UNIVERSITÀ PARTHENOPE
DiST
Dipartimento di Scienze e Tecnologie

Programmazione 3
e
Laboratorio di Programmazione 3

Behavioral Patterns

Angelo Ciaramella

Behavioral Patterns

- Pattern comportamentali
  - forniscono soluzione alle più comuni tipologie di interazione tra gli oggetti

- Design Pattern
  - Chain of Responsibility
  - Command
  - Interpreter
  - Iterator
  - Mediator
  - Memento
  - Observer
  - State
  - Strategy
  - Template Method
  - Visitor

Chain of Responsability

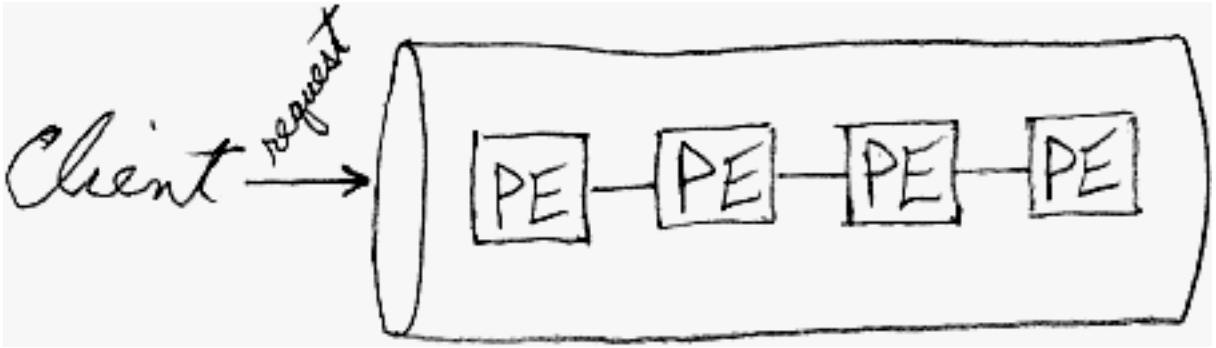
- Scopo
  - *Consente di separare il mittente di una richiesta dal destinatario, in modo da consentire al più ad un oggetto di gestire la richiesta. Gli oggetti destinatari vengono messi in "catena" e la richiesta viene trasmessa fino a trovare un oggetto che la gestisca.*

- Motivazione
  - CoR permette ad un oggetto di inviare un comando senza conoscere quale oggetto la riceverà e la gestirà
  - la richiesta è inviata da un oggetto all'altro nella catena

Chain of Responsability

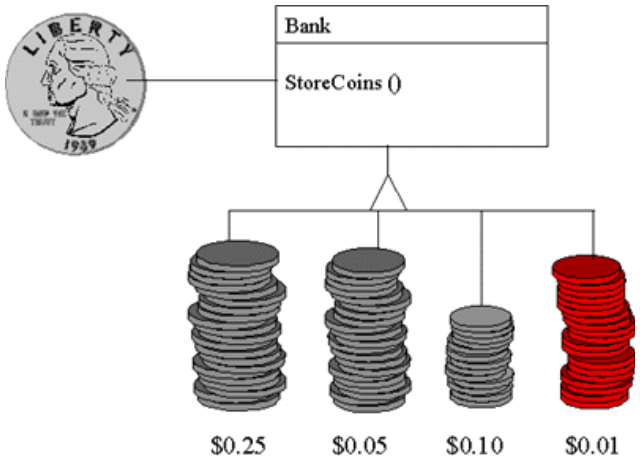
- Applicabilità
  - Il pattern CoR è usato quando
    - non conosciamo a priori quale oggetto è in grado di gestire una determinata richiesta
    - l'oggetto può essere sconosciuto staticamente
    - l'insieme degli oggetti in grado di gestire richieste cambia dinamicamente a runtime

