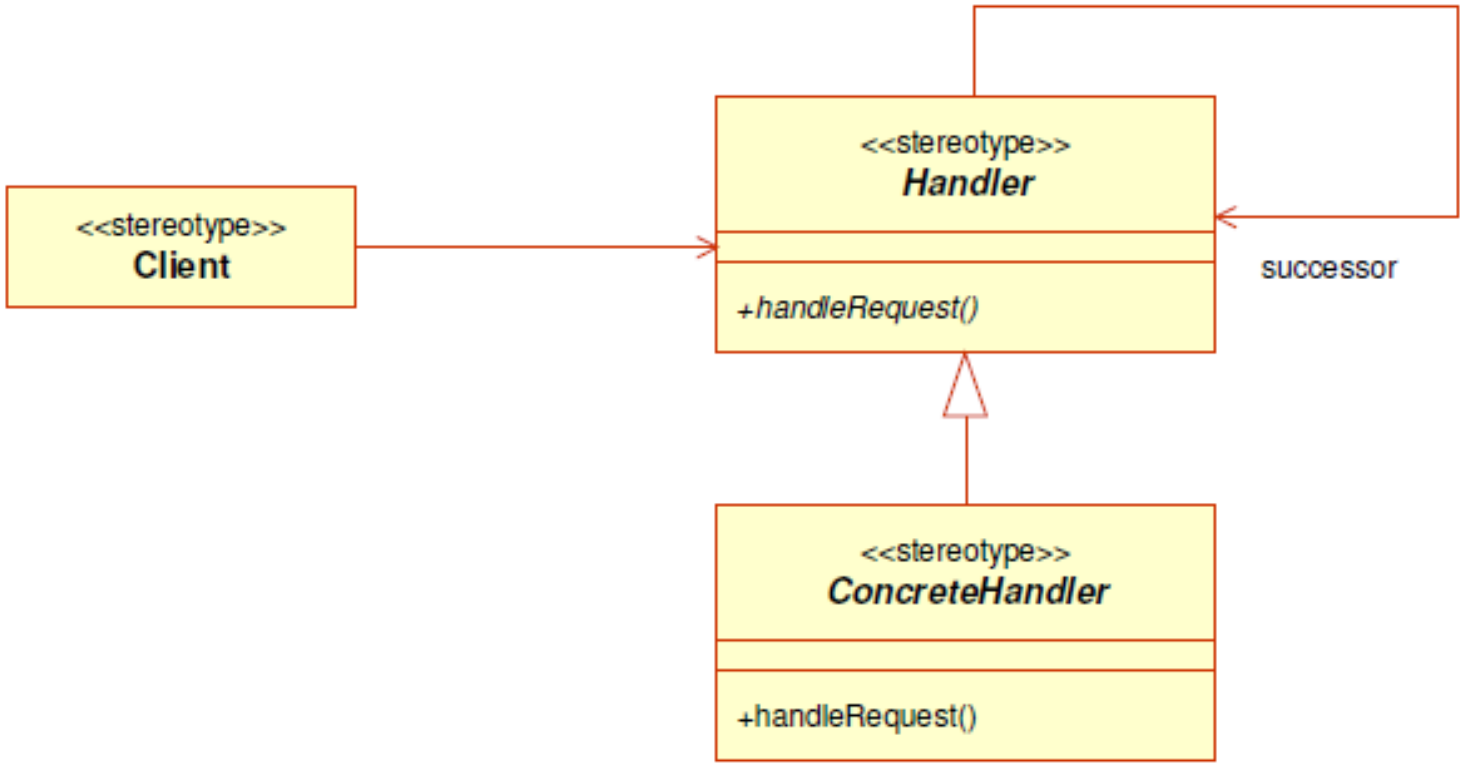
- ConcreteHandler (Vendor, SalesManager, ClientAccountManager)
  - Può accedere al successor → superiore *specifico*
  - Tratta le richieste di cui è responsabile (poiché conosce range ammontare)
  - Se non può trattare la richiesta, la inoltra al successor → metodo forward

- Client (Customer)
  - Sottopone la richiesta ad un ConcreteHandler
  - Non conosce il reale venditore che si occuperà di evadere la richiesta

1) Chain of Responsibility

- **Problema**
  - Separare il mittente di una richiesta dal suo ricevitore
  - Prevedere diversi ricevitori per gestire la richiesta
- **Soluzione**
  - Creare una catena di oggetti, i cui elementi sono in grado di
    - trattare la richiesta *oppure*
    - inoltrarla ad elementi della catena

1) Class diagram CoR



1) Chain of Responsibility

Conseguenze

- Il mittente non sa chi gestisce la sua richiesta (non è legato al suo Handler)
- Una richiesta potrebbe essere gestita da più Handler.
- Posso trattare facilmente nuove richieste
- Posso aggiungere nuovi Handler (estensibilità)
- E' possibile definire la priorità tra Handler

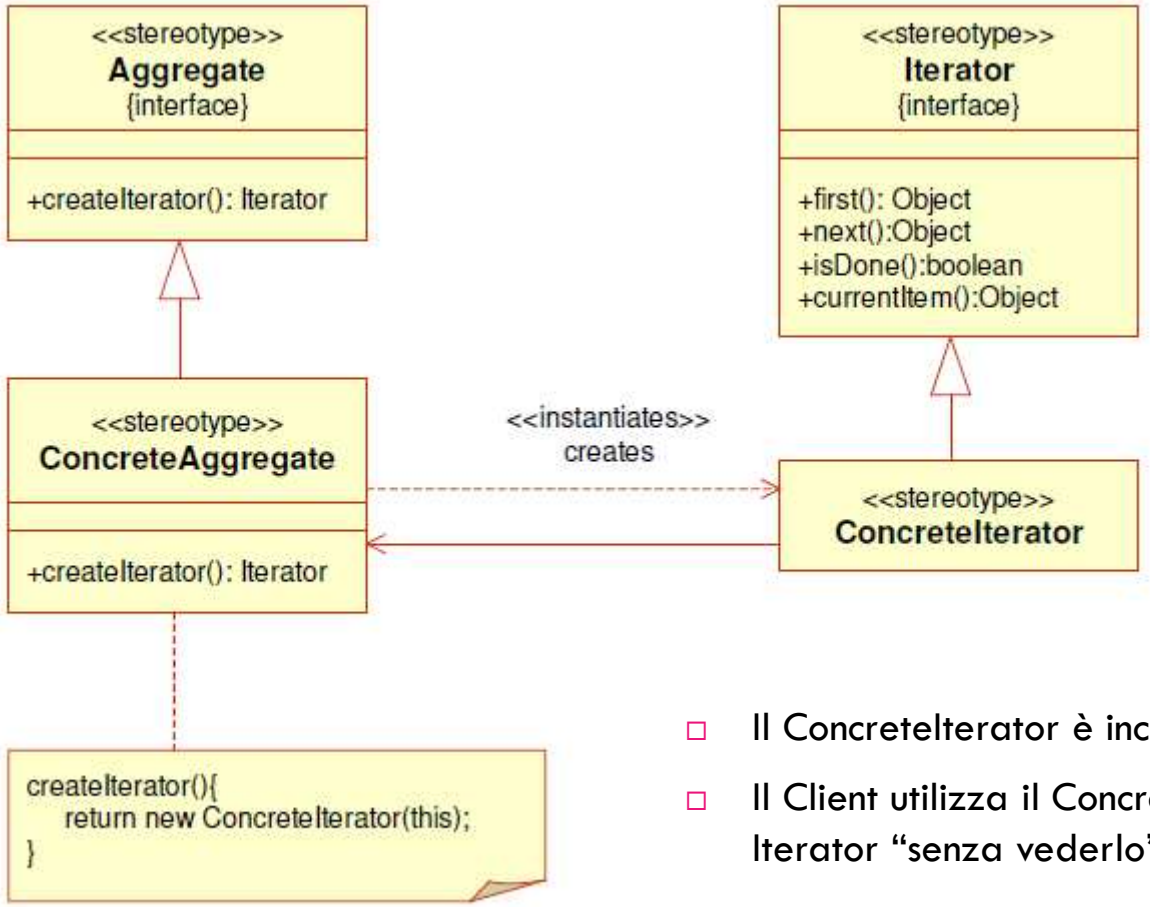
Note

- Se il numero di Handler è troppo elevato → overhead per scorrere la catena di Handler

2) Iterator

- **Problema**
  - Oggetti aggregati su cui operare in modo iterativo
  - Il client vuole ottenere gli oggetti senza conoscerne la rappresentazione interna (es. liste a puntatori, tavole hash, dizionari, ecc…)
- **Soluzione**
  - Definire 2 interfacce:
    - aggregatore → insieme
    - iteratore → navigatore
  - Definire 2 implementazioni →l'iteratore concreto conosce la rappresentazione interna dell'aggregatore concreto

2) Iterator Class Diagram



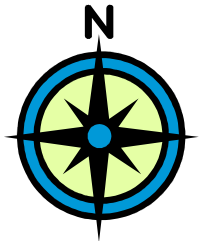
- Il ConcreteIterator è incapsulato e non visibile
- Il Client utilizza il ConcreteAggregator con il suo Iterator "senza vederlo"

2) Iterator

Esempio: L'**Iterator** di java.util

```
// creo un aggregatore concreto
List<Employee> lista = new ArrayList<Employee>();
lista.add(new Employee(…));
lista.add(new Employee(…));

// ciclo tramite un generico iteratore
Iterator iterator = lista.iterator();
while(iterator.hasNext()) {
   Employee e = iterator.next();
   System.out.print(e.getNome() + " guadagna ");
   System.out.println(e.getSalario());
}
```



2) Iterator

Partecipanti

- Iterator (Iterator)
  - Definisce un'interfaccia per attraversare la collection e recuperare gli elementi
- ConcreteIterator (Iterator "concreto")
  - Implementa l'interfaccia Iterator.
  - Tiene traccia della posizione corrente.
- Aggregate (List)
  - Definisce un'interfaccia per creare/ottenere l'Iterator
- ConcreteAggregate (ArrayList)
  - Restituisce un'istanza del ConcreteIterator abbinato al ConcreteAggregator

2) Iterator

Esempio: L'interfaccia **ResultSet** di java.sql

```
// preparo ed eseguo una query con JDBC
String sql = "select * from utenti where user = ?";
PreparedStatement pst = connection.prepareStatement(sql);
pst.setString(1,x);
ResultSet rs = pst.executeQuery();
// ciclo i risultati con un generico iteratore
while(rs.next()) {
   Utente utente = new Utente();
   utente.setUser(rs.getString("user"));
   utente.setPassword(rs.getString("password"));
   utente.setNome(rs.getString("nome"));
   utente.setCognome(rs.getString("cognome"));
}
```

2) Iterator

Conseguenze

- Iterator incapsula criterio di attraversamento quindi posso avere diversi iterator per la stessa aggregazione
  - diversi iterator possono indurre diverse navigazioni
  - differenti current item → a seconda dell'iterator scelto

- L'aggregazione non necessita di interfaccia di attraversamento quindi è più semplice e dinamica

3) Command

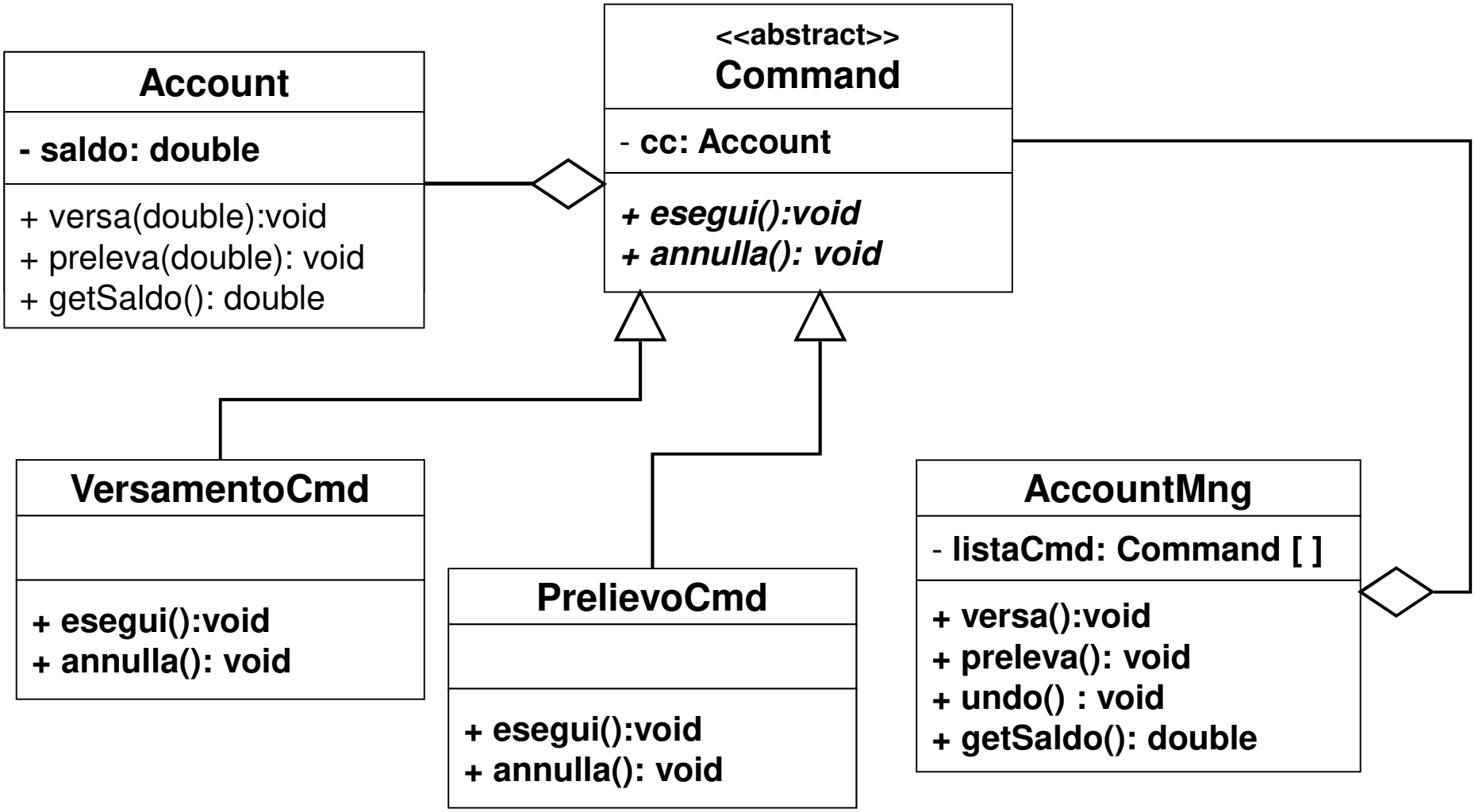
- **Esempio**
  - Una classe Account modella conti correnti.
  - Le funzionalità che si vogliono realizzare sono:
    - Prelievo
    - Versamento
    - Annullamento

    Questa operazione consente di annullare una delle precedenti, ma con il vincolo che l'annullamento deve avvenire con ordine cronologico inverso.