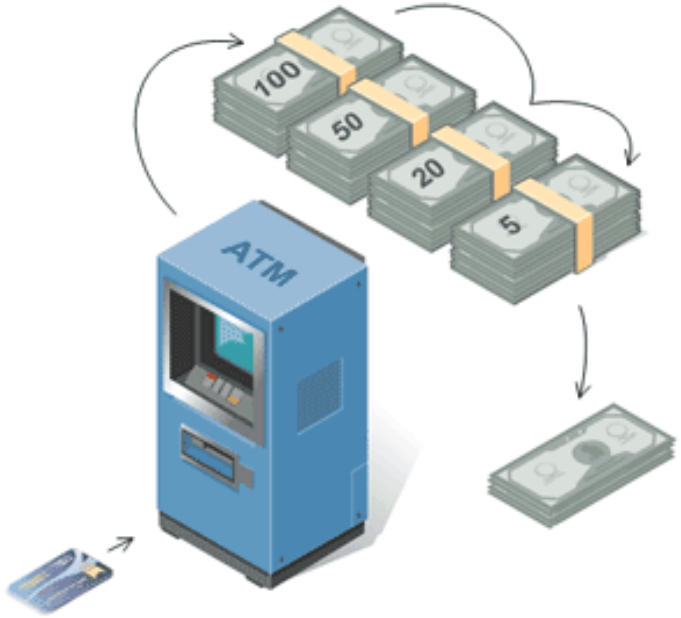
Chain of Responsability



Esemplificazione del processo di richiesta

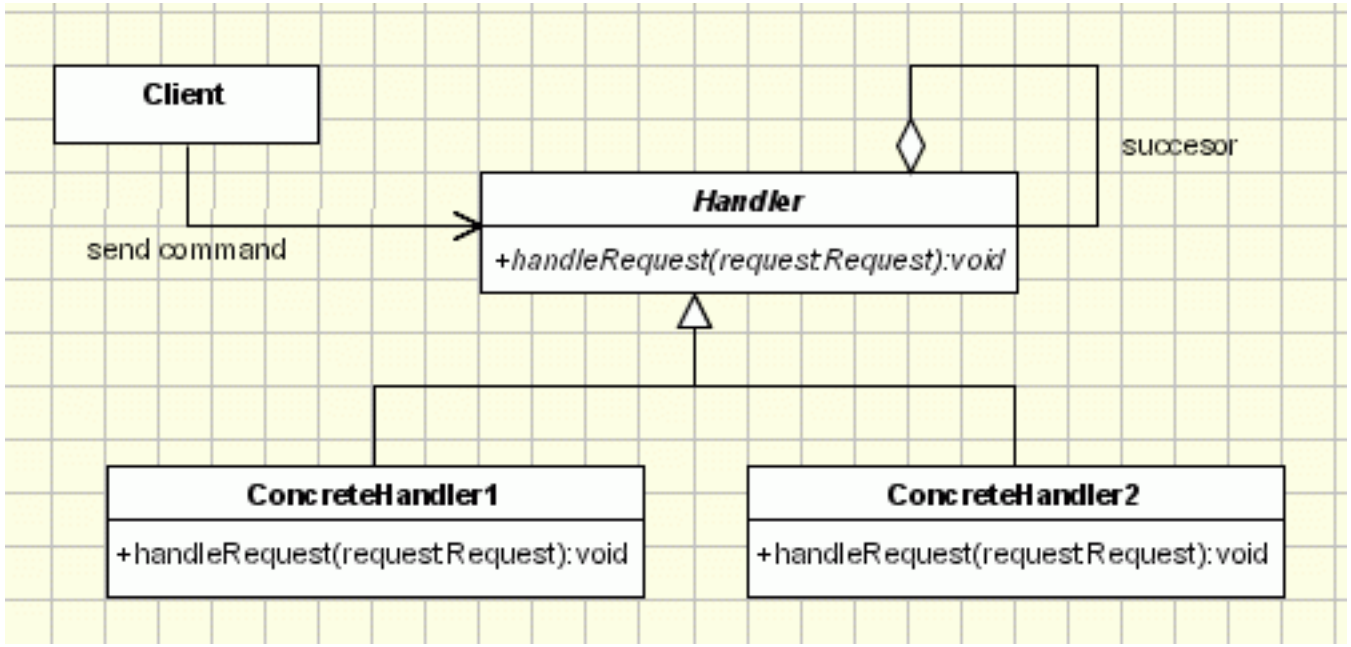
Chain of Responsability





Esemplificazione del meccansismo di CoR

Chain of Responsability



Struttura del pattern CoR

Chain of Responsability

```
public class Request {
	private int m_value;
	private String m_description;

	public Request(String description, int value)
	{
		m_description = description;
		m_value = value;
	}

	public int getValue()
	{
		return m_value;
	}

	public String getDescription()
	{
		return m_description;
	}
}
```


Esempio di implementazione del pattern CoR

Chain of Responsability

```
public abstract class Handler
{
        protected Handler m_successor;
        public void setSuccessor(Handler successor)
        {
                m_successor = successor;
        }

public abstract void handleRequest(Request request);
}
```

Esempio di implementazione del pattern CoR

Chain of Responsability

```
public class ConcreteHandlerOne extends Handler
{
      public void handleRequest(Request request)
        {
                if (request.getValue() < 0)
                {    //if request is eligible handle it
System.out.println("Negative values are handled by
ConcreteHandlerOne:");

System.out.println("\t ConcreteHandlerOne.HandleRequest :
" + request.getDescription() + request.getValue());
                }
                else
                {
                        m_successor.handleRequest(request);
                }
        }
 }
```

Esempio di implementazione del pattern CoR

Chain of Responsability

```
public class ConcreteHandlerThree extends Handler
{
      public void handleRequest(Request request)
        {
                if (request.getValue() == 0)
                {    //if request is eligible handle it
System.out.println(”Zero values are handled by
ConcreteHandlerThree:");

System.out.println("\t ConcreteHandlerThree.HandleRequest
: " + request.getDescription() + request.getValue());
                }
                else
                {
                        m_successor.handleRequest(request);
                }
        }
 }
```

Esempio di implementazione del pattern CoR

Chain of Responsability

```
public class ConcreteHandlerTwo extends Handler
{
      public void handleRequest(Request request)
        {
                if (request.getValue() > 0)
                {    //if request is eligible handle it
System.out.println(”Positive values are handled by
ConcreteHandlerTwo:");

System.out.println("\t ConcreteHandlerTwo.HandleRequest :
" + request.getDescription() + request.getValue());
                }
                else
                {
                        m_successor.handleRequest(request);
                }
        }
 }
```

Esempio di implementazione del pattern CoR

Chain of Responsability

```
public class Main
{
        public static void main(String[] args)
        {
        // Setup Chain of Responsibility
        Handler h1 = new ConcreteHandlerOne();
        Handler h2 = new ConcreteHandlerTwo();
        Handler h3 = new ConcreteHandlerThree();
        h1.setSuccessor(h2);
        h2.setSuccessor(h3);

        // Send requests to the chain
        h1.handleRequest(new Request("Negative Value ", -1));
        h1.handleRequest(new Request("Zero Value ",  0));
        h1.handleRequest(new Request("Positive Value ",  1));
        h1.handleRequest(new Request("Positive Value ",  2));
        h1.handleRequest(new Request("Negative Value ", -5));

        }
}
```

Esempio di implementazione del pattern CoR

Esercizi

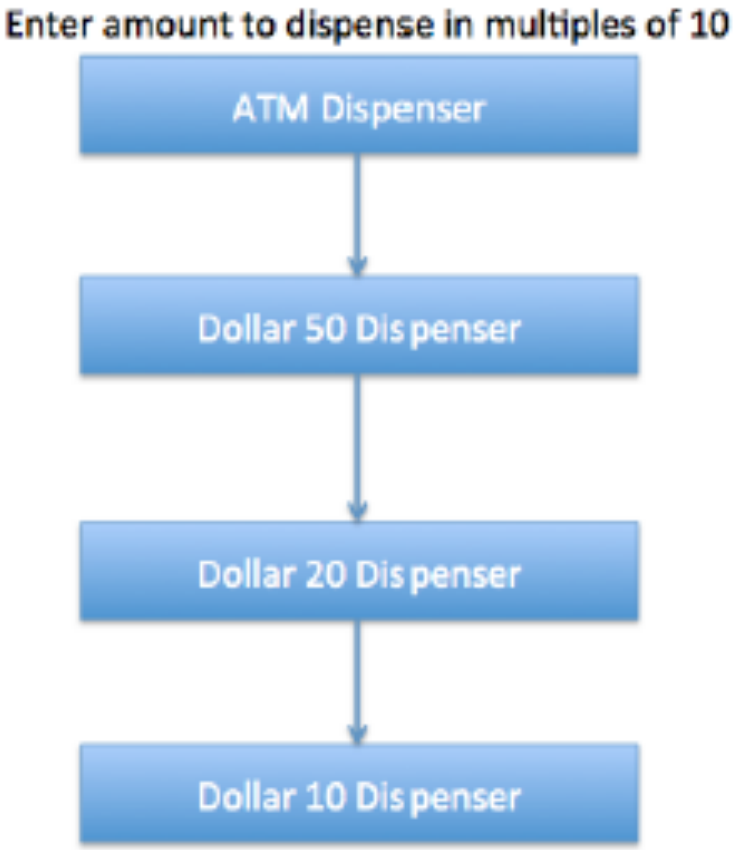
- *Sistema di approvazione di richieste di acquisto*
  - invio richiesta ad un'autorità di omologazione acquisti
  - a seconda del valore, tale autorità può approvare la richiesta o trasmettere alla prossima autorità della catena

- *GUI*
  - propagazione eventi GUI tra catene di oggetti
  - quando un evento viene generato (click del mouse) deve essere trasmesso all'oggetto che lo ha generato

Esercizi

- *Sistema per gli ordini elettronici*
  - Una azienda commerciale deve gestire le richieste di credito dei clienti (*customers*)
  - Internamente l'azienda si organizza in diversi *livelli di responsabilità*
    - I livello (*vendor*)
      - viene consentita l'approvazione di richieste fino a un importo determinato
    - livello superiore (*sales manager*)
      - altro importo massimo da gestire
    - alto livello (*client account manager*)

Esercizi



Schema dell'ATM dispencer

Command

- Scopo
  - *Incapsula una richiesta all'interno di un oggetto, consentendo così di parametrizzare i client con richieste diverse, accodare o rintracciare le richieste, oppure supportare operazioni di undo*
- *Anche conosciuto come*
  - *Action, Transaction*
- Motivazione
  - Spesso si rende necessario richiedere dei compiti ad oggetti senza conoscere tutto sulle operazioni richieste o il ricevente delle richieste
    - e.g., tool per l'interfaccia utente. Non è possibile implementare le azioni specifiche (Button, Menu, …)