3) Command

- **Soluzione**
  - Modellare la classe Account solo con le funzioni di business → prelievo e versamento.
  - Creare un AccountMng per coordinare ad alto livello
    - le funzioni di business dell'Account
    - la funzionalità di annullamento.
  - L'AccountMng ha bisogno di classi che modellino i comandi che può eseguire → modellare una classe Command generica che prevede i generici
    - `esegui()` e `annulla()`
  - Infine fornire le specializzazioni per i comandi di prelievo e versamento → PrelievoCmd e VersamentoCmd

3) Command


Account
- saldo: double
+ versa(double):void
+ preleva(double): void
+ getSaldo(): double
<<abstract>>
Command
- cc: Account
+ esegui():void
+ annulla(): void
VersamentoCmd
+ esegui():void
+ annulla(): void
PrelievoCmd
+ esegui():void
+ annulla(): void
AccountMng
- listaCmd: Command [ ]
+ versa():void
+ preleva(): void
+ undo() : void
+ getSaldo(): double

3) Command

- **Problema**
  - Spesso un sistema deve eseguire delle operazioni che hanno per oggetto altre azioni sul sistema.

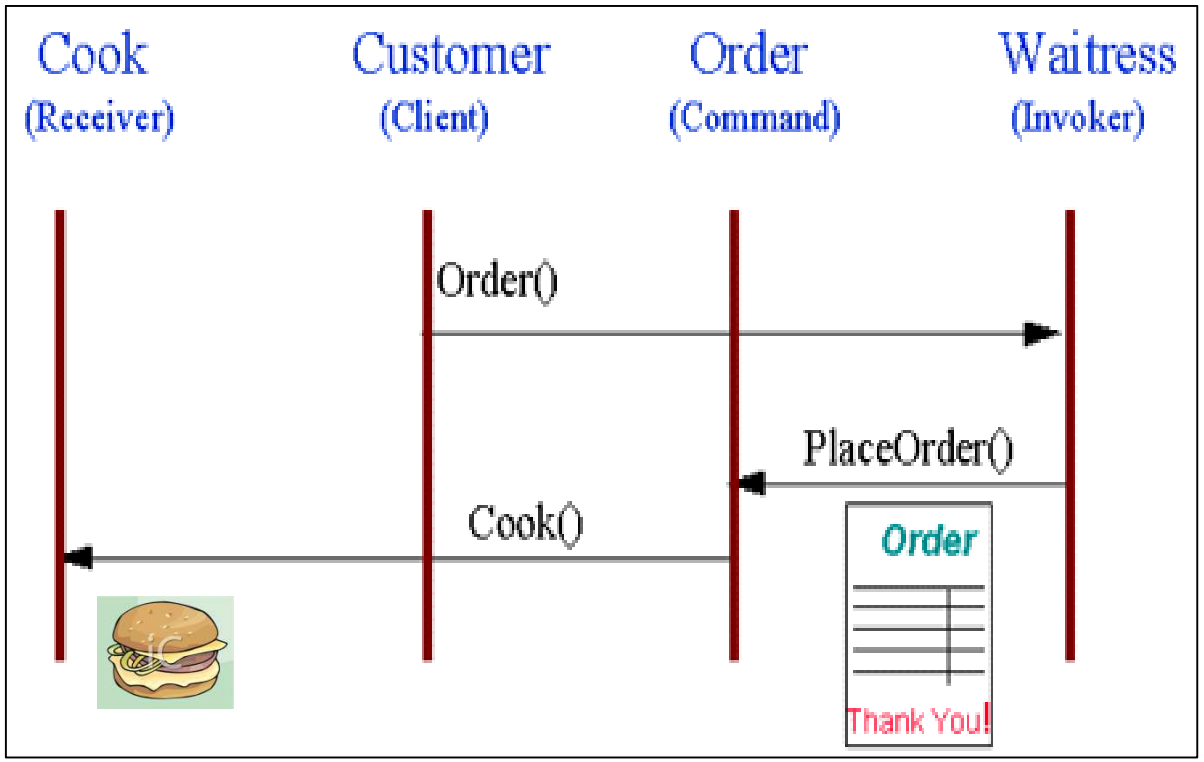
  - Esempi:
    - il comando Undo/Redo per annullare/ripetere un comando
    - il salvataggio di una sequenza di passi
    - aggiunta/rimozione comandi da una toolbar/menu

3) Command

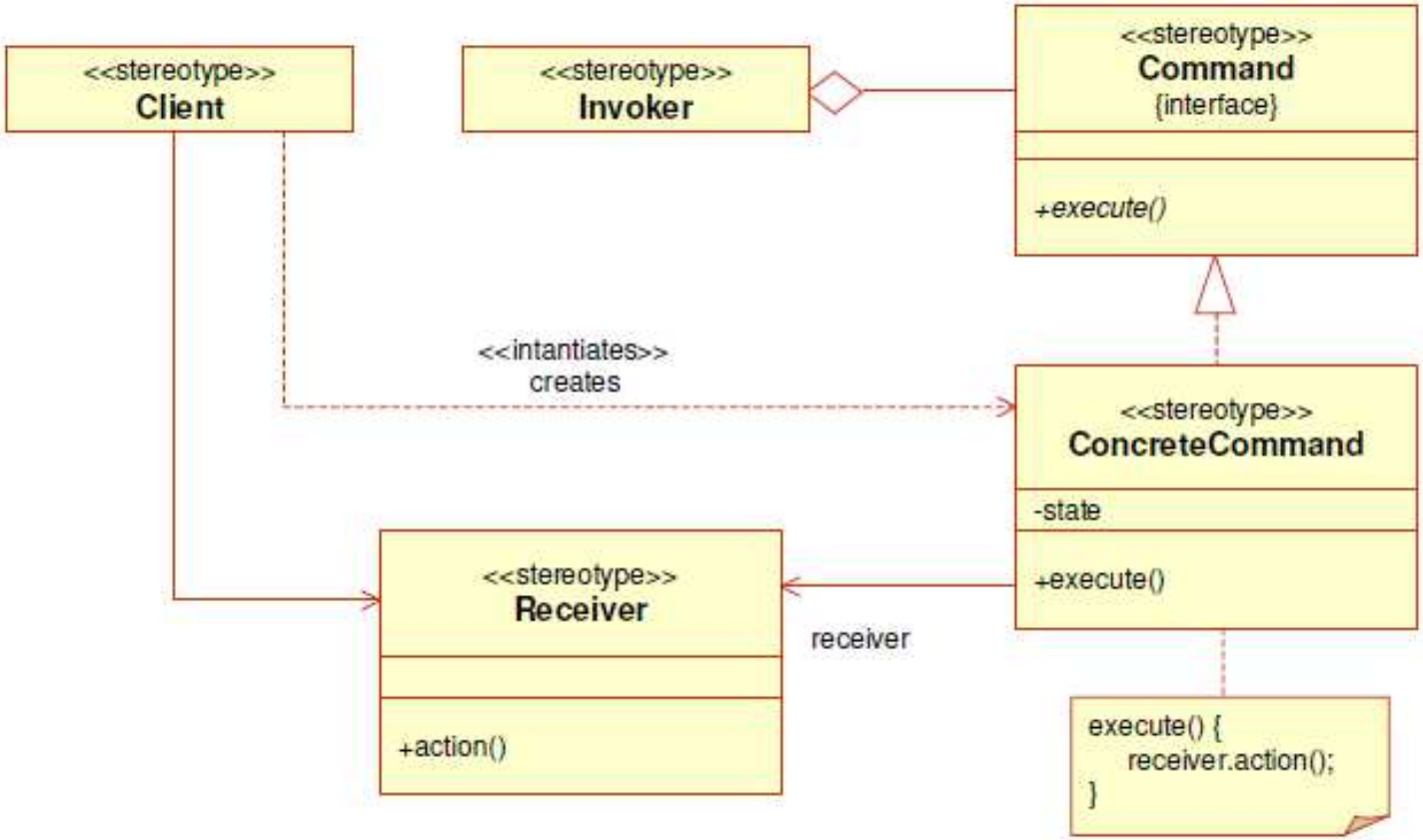
- **Soluzione**
  - Modellare le richieste come classi → Command
    - Un AbstractCommand per gestire comandi (annulla comando, ripeti comando, ecc)
    - Una serie di classi Command che realizzano il singolo comando
  - Creare 2 classi:
    - RequestCollector → che raccoglie la richiesta e invoca il
    - RequestHandler → riceve richiesta e eroga il servizio

3) Command

Esempio: Processo di ordinazione al pub



3) Command Class Diagram



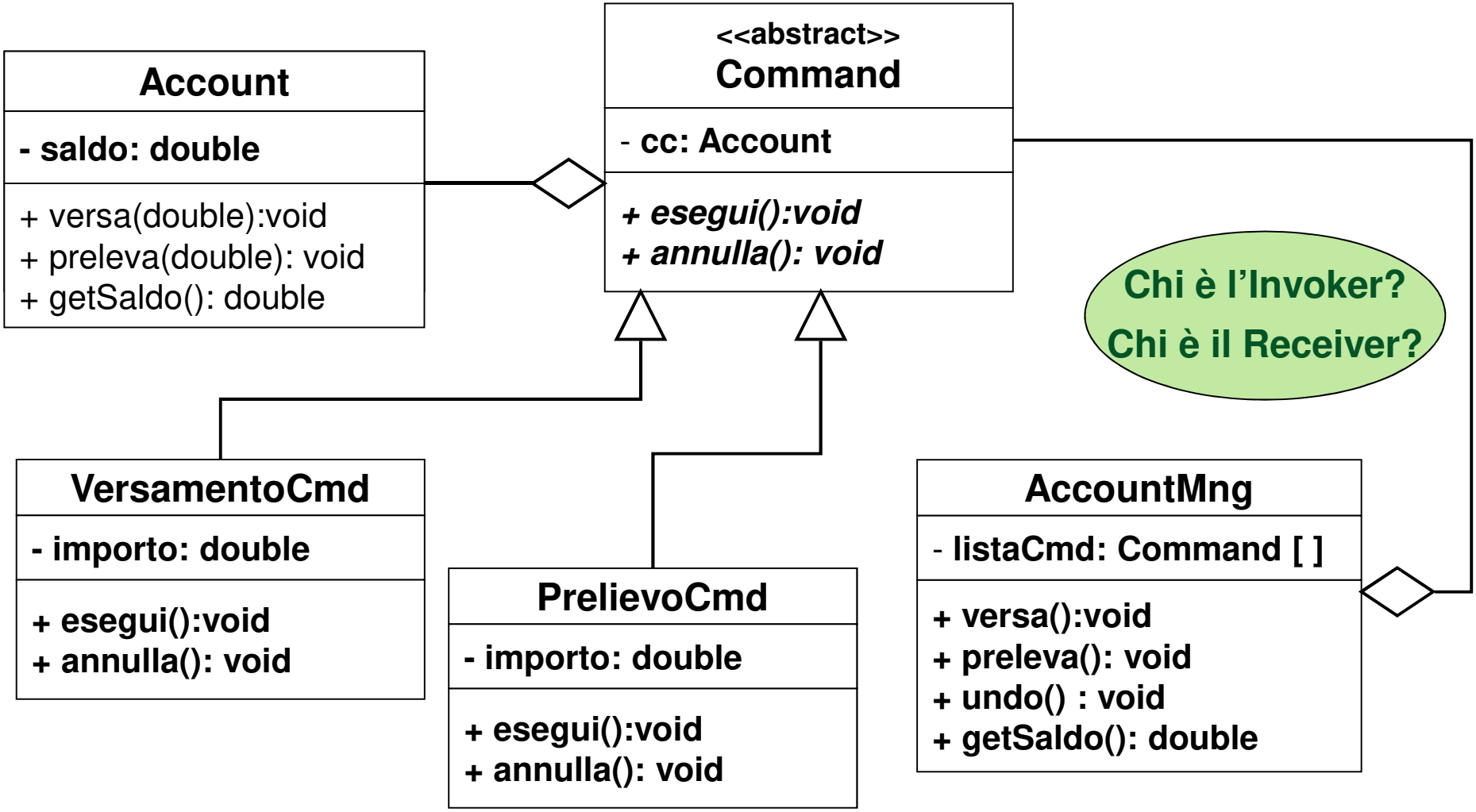
- Il **Client** esegue una richiesta concreta, tra quelle previste dall'**Invoker**
- La richiesta è evasa dal **Receiver**, coordinato dal **ConcreteCommand**

3) Command

Partecipanti

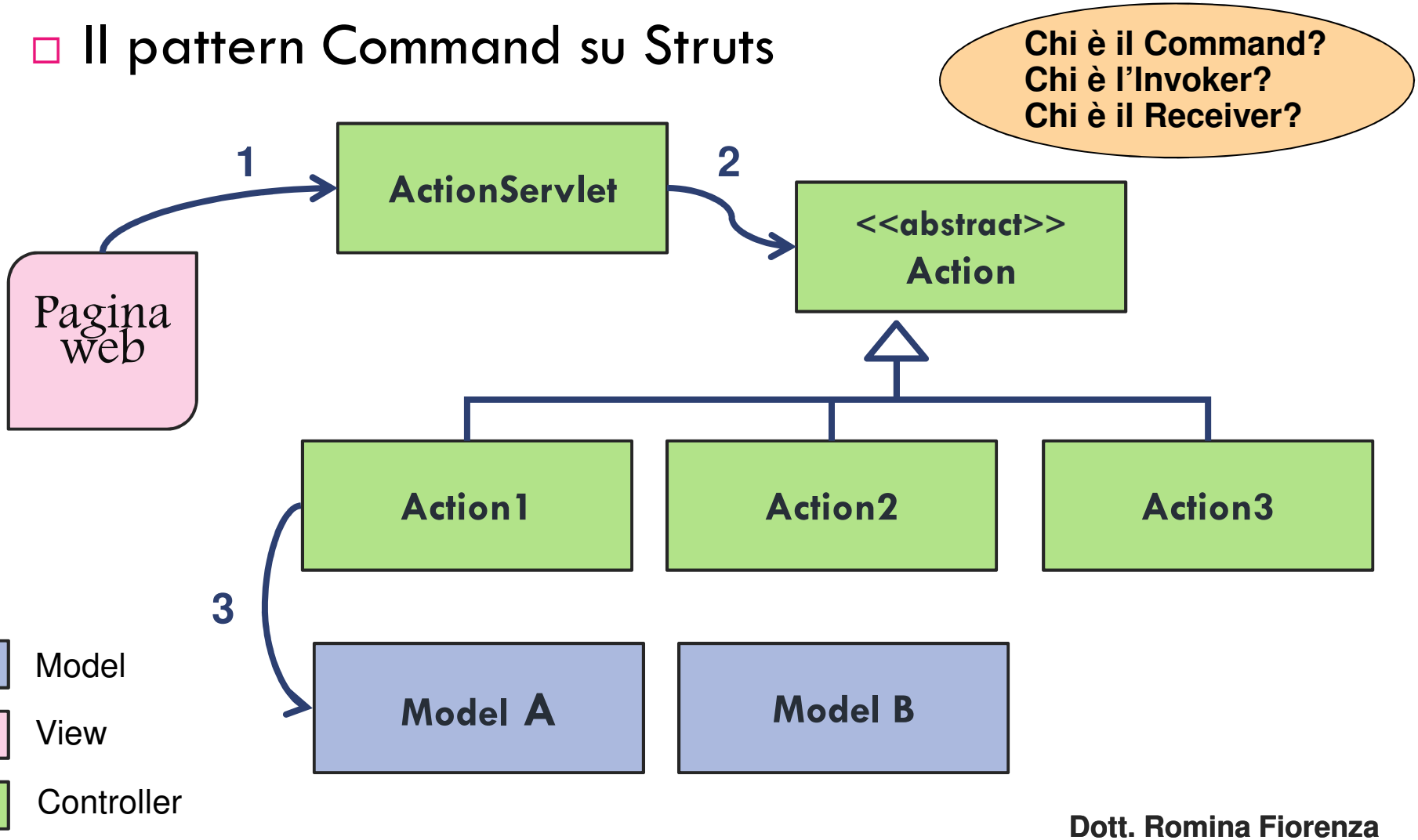
- Command
  - Dichiara una interfaccia per l'operazione → metodo *execute*
- ConcreteCommand
  - E' un binding tra il Receiver ed l'azione.
  - Implementa *execute* delegando il Receiver
- Client
  - Crea un ConcreteCommand e imposta il Recevier
- Invoker
  - Chiede al comando di eseguire la richiesta
- Receiver
  - Conosce come eseguire l'operazione associata alla richiesta

3) Command


Account
- saldo: double
+ versa(double):void
+ preleva(double): void
+ getSaldo(): double
<<abstract>>
Command
- cc: Account
+ esegui():void
+ annulla(): void
Chi è l'Invoker?
Chi è il Receiver?
VersamentoCmd
- importo: double
+ esegui():void
+ annulla(): void
PrelievoCmd
- importo: double
+ esegui():void
+ annulla(): void
AccountMng
- listaCmd: Command [ ]
+ versa():void
+ preleva(): void
+ undo() : void
+ getSaldo(): double

3) Command

- Il pattern Command su Struts



3) Command

Conseguenze

- Il client è fortemente disaccoppiato dal Receiver che esegue materialmente la sua richiesta
- E' possibile creare comandi composti e che manipolano altri comandi
  - Inoltre è possibile inserire comportamenti particolari attraverso un'ulteriore classe Scheduler (sincronizzazione, priorità, ecc…) → schedula i comandi concreti.

Note

- E' conveniente per realizzare comandi che lavorano su altri comandi, viceversa si ha un overhead di classi inutile!!!

4) Observer

- **Problema**
  - Si vuole gestire un oggetto per cui il cambiamento di stato produce una notifica ad altri oggetti.
  - Il numero dei notificati può cambiare a run-time
- **Soluzione**
  - Creare 2 classi
    - l'**Osservato** → aggancia gli Osservatori e li avvisa
    - l'**Osservatore** → esegue comportamenti in funzione dell'evento generatosi sull'Osservato
  - Gli Osservatori si potranno aggiungere a run-time

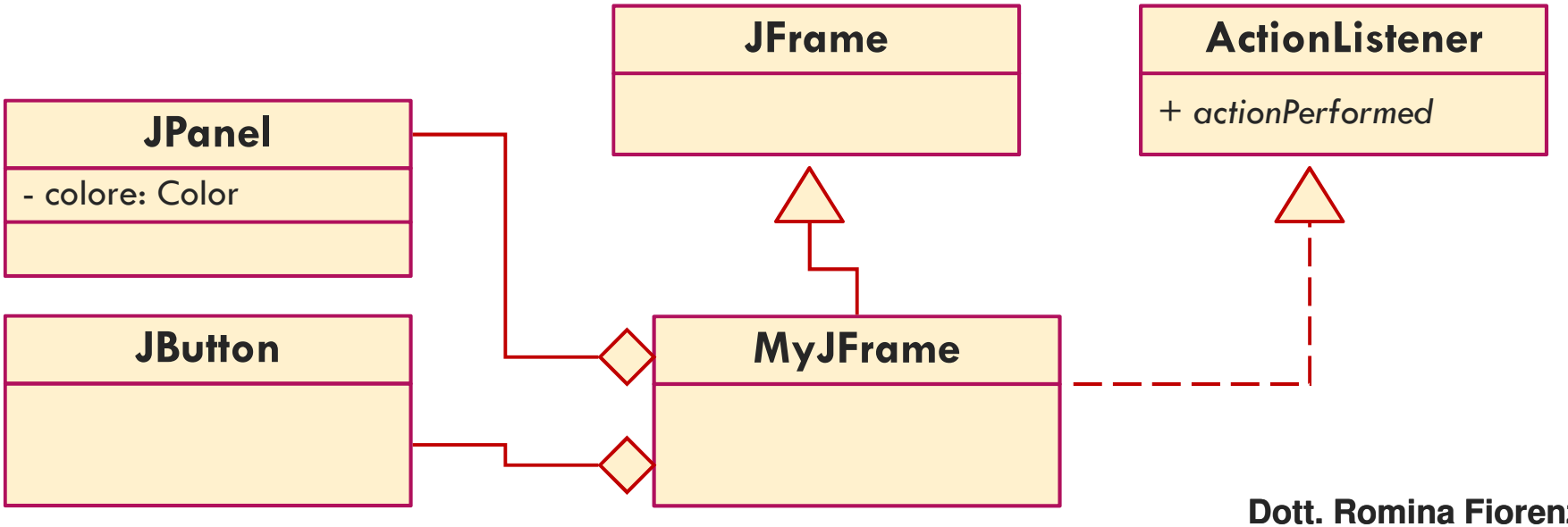
4) Observer

- **Esempio:** la programmazione ad eventi
  - in java si realizza con i package **java.awt** e **java.swing**

- Si vuole realizzare una finestra grafica con un bottone, tale che premendo il bottone cambi il colore di sfondo della finestra.
  - i colori che si alterneranno sono azzurro e verde

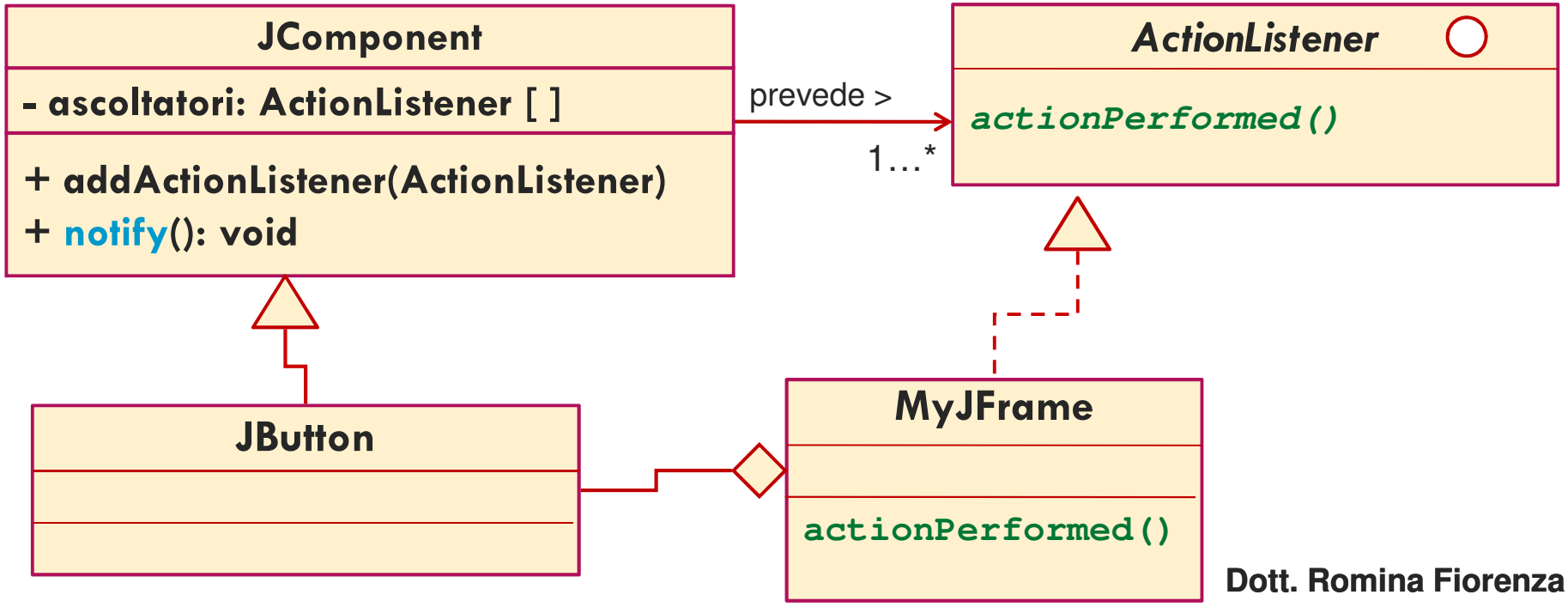
4) Observer

- **Soluzione**
  - Realizzare una classe che estende **JFrame** con 2 proprietà **JButton** e **JPanel**
  - MyJFrame quindi si registrà come **ActionListener** del JButton, cioè un osservatore di eventi sul bottone → potrà agire sul JPanel quando viene premuto il bottone

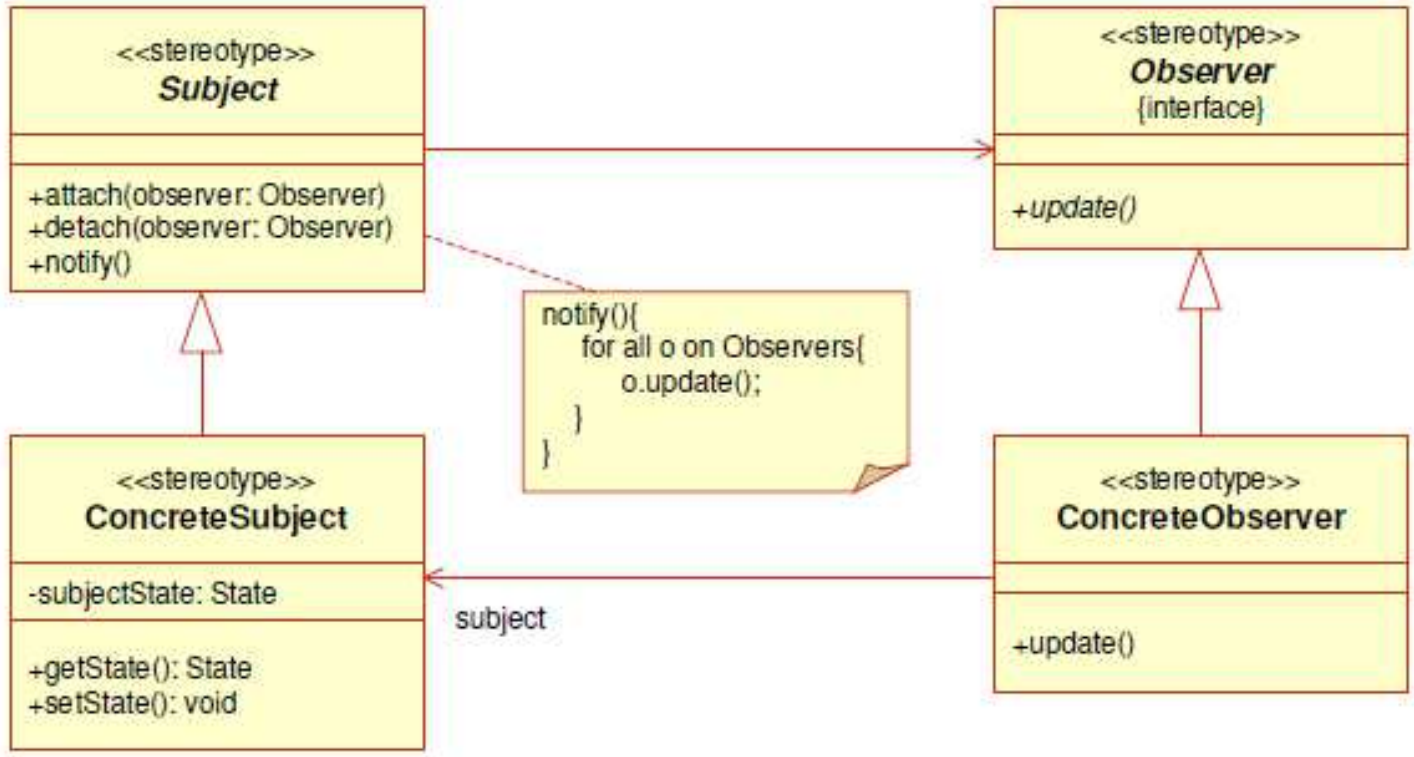


4) Observer

- Tutti i **JComponent** prevedono una lista configurabile di **Listener**
- MyJFrame si registra con `bottone.addActionListener(this)`
- Cliccando sul pulsante, il framework delle swing invoca il metodo `notify` sul JButton che invoca il metodo `actionPerformed` su tutti gli oggetti presenti nel vettore "ascoltatori"