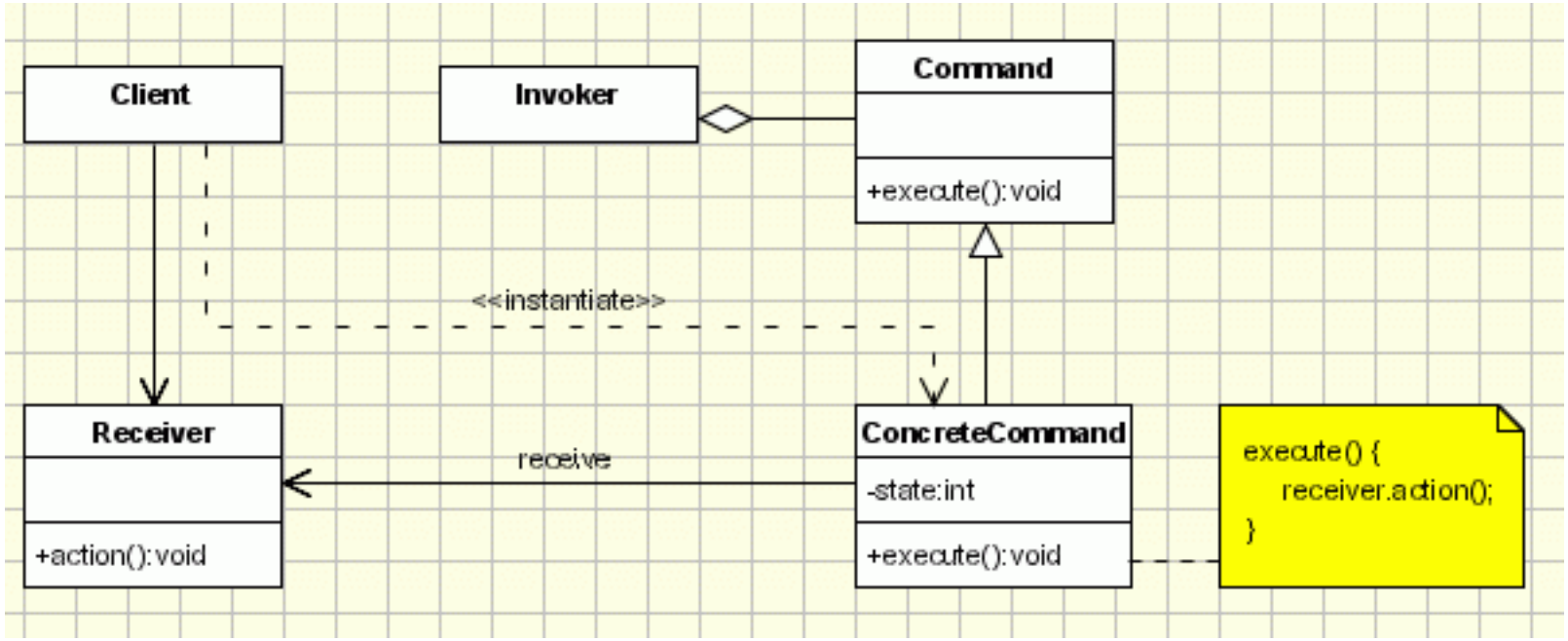
Command

- Applicabilità
  - parametrizzare oggetti a seconda dell'azione che devono svolgere
  - specificare o aggiungere informazioni in una coda ed eseguere le richieste in diversi momenti nel tempo
  - sostegno ad azioni annullabili
    - in grado di memorizzare lo stato e consentire di tornare a quello stato
  - strutturare il sistema in operazioni di alto livello che, sulla base operazioni primitive
  - disaccoppia l'oggetto che richiama l'azione dall'oggetto che esegue l'azione.
    - è conosciuto anche come produttore - consumatore *design pattern*

Command pattern

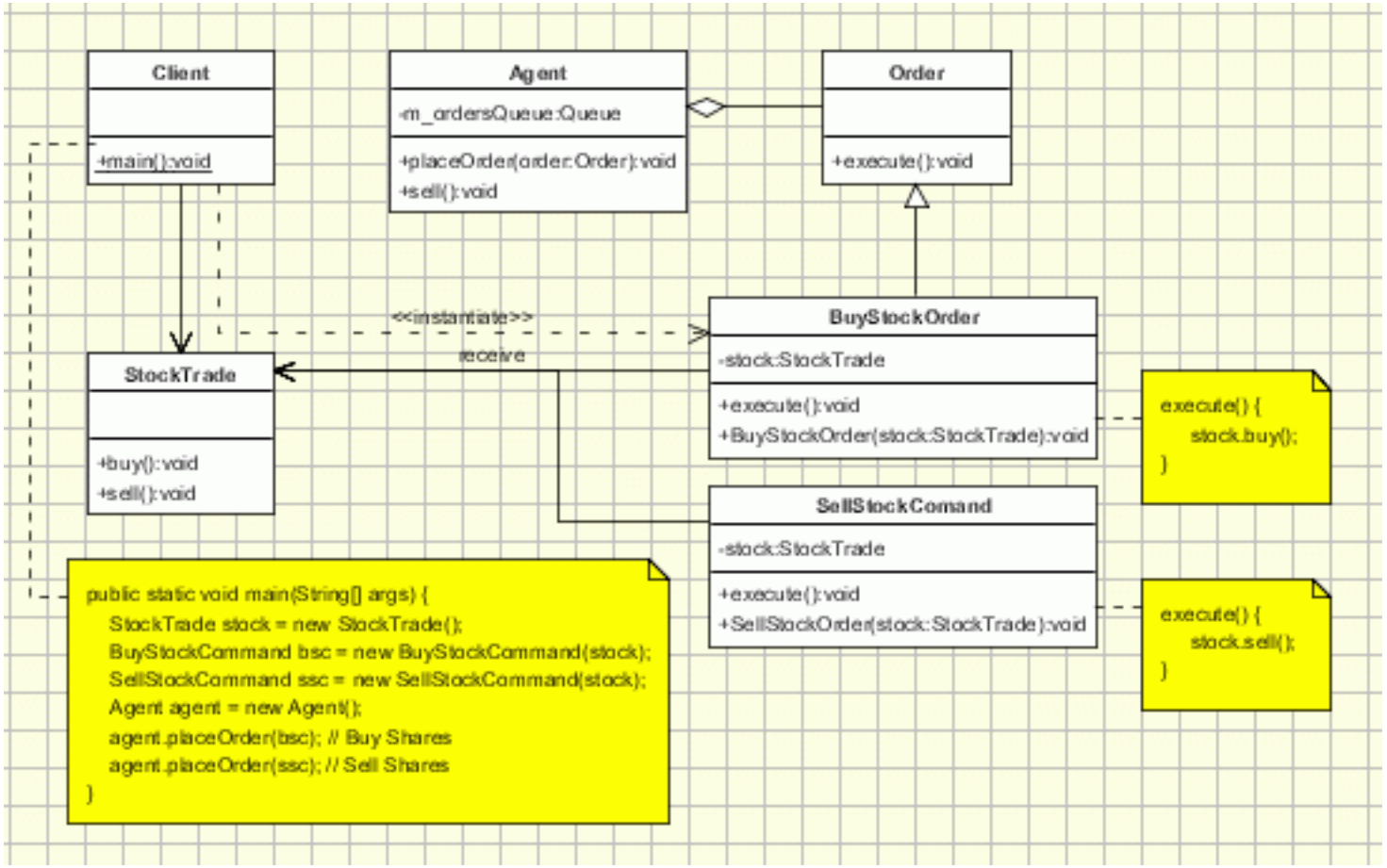
Esemplificazione. Una sequenza di oggetti `Command` possono essere assemblati in comandi compositi

Command pattern



Struttura del pattern Command

Esempio Command



Esempio del pattern Command. Immissione ordini per l'acquisto e vendita di azioni.

Command pattern

```
public interface Order {
    public void execute ();
}

// Receiver class.
class StockTrade {
    public void buy() {
        System.out.println("You want to buy stocks");
    }
    public void sell() {
        System.out.println("You want to sell stocks ");
    }
}
```

Esempio di implementazione del pattern Command

Command pattern

```
// Invoker.
class Agent {
    // definire una coda

    public Agent() {
    }

    void placeOrder(Order order) {
        // aggiungere order nella coda
        // richiamare order.execute(); sul primo elemento
        //   estratto

    }
}
```

Esempio di implementazione del pattern Command

Command pattern

```
//ConcreteCommand Class.
class BuyStockOrder implements Order {
    private StockTrade stock;
    public BuyStockOrder ( StockTrade st) {
        stock = st;
    }
    public void execute( ) {
        stock.buy();
    }
}
```

Esempio di implementazione del pattern Command

Command pattern

```
//ConcreteCommand Class.
class SellStockOrder implements Order {
    private StockTrade stock;
    public SellStockOrder ( StockTrade st) {
        stock = st;
    }
    public void execute() {
        stock.sell();
    }
}
```