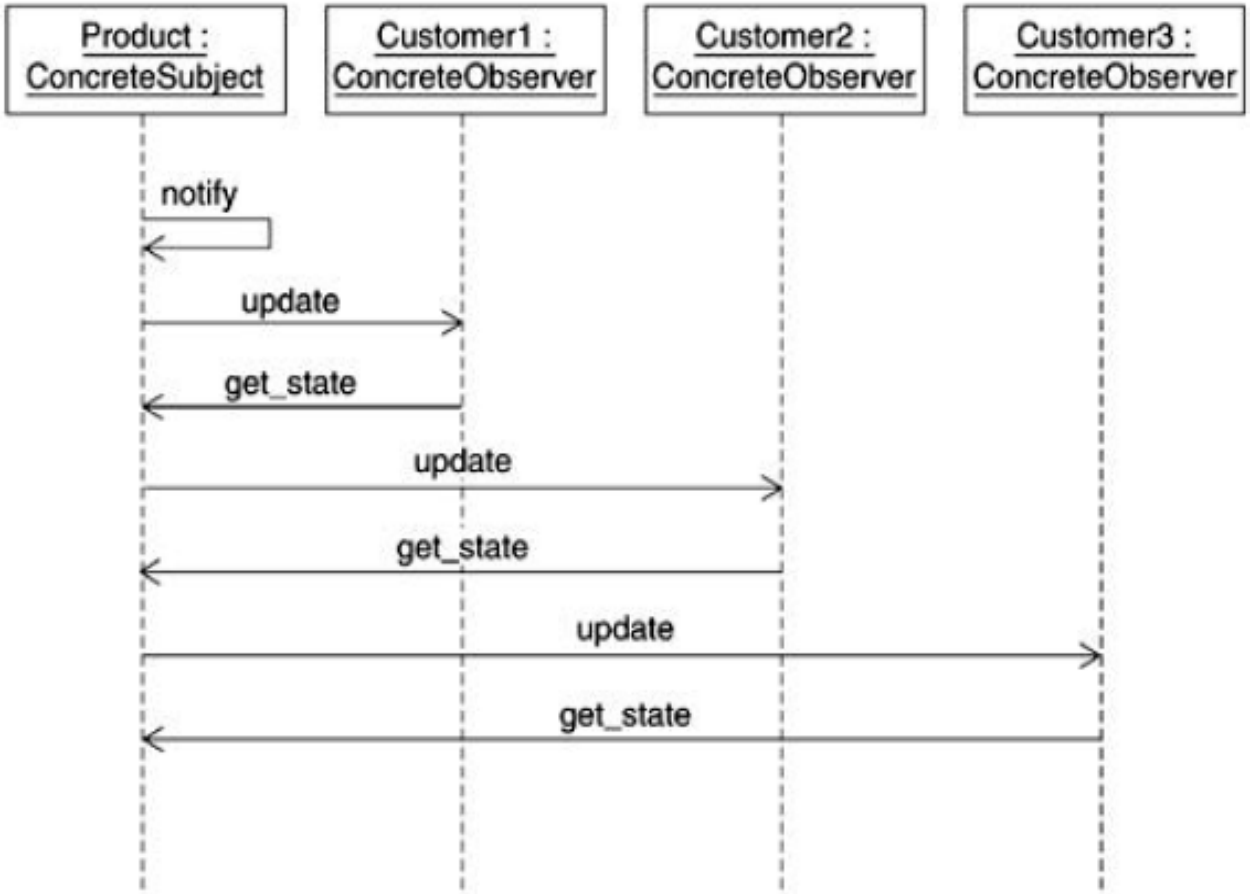


4) Observer Class Diagram



4) Observer Sequence Diagram

- Dettaglio del metodo Notify del Subject concreto



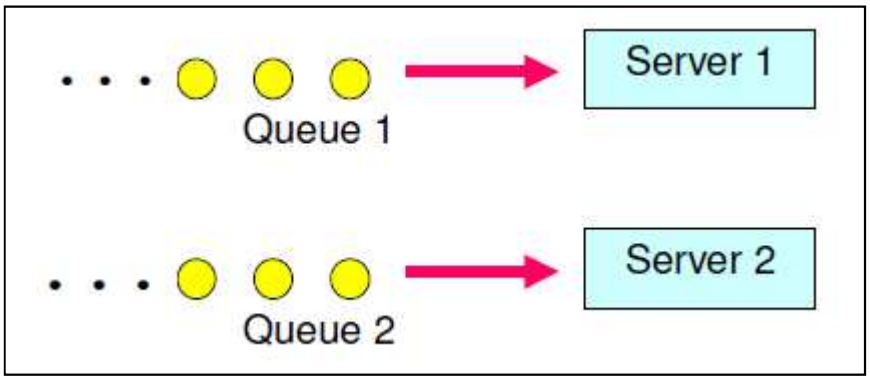
4) Observer

Conseguenze

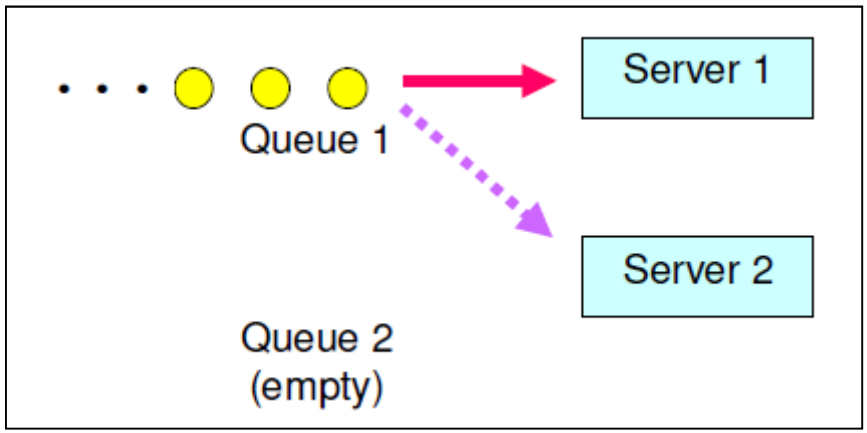
- L'Observer è esterno al Subject e può essere configurato in modo indipendente
- E' possibile realizzare una comunicazione multicast
- Uno stesso Observer può essere registrato su diversi Subject
- L'Observer è "autorizzato" dal Subject ad agire su di lui, quindi può effettuare anche operazioni sul suo stato.

5) Mediator

- **Esempio**
  - Un simulatore di un sistema bancario consente di creare diverse configurazioni di cassieri (server) e clienti in coda (queue)
  - Si vogliono prevedere, per ora, 2 configurazioni:

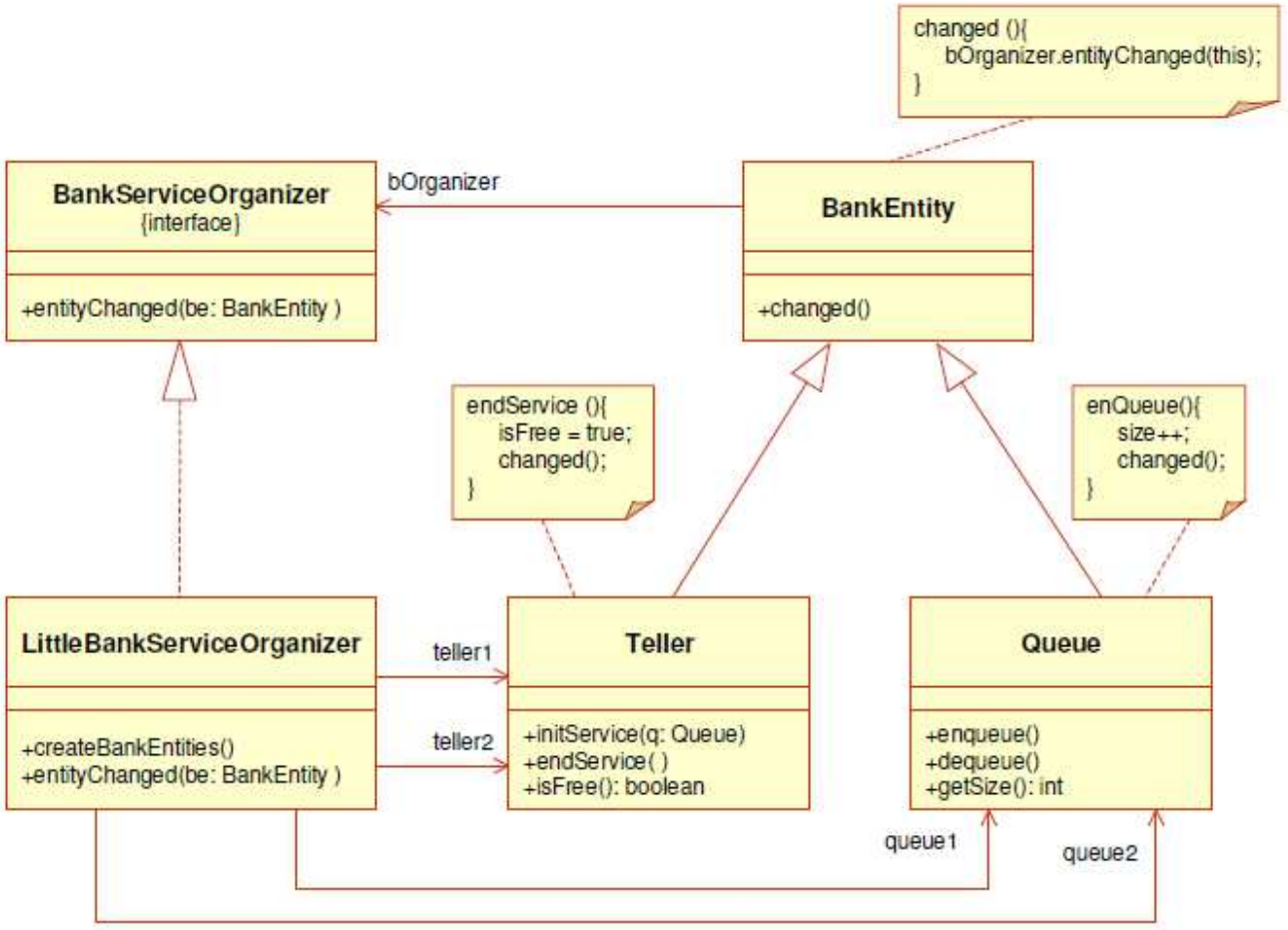


Ogni cassiere serve SOLO la propria coda



Un cassiere potrebbe aiutare a servire altre code, se la propria è esaurita

5) Mediator


changed (){
bOrganizer.entityChanged(this);
}
BankServiceOrganizer
{interface}
+entityChanged(be: BankEntity)
bOrganizer
BankEntity
+changed()
endService (){
isFree = true;
changed();
}
enQueue(){
size++;
changed();
}
LittleBankServiceOrganizer
+createBankEntities()
+entityChanged(be: BankEntity)
teller1
teller2
Teller
+initService(q: Queue)
+endService( )
+isFree(): boolean
Queue
+enqueue()
+dequeue()
+getSize(): int
queue1
queue2

5) Mediator

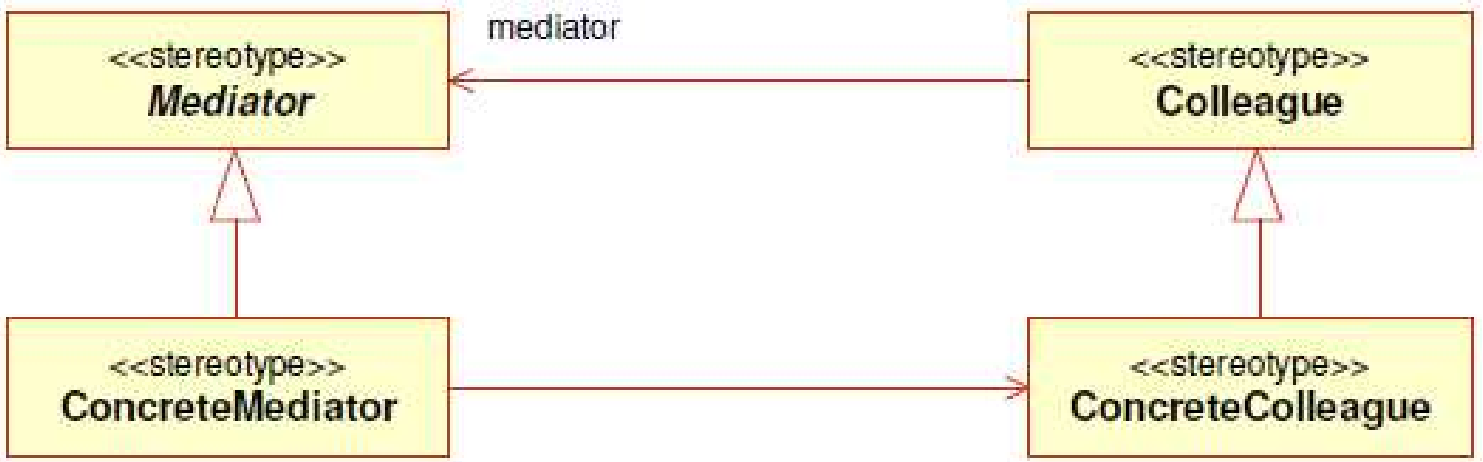
Partecipanti

- Mediator (BankServiceOrganizer)
  - Definisce un'interfaccia di comunicazione tra Colleague
- ConcreteMediator (LittleBankServiceOrganizer)
  - Implementa il funzionamento cooperativo coordinando i Colleague
  - Conosce e gestisce i Colleague
- Colleague classes (BankEntity)
  - ogni Colleague conosce il Mediator
  - ogni Colleague comunica con il Mediator al posto di un Colleague

5) Mediator

- **Problema**
  - Definire un oggetto che incapsuli l'interazione di un insieme di oggetti relativi ad una struttura complessa.
  - Gli oggetti non devono avere riferimento tra loro
- **Soluzione**
  - Creare una classe che concentra il comportamento invece di distribuirlo

5) Mediator Class Diagram



5) Mediator

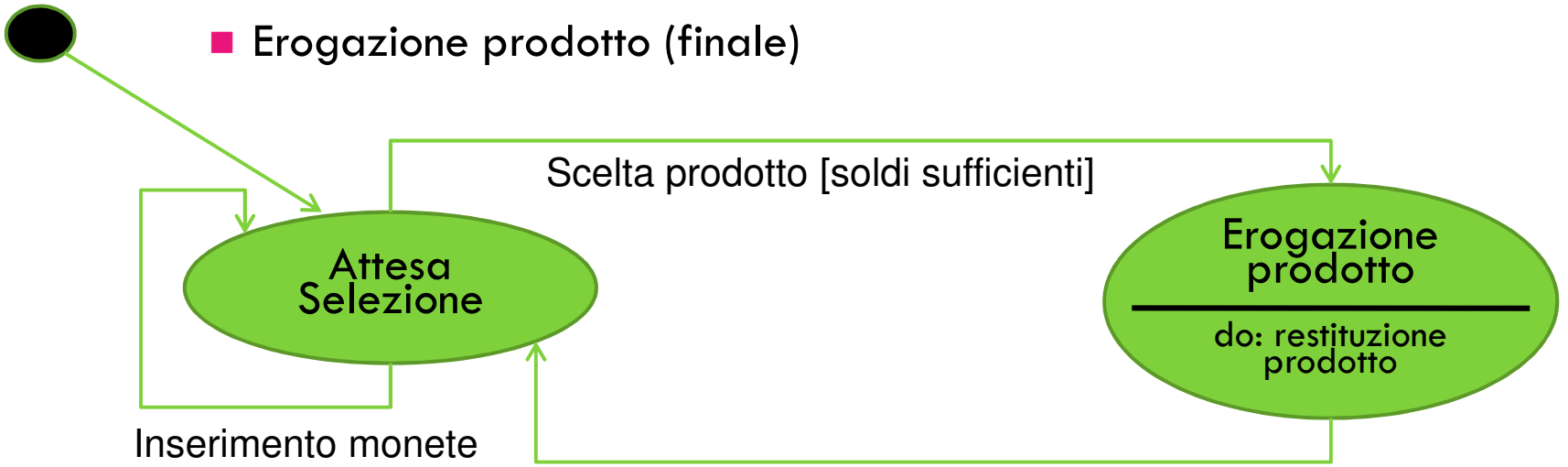
- **Conseguenze**
  - Il comportamento di un partecipante è localizzato
  - I partecipanti sono leggermente accoppiati
  - Posso modificare il protocollo di comunicazione senza renderlo noto ai partecipanti
  - Può essere combinato con il pattern Observer.

6) State

- **Problema:**
  - Si desidera modellare oggetti il cui comportamento dipenda dallo stato interno dell'oggetto stesso.
  - Si vuole garantire indipendenza dei vari stati (aggiungerne altri, eliminarli…)
- **Soluzione:**
  - Disaccoppiare gli stati dell'oggetto dall'oggetto stesso
  - Abbinare un comportamento che dipenda dallo stato interno dell'oggetto stesso

6) State

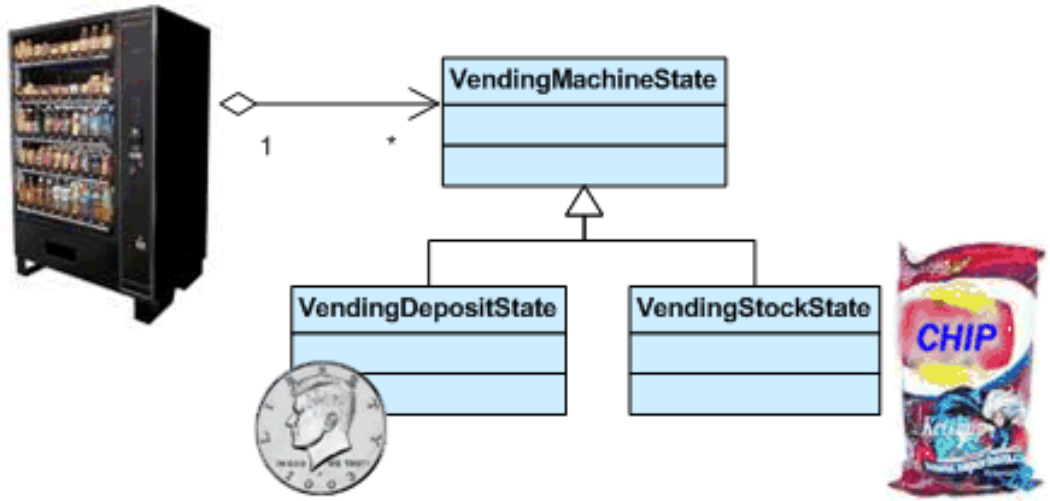
- **Esempio:**
  - Vogliamo realizzare il software relativo ad un distributore di snack, patatine, bibite, ecc.
  - Il distributore è una macchina a stati finiti, che prevede questi 2 stati:
    - Attesa selezione (iniziale)
    - Erogazione prodotto (finale)



6) State

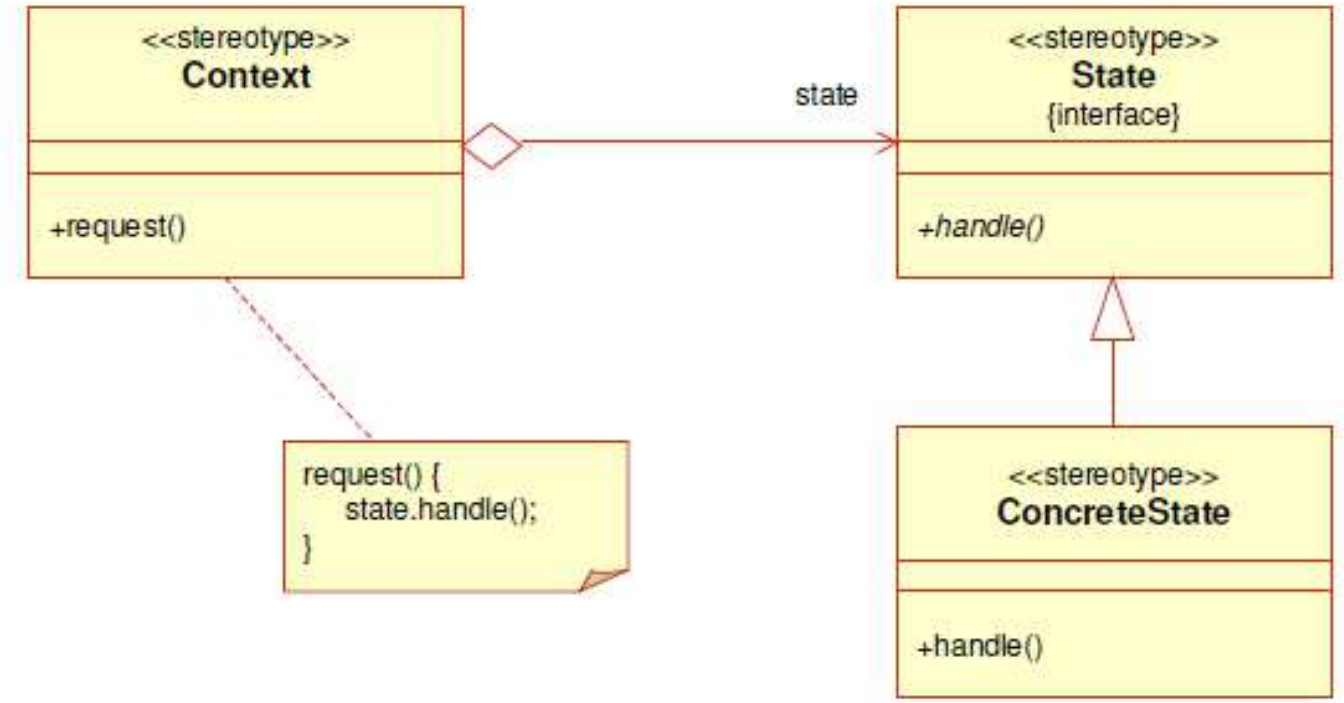
Soluzione Esempio

- La macchina ha una proprietà per lo stato corrente
- Si realizza un albero di generalizzazione degli stati: una classe per stato.
- La super classe astratta State definisce le azioni da compiere.
- Le sottoclassi realizzano le azioni per lo stato che rappresentano e impostano le regole per il passaggio di stato



NB: La macchina ha un riferimento allo stato corrente.
Ogni stato ha un riferimento alla macchina → la super classe State avrà la proprietà relativa

6) State Class Diagram



Il metodo `handle()` del `ConcreteState` dovrà poter agire sul `Context`, quindi l'oggetto deve disporre di un riferimento al `Context`.
I meccanismi di passaggio di stato sono impostati nelle classi `State`, che risulteranno un po' accoppiate tra loro

6) State

- **Conseguenze**
  - E' il miglior modo per rappresentare una macchina a stati finiti, oppure software è caratterizzato da stati e eventi

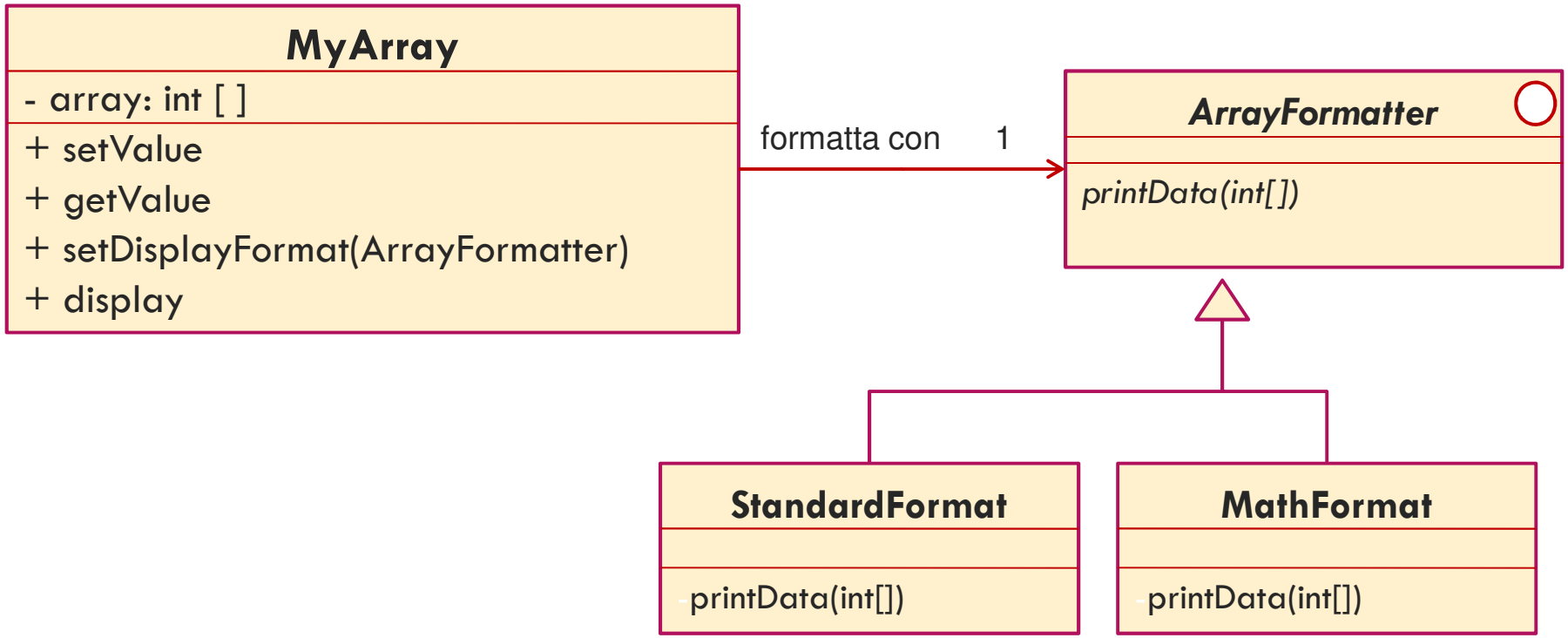
7) Strategy

- **Esempio**
  - Si vuole realizzare una classe MyArray per disporre di tutte le funzioni utili per lavorare con vettori di numeri.
  - Si prevedono 2 funzioni di stampa:
    - Formato matematico → { 67, -9, 0, 4, …}
    - Formato standard →