Esempio di implementazione del pattern Command

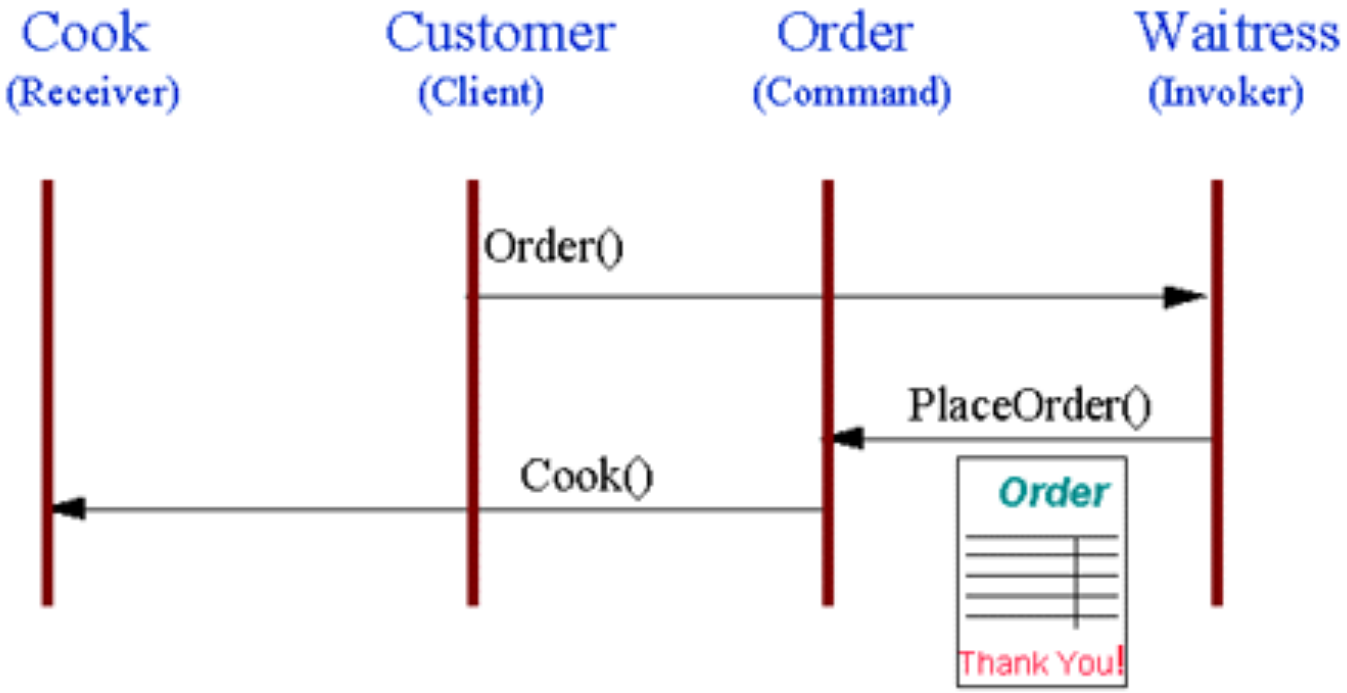
Command pattern

```
// Client
public class Client {
    public static void main(String[] args) {
        StockTrade stock = new StockTrade();
        BuyStockOrder bsc = new BuyStockOrder (stock);
        SellStockOrder ssc = new SellStockOrder (stock);
        Agent agent = new Agent();

        agent.placeOrder(bsc); // Buy Shares
        agent.placeOrder(ssc); // Sell Shares
    }
}
```

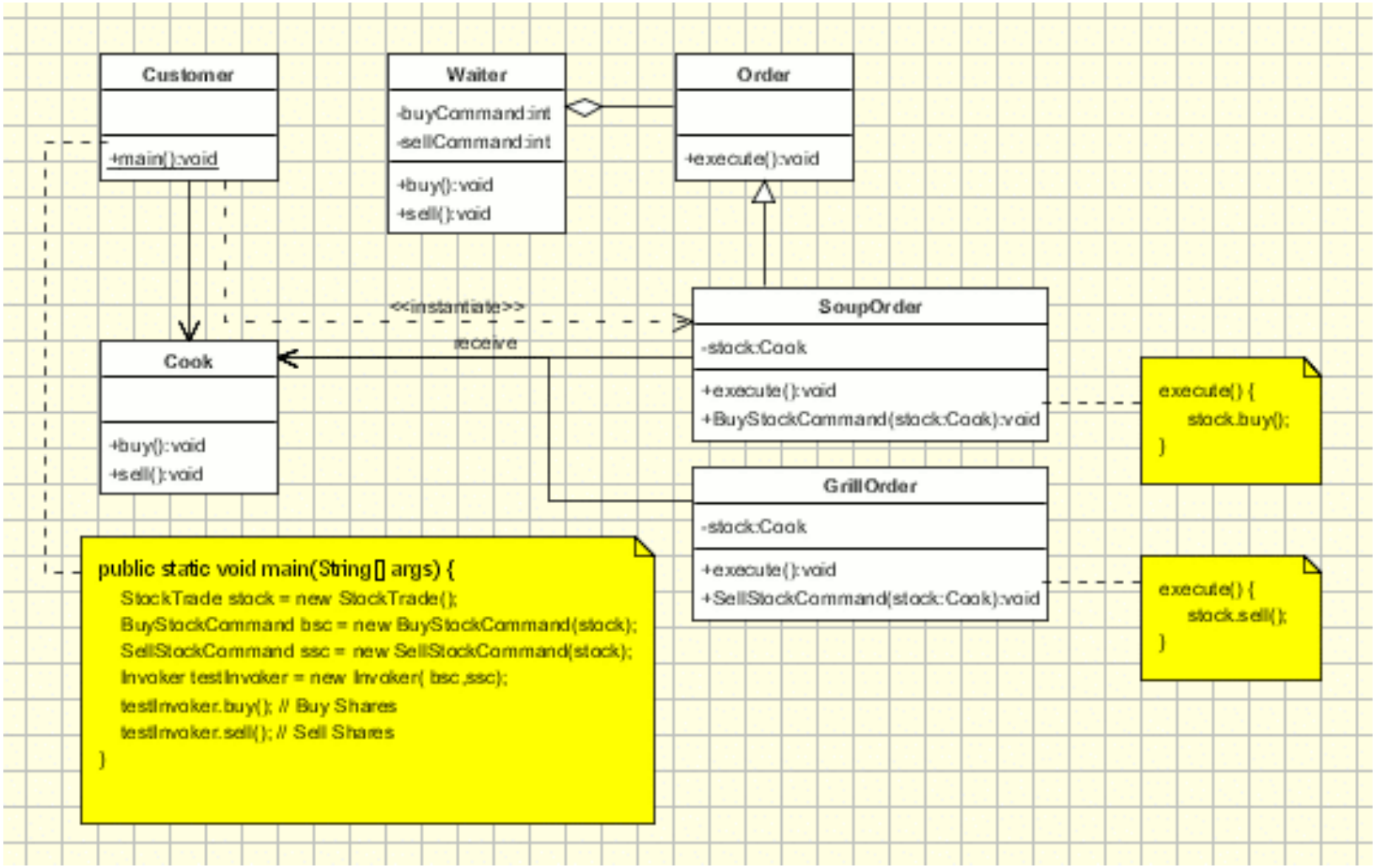
Esempio di implementazione del pattern Command