    Arr[0] = 67 Arr[1] = -9 Arr[2] = 0 Arr[3] = 4 … …

  - Questi formati potrebbero, in futuro, essere sostituiti o incrementati

7) Strategy

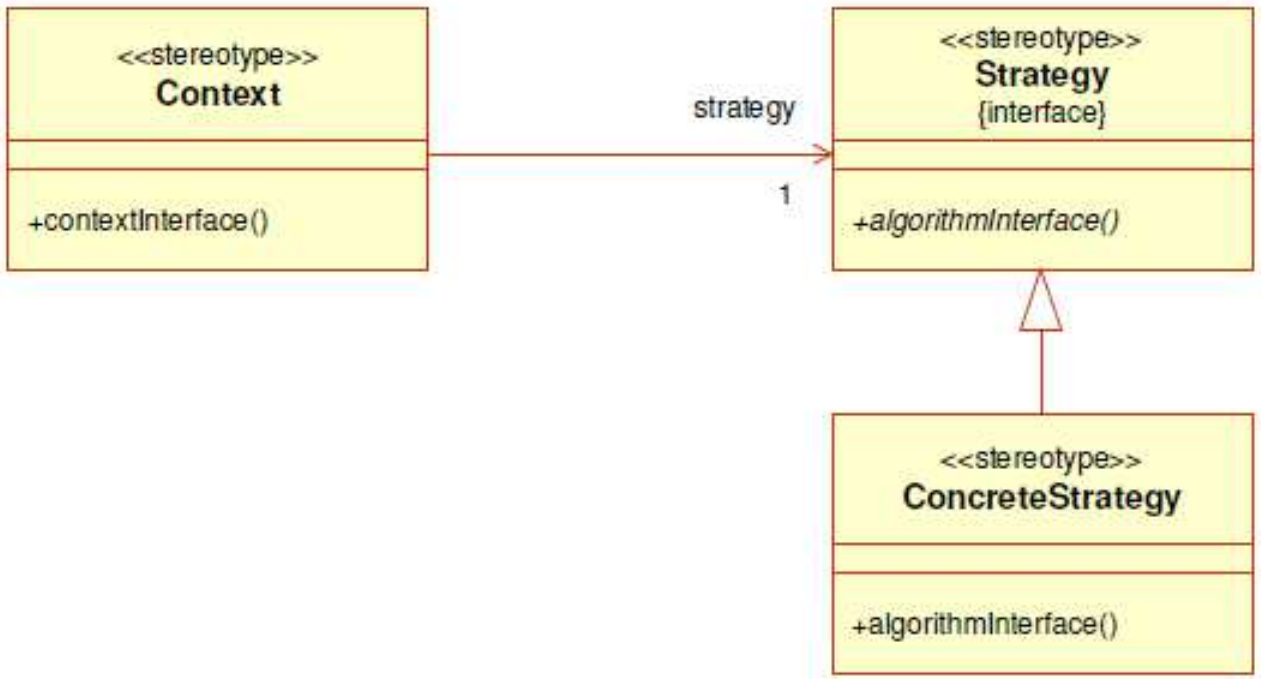


- Si definisce un formattatore astratto e 2 implementazioni concrete
- La classe di contesto imposta il formattatore con **`setDisplayFormat`**
- Il metodo display invoca il **`printData`** sul formattatore scelto

7) Strategy

- **Problema:**
  - Un programma deve fornire più varianti di un algoritmo o comportamento.
  - I diversi comportamenti devono restare indipendenti dalle classi che li espongono
- **Soluzione:**
  - Incapsulare la logica degli algoritmi in apposite classi
  - Prevedere meccanismi dinamici di scelta e configurazione degli algoritmi.

7) Strategy Class Diagram



- Il Context delega un attività di sua competenza ad un classe che incapsula una strategia.
- Il Context sceglie una strategia, ma la invoca senza conoscerne i dettagli implementativi

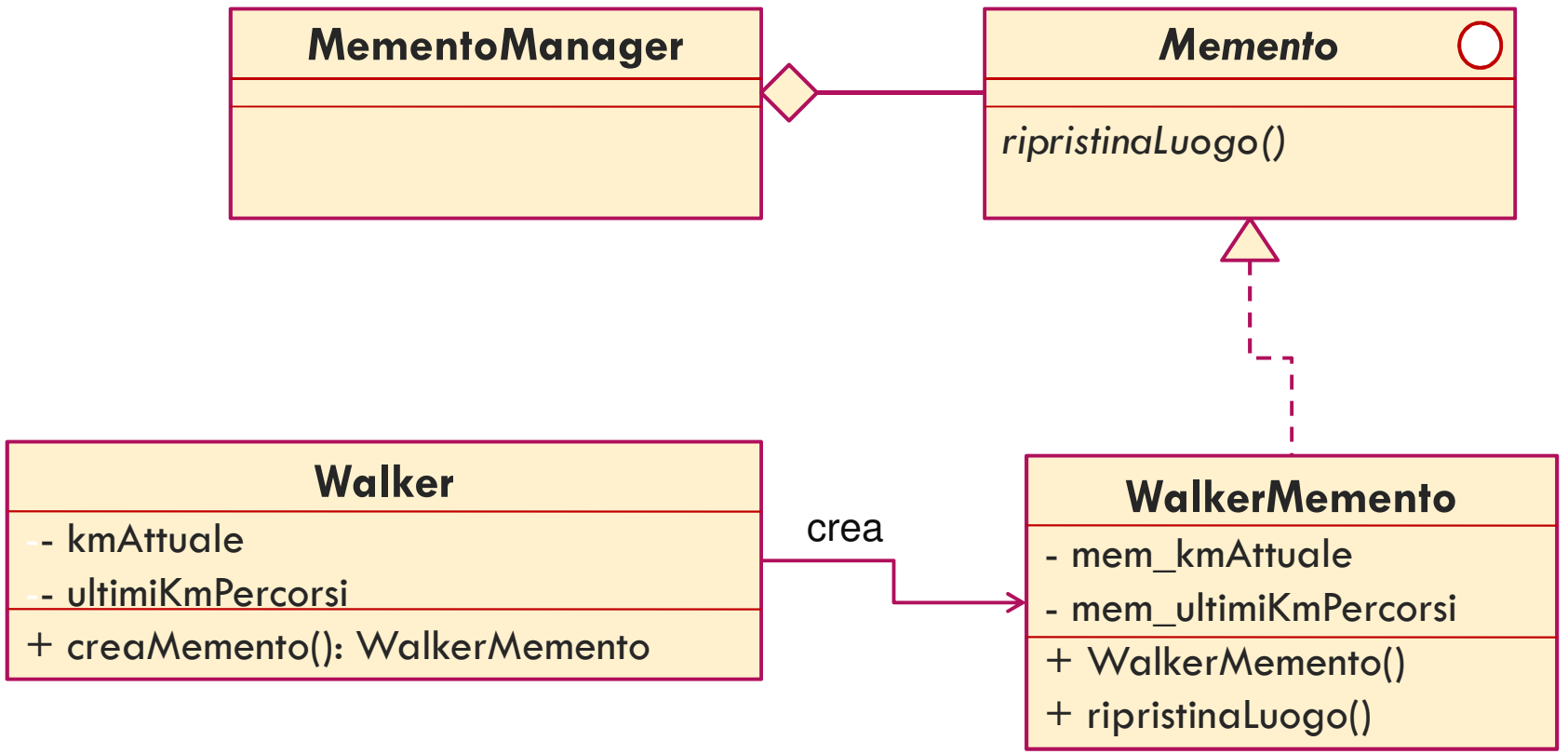
7) Strategy

- **Conseguenze**
  - Si determina dinamicamente il comportamento di oggetti eliminando cascate di if o switch
  - Vantaggi dovuti al polimorfismo, senza uso banale dell'ereditarietà
  - Forti similitudini con pattern Bridge, anche se gli intenti sono diversi:
    - Bridge → variano le implementazioni di un'astrazione → variano le strutture interne di un oggetto
    - Strategy → variano gli algoritmi → variano i comportamenti di un oggetto
  - Nota: si rompe in parte l'incapsulamento di una classe, perché si portano fuori dalla classe le strategie relative ai metodi

8) Memento

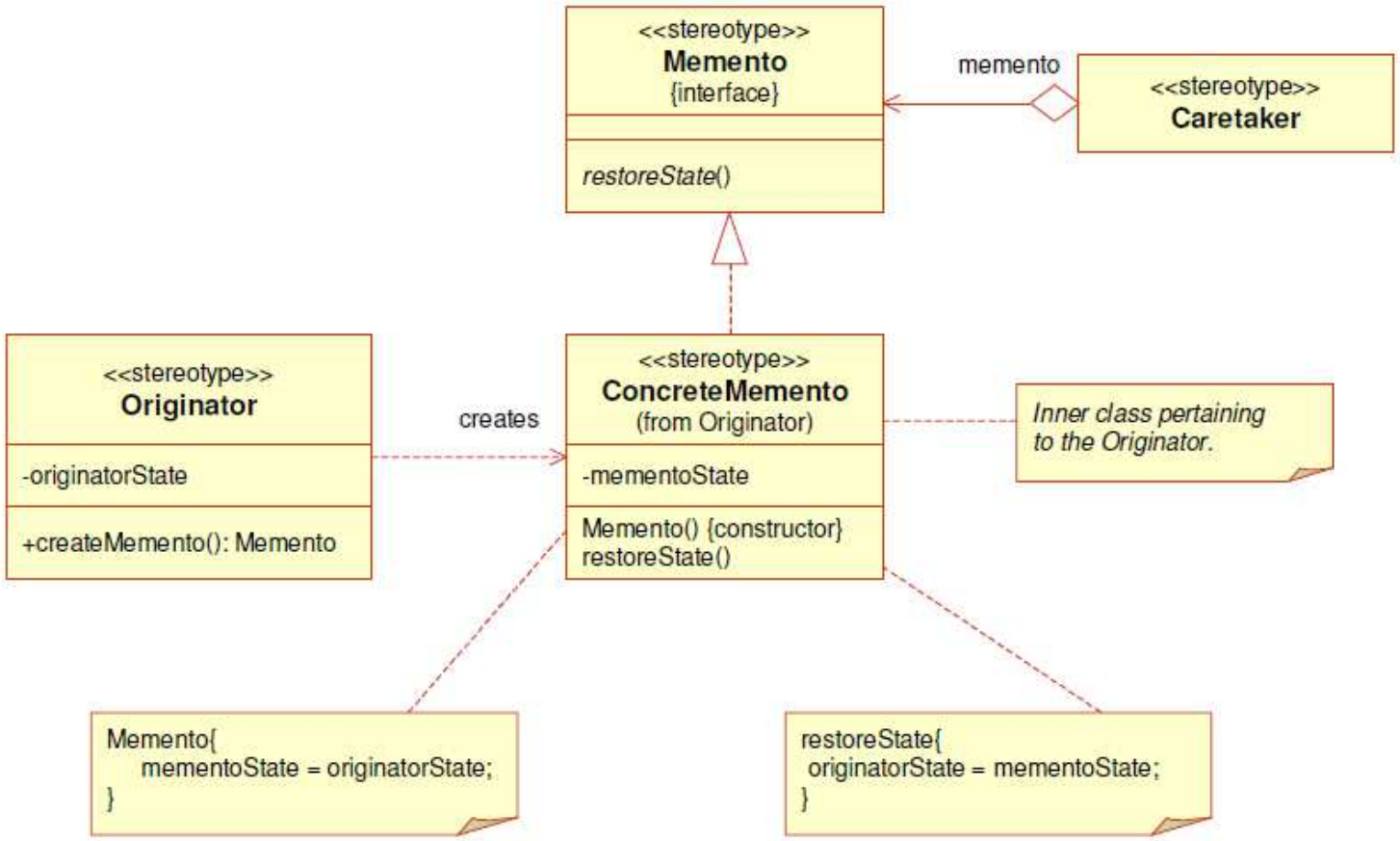
- **Esempio**
  - Un viaggiatore sta facendo un viaggio a tappe
  - Le tappe sono determinate in modo casuale, ma seguono un criterio:
    - Quando il viaggiatore arriva in un luogo, decide se è di suo gradimento oppure no.
    - Nel primo caso, resta in quel luogo e sceglie casualmente il prossimo luogo
    - Nel secondo caso, torna al luogo precedente
  - L'algoritmo non consente di determinare le condizioni di partenza tramite quelle di arrivo.

8) Memento



- Il **Walker** crea un Memento se il luogo gli piace
- Il **Memento** memorizza le info relative all'ultima tappa significativa
- Il **MementoManager** incapsula la logica di creazione e ripristino del Memento (Nell'esempio conserverà un solo memento per volta)

8) Memento Class Diagram



8) Memento

- **Problema**
  - Trovare un meccanismo per esternalizzare lo stato di un oggetto per poter supportare operazioni di ripristino
  - Non si vogliono usare variabili esterne, per non rompere l'incapsulamento dei dati
- **Soluzione**
  - Disaccoppiare lo stato dall'oggetto
  - Solo l'oggetto esterno è visibile dal client
  - Per l'oggetto è previsto un metodo di ripristino gestito dal un entità che cura tale attività

8) Memento

- **Conseguenze**
  - Un gestore di memento (CareTaker) potrebbe gestire uno stack di memento e fornire un meccanismo di ***undo*** relativo a tutti gli stati dell'oggetto iniziale (Originator)

9) Template Method

- **Esempio**
  - Si vuole realizzare un set di funzioni per effettuare operazioni sugli array.
  - Si prevedono 2 funzioni aritmetiche:
    - Somma di tutti gli elementi
    - Prodotto di tutti gli elementi
  - Si desidera realizzare un meccanismo che consenta di
    - evitare algoritmi simili ripetuti
    - aggiungere funzionalità, senza dover ripetere le strutture → violazione principio DRY

9) Template Method

```
public int somma(int[] array){
   int somma = 0;
   for (int i = 0; i < array.length; i++) {
        somma += array[i];
   }
   return somma;
}
```

```
public int prodotto(int[] array){
   int prodotto= 1;
   for (int i = 0; i < array.length; i++) {
        prodotto *= array[i];
   }
   return prodotto;
}
```

ESEMPIO

Queste 2 funzioni hanno una struttura molto simile.

E' utile, per non effettuare ripetizioni, estrapolare la schema comune e fornire poi le parti differenti

Più è alta la complessità, maggiormente ne varrà la pena!!

9) Template Method

SOLUZIONE

```
public abstract class Calcolatore {

public final int calcola(int[] array){
   int value = valoreIniziale();
   for (int i = 0; i < array.length; i++) {
        value = esegui(value, array[i]);
   }
   return value;
}

public abstract int valoreIniziale();
public abstract int esegui(int currentValue, int element);
}
```

schema

parti

Si crea il metodo con lo schema che richiama i metodi “parte” che bisogna “agganciare”.
Magari lo schema viene marcato final, in modo che non possa essere modificato dalle sottoclassi.
I metodi con le parti dell’algoritmo sono astratti e verranno implementati nelle sottoclassi

9) Template Method

```
public class CalcolatoreSomma extends Calcolatore {

   public int esegui(int currentValue, int element) {
        return currentValue + element;
   }

   public int valoreIniziale() {
        return 0;
   }
}
```

SOLUZIONE

- L'operazione da eseguire è la somma
- L'elemento neutro per la somma è 0

9) Template Method

```
public class CalcolatoreProdotto extends Calcolatore  {

   public int esegui(int currentValue, int element) {
        return currentValue * element;
   }

   public int valoreIniziale() {
        return 1;
   }
}
```

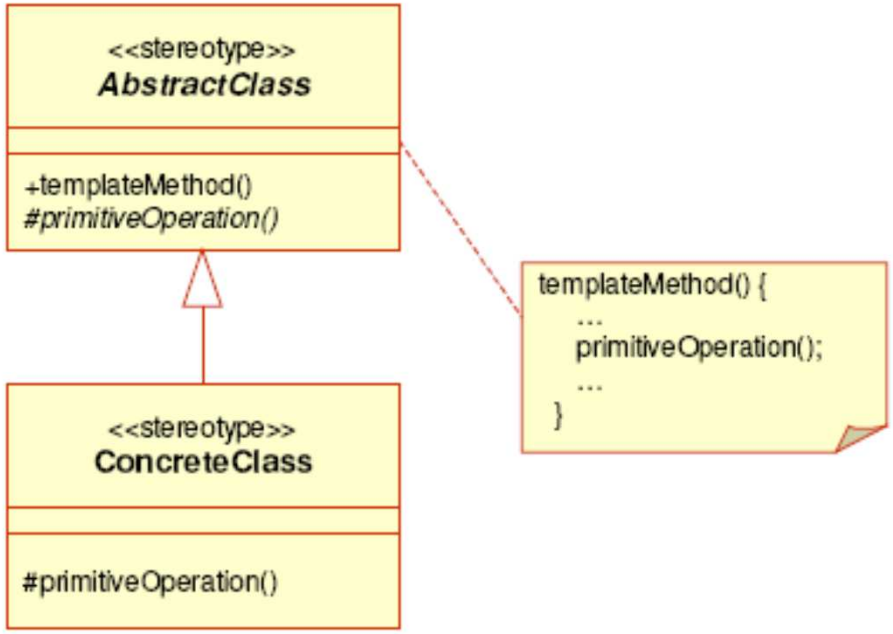
SOLUZIONE

- L'operazione da eseguire è la moltiplicazione
- L'elemento neutro per la moltiplicazione è 1

9) Template Method

- **Problema:**
  - Un programma deve fornire diverse funzioni che hanno una struttura comune, ma differiscono per operazioni di dettaglio
  - Si vogliono evitare inutili ripetizioni
- **Soluzione:**
  - Incapsulare la struttura delle funzioni in una classe astratta che fornisce il template (lo schema) dell'algoritmo
    - un **`templateMethod`** conterrà la struttura e invocherà degli **`hookMethod`** per le parti da personalizzare
    - i metodi da "agganciare" possono essere abstract o avere implementazione di default
  - Realizzare le sottoclassi concrete che realizzano le parti differenti dell'algoritmo → implementando gli **`hookMethod`**

9) Template Method
Class Diagram



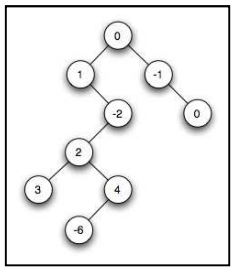
- La classe astratta espone il metodo template che riporta la struttura delle chiamate ai metodi da agganciare (metodi primitivi)
- I metodi primitivi sono realizzati nelle classi concrete

9) Template Method

- **Conseguenze**
  - Si separa la struttura generale comune dai dettagli che differiscono →
    - facilita il riuso della struttura
    - semplifica lo sviluppo dei dettagli
      - perché chi sviluppa i dettagli non deve interessarsi (preoccuparsi) della struttura generale dell'algoritmo
    - garantisce coerenza tra le soluzioni a problemi analoghi
- **Note**
  - Se il numero di **`hookMethod`** è eccessivo → meglio una **Factory**
  - Assomiglia allo **Strategy**, ma le implementazioni di una strategia non usano un template comune, come fanno le sottoclassi della classe Template astratta.

10) Visitor

- **Esempio**
- Si vuole modellare una struttura di albero binario
- Un albero è composto da nodi e foglie
  - Un nodo ha un nome, un nodo destro e un nodo sinistro
  - Una foglia ha solo un valore
    - entrambi sono parti dell'albero



- Si desidera disporre di 2 metodi di stampa:
  - Dalla radice alle foglie
  - Dalle foglie alla radice

  L'idea è che queste operazioni vengano fatte in modo esterno alla struttura e ai suoi oggetti → occorre solamente che l'oggetto preveda di essere acceduto (visitato) per la stampa

10) Visitor

```
public interface Visitor {
    public void visit(Node node);
    public void visit(Leaf leaf);
}
```

```
public abstract class Tree {
     public abstract void accept(Visitor v);
}
```

```
public class Node extends Tree {
… …
    public void accept(Visitor v) {
         v.visit(this);
    }
}
```

```
public class Leaf extends Tree {
… …
    public void accept(Visitor v) {
         v.visit(this);
    }
}
```

SOLUZIONE

Definiamo il visitatore della struttura **Tree.** Tree è implementato da **Node** e **Leaf**

Dunque il Visitor deve avere un metodo per visitare nodi e uno per visitare foglie.

Le 2 classi che compongono l'albero avranno in aggiunta i propri generici metodi di lettura

10) Visitor

```
public class RadiciToFoglieVisitor implements Visitor {
   public void visit(Node node) {
         System.out.println(node.getName());
         node.getLeft().accept(this);
         node.getRight().accept(this);
   }
   public void visit(Leaf leaf) {
        System.out.println(leaf.getValue());
  }}
```

```
public class FoglieToRadiciVisitor implements Visitor {
   public void visit(Node node) {
         node.getLeft().accept(this);
         node.getRight().accept(this);
         System.out.println(node.getName());
   }
   public void visit(Leaf leaf) {
        System.out.println(leaf.getValue());
   }}
```

SOLUZIONE

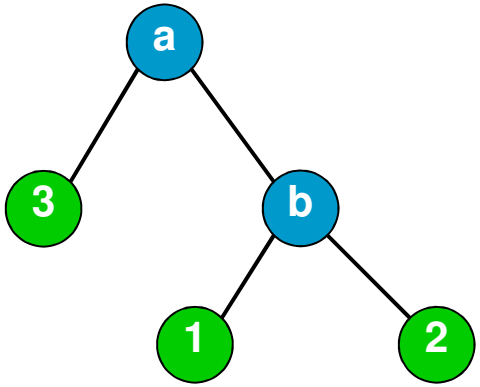
I 2 Visitator concreti realizzano le funzioni di stampa, scorrendo e navigando la struttura esternamente ad essa.

Per fare questo è necessario che i Visitor siano accettati.

Per questo ogni componente del Tree implementa il metodo **accept**()

10) Visitor

- Quindi se l'albero è così composto



- Si avrà che

```
Tree t = new Node("a", new Node("b", new Leaf(1), new Leaf(2)),
                       new Leaf(3));

System.out.println("dalle radici alle foglie");
t.accept(new RadiciToFoglieVisitor());
System.out.println("dalle foglie alle radici");
t.accept(new FoglieToRadiciVisitor());
```

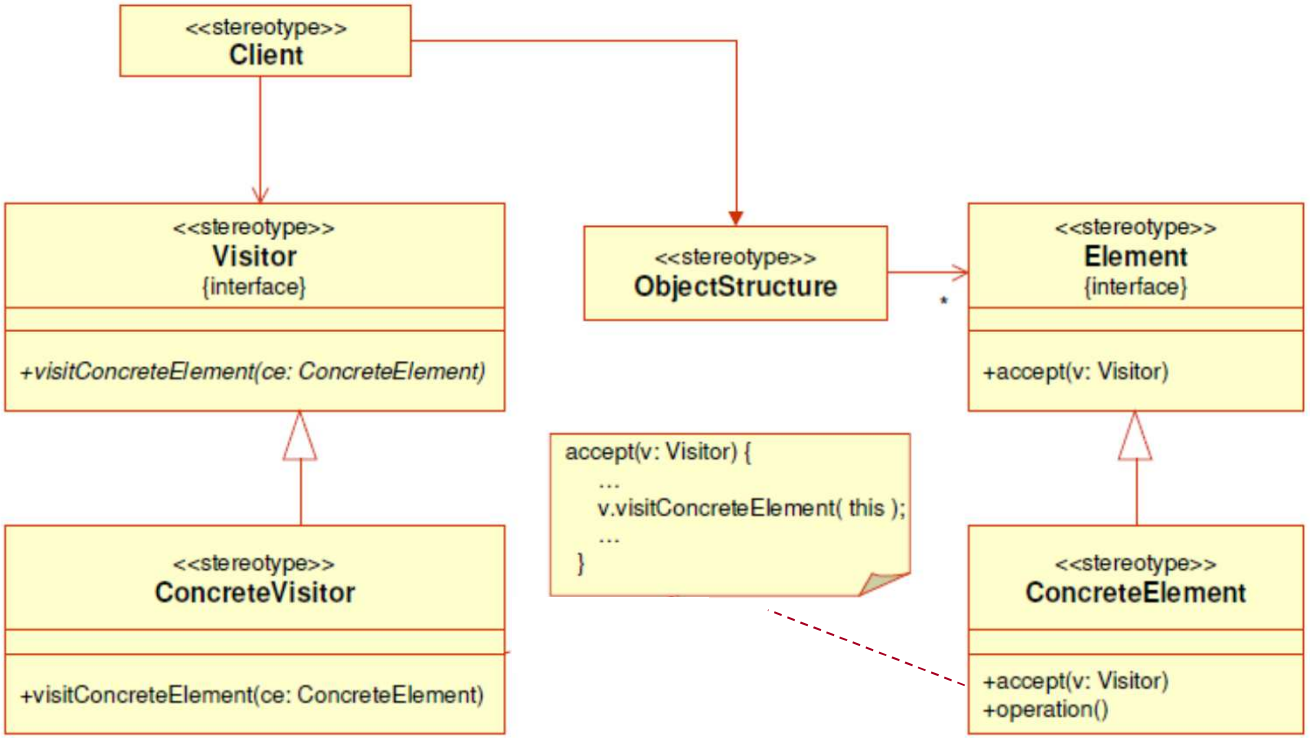
Stampa
a b 1 2 3

Stampa
1 2 b 3 a

10) Visitor

- **Problema:**
  - Dato un insieme di oggetti, si vuole accedere, visitare questi oggetti per effettuare un'operazione esternamente ad essi
  - L'operazione deve essere fatta per tutti gli oggetti, ma magari in modo diverso a seconda degli oggetti
- **Soluzione:**
  - Realizzare un ConcreteVisitor che è in grado di eseguire l'operazione richiesta su un Element dell'insieme.
  - L'Element dovrà dimostrarsi disponibile ad essere "visitato" da un generico Visitor che agirà su di lui

10) Visitor Class Diagram



- Il **ConcreteElement** accetta il Visitor e invoca la visita su se stesso
- Il **ConcreteVisitor** effettua l'operazione in modo esterno all'oggetto
- Il **Client** usa la struttura e creando ConcreteVisitor agisce su di essa senza doverla modificare

10) Visitor

- **Conseguenze**
  - Disaccoppiamento di operazioni su strutture di oggetti dalla struttura stessa
    - Aggiungere nuove operazione è molto agevole, perché non bisogna modificare la struttura
  - Gli Element sono legati al generico Visitor, mentre i ConcreteVisitor sono legati ai ConcreteElement.
- **Note**
  - E' interessante l'implementazione che attraverso la tecnica della Reflection sui metodi, consente di avere un unico metodo di visita nel ConcreteVisitor

11) Interpreter

- **Problema**
  - Definire la rappresentazione della grammatica di un linguaggio e interpretare espressioni booleane costruite con questo linguaggio
- **Soluzione**
  - Realizzare una classe per rappresentare ogni regola grammaticale
    - Se la classe modella un'espressione che non usa altre espressioni → un'espressioni final
    - viceversa se usa altre espressioni → un'espressione non final

11) Interpreter

- **Esempio:**

  vogliamo valutare la seguente espressione:

  (true AND p) OR (q AND NOT p)

  dove p e q sono variabili e "true" è una costante.