Command pattern



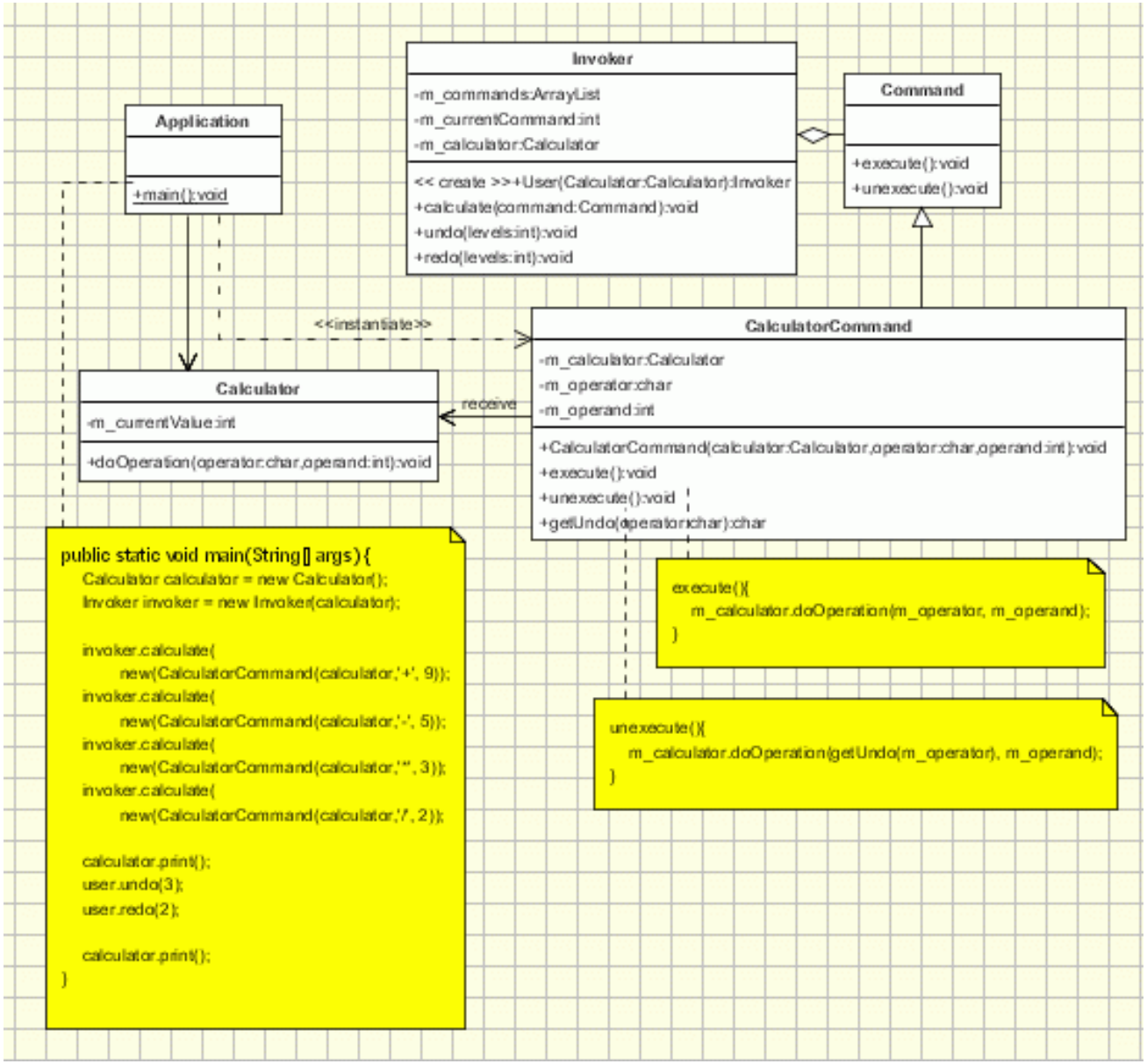
Esemplificazione del meccanismo del `Command pattern`

Esempio Command



Esempio di `Command pattern`. Ordinazione al ristorante.