- Il risultato di questa espressione si presenta di seguito per ogni **contesto** → possibile combinazione di **p** e **q**

Contesto	p	q	Risultato
1	*true*	*true*	*true*
2	*true*	*false*	*true*
3	*false*	*true*	*true*
4	*false*	*false*	*false*

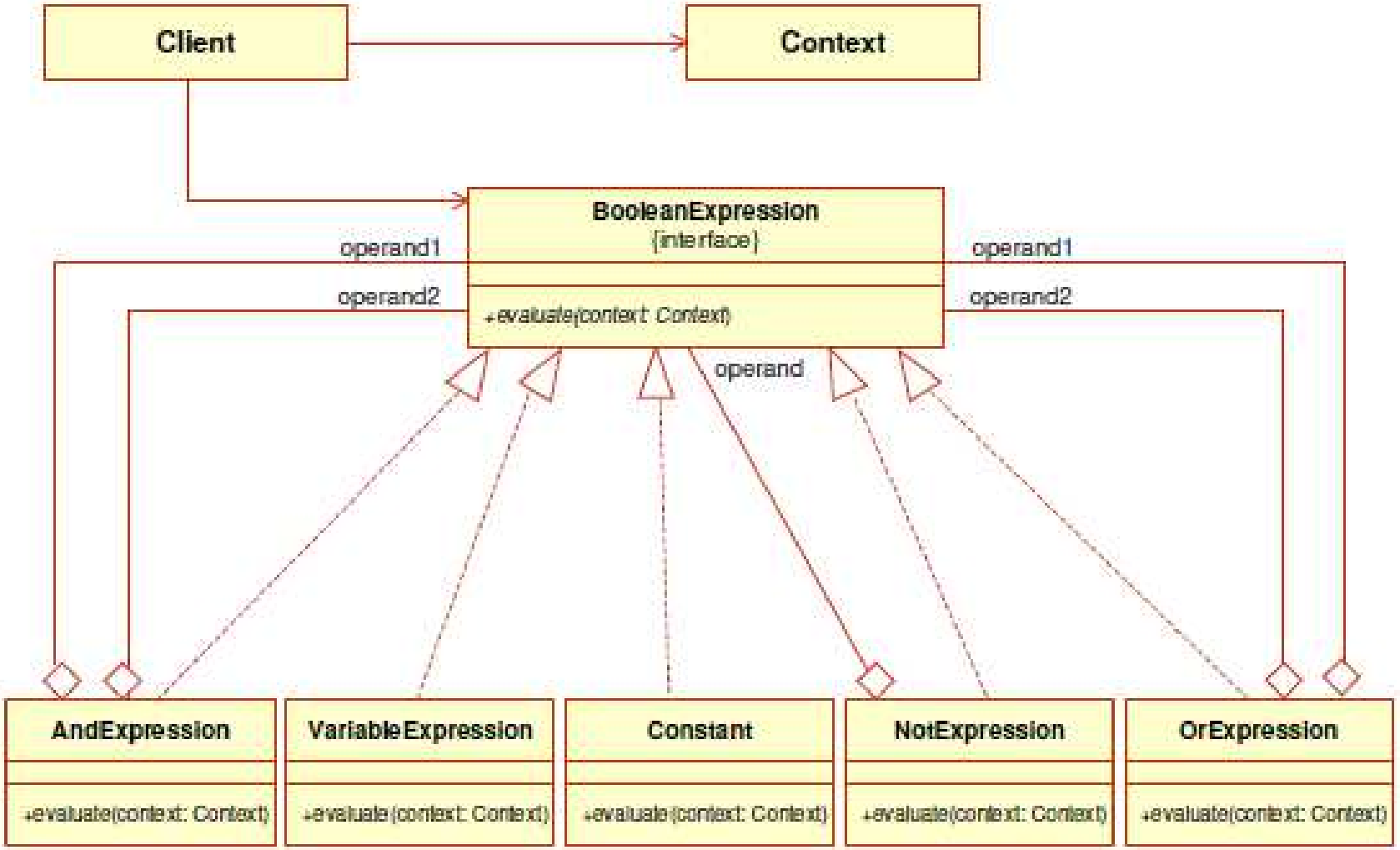
11) Interpreter

- La grammatica definita in questo esempio è:
  - BooleanExpression ::=

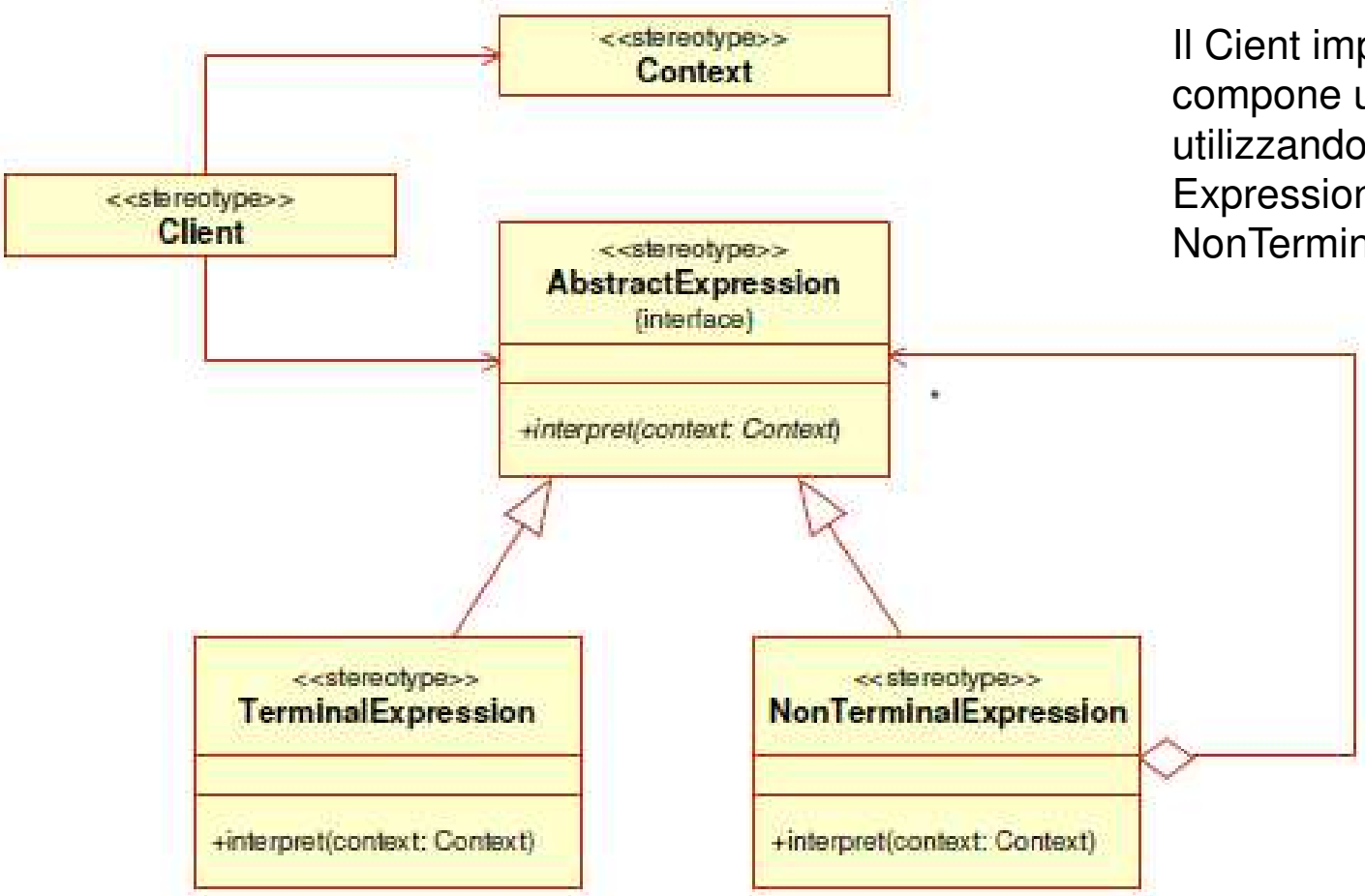
    VariableExpression | Constant | OrExpression | AndExpression | NotExpression | '(' BooleanExpression ')'
  - AndExpression ::= BooleanExpression 'AND' BooleanExpression
  - OrExpression ::= BooleanExpression 'OR' BooleanExpression
  - NotExpression ::= 'NOT' BooleanExpression
  - Constant ::= 'true' | 'false'
  - VariableExpression ::= 'a' | 'b' | … | 'x' | 'y' | 'z'

- I valori delle variabili (true/false) andranno gestiti separatamente in un oggetto Context

11) Interpreter



11) Interpreter Class Diagram



Il Cient imposta un Context e compone un'espressione, utilizzando oggetti Expression (Terminal e NonTerminal)

La valutazione dell'espressione si realizza in modo ricorsivo, poiché ogni **espressione** prevede il suo **meccanismo di valutazione**, in funzione del **contesto**

11) Interpreter

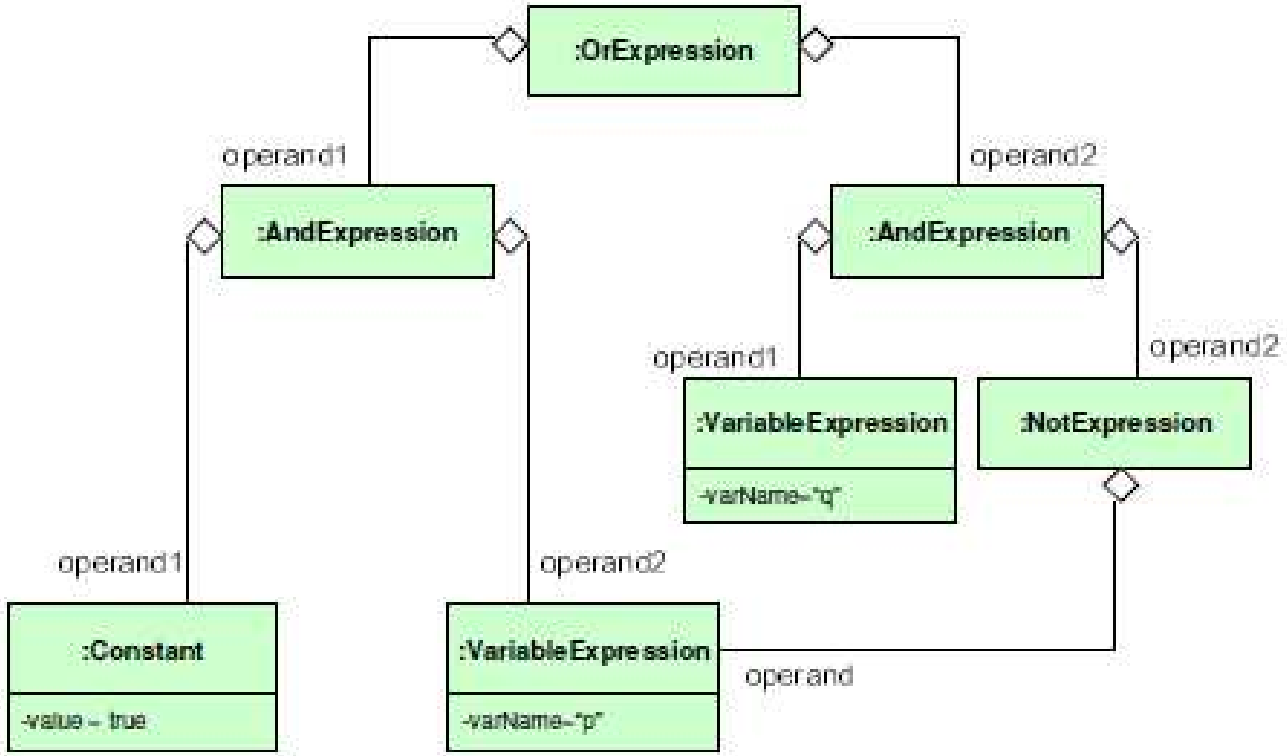
Conseguenze

- Rappresentazione delle espressioni come oggetti aggregati
- Separazione tra gli elementi del linguaggio e i valori
  - I valori sono abbinati ad un Context, che viene usato dinamicamente per valutare tutte le espressioni

11) Interpreter

Diagramma degli oggetti dell'espressione:

(true AND p) OR (q AND NOT p)



11) Interpreter: il codice (1/4)

- Creiamo il **Context** che memorizza in una **HashMap** le variabili del contesto, in forma di coppie **String/Boolean**.
- I metodi **set/get** impostano e leggono i valori delle variabili.

```
public class Context {
   HashMap<String,Boolean> mappa;
   public Context() {
        this.mappa = new HashMap<String, Boolean>();
   }
   // carica valori nel contesto
   public void set(VariableExpression var, boolean value){
        this.mappa.put(var.getName(), value);
   }
   // recupera valori dal contesto
   public boolean get(String name) {
        return this.mappa.get(name);
   }
}
```

11) Interpreter: il codice (2/4)

- Una classe di tipo "**TerminalExpression**"

```
public class VariableExpression implements BooleanExpression {
   private String name;
   public VariableExpression(String name) {
        this.name = name;
   }
   public String getName() {
        return name;
   }
   @Override
   public boolean evaluate(Context context) {
        return context.get(name);
   }
}
```

11) Interpreter: il codice (3/4)

- Una classe di tipo "**NonTerminalExpression**"

```
public class AndExpression implements BooleanExpression {
   private BooleanExpression expr1;
   private BooleanExpression expr2;
   public AndExpression(BooleanExpression express1,
                        BooleanExpression express2) {
        this.expr1 = express1;
        this.expr2 = express2;
   }
   @Override
   public boolean evaluate(Context context) {
        return expr1.evaluate(context) && expr2.evaluate(context);
   }
}
```

NOTA: Le classi non memorizzano i valori che sono conservati nella classe **Context**

11) Interpreter: il codice (4/4)

- Infine creiamo un'espressione **expr**, componendo oggetti **Booleanspression**, e un contesto **contesto** per impostare le variabili.
- Quindi lanciamo il metodo **evaluate**

```
public static void main(String[] args) {
    VariableExpression var1 = new VariableExpression("p");
    VariableExpression var2 = new VariableExpression("q");

    BooleanExpression expr = new OrExpression(
        new AndExpression(new ConstantExpression(true),var1),
        new AndExpression(var2, new NotExpression(var1)));

    Context contesto = new Context();
    contesto.assign(var1, true);
    contesto.assign(var2, true);

    System.out.println("variabili: TRUE - TRUE "+
                                expr.evaluate(contesto));
}
```