Esempio Command



Esempio di `Command pattern`. Calcolatrice.

Esempio di operazioni annullabili

Esercizi

- Officina riparazione auto
  - autovetture con problemi diversi
  - reception prende informazioni e colloca la macchina in una coda per la riparazione
  - le informazioni sull'ordine sono consegnate al proprietario per effettuare il ritiro
  - al turno corrispondente il meccanico ripara l'auto

Esercizi

- File System
  - supportati `Windows` e `Unix`
  - utility per `open`, `write` e `close`

Esercizi

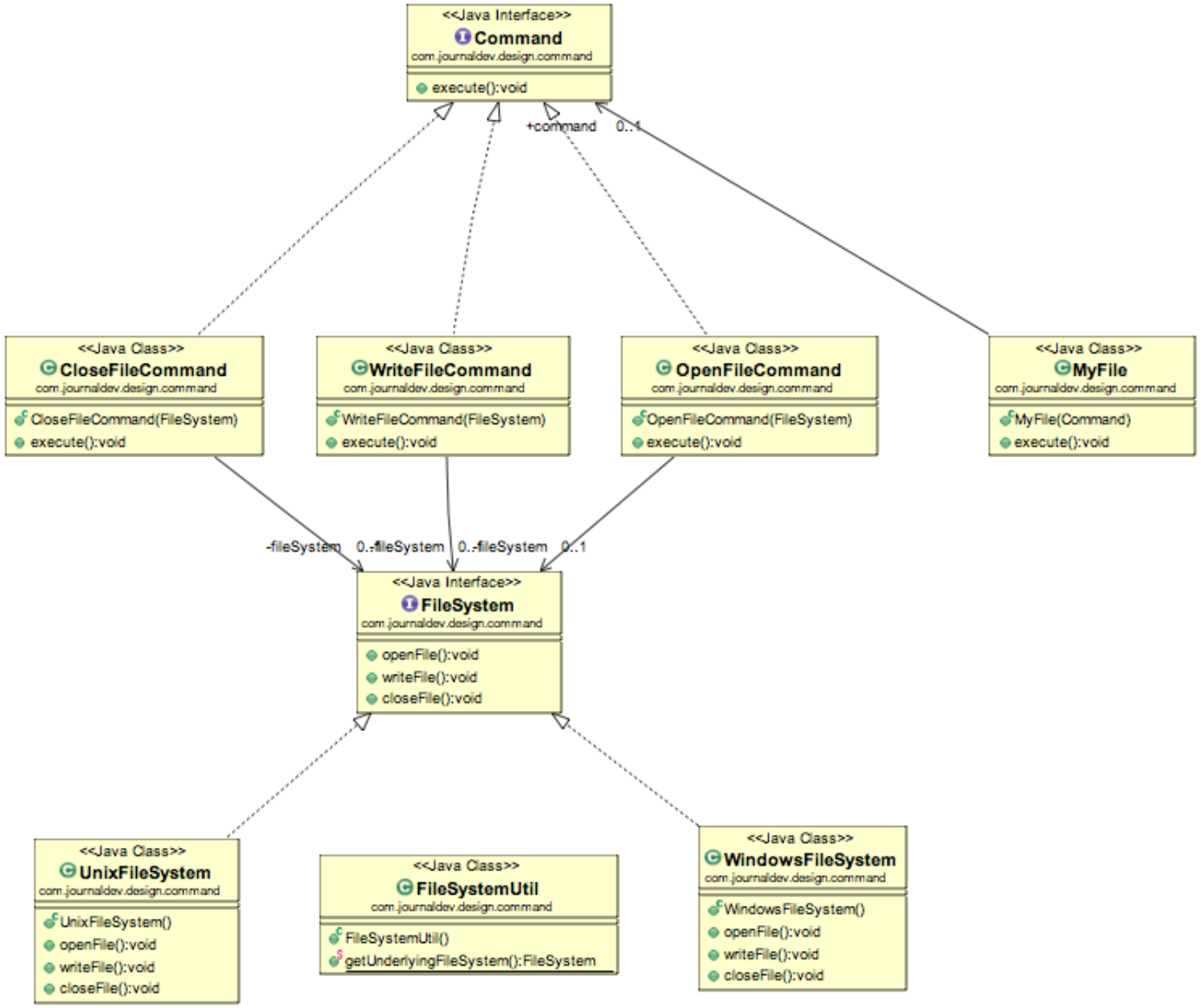


Diagramma delle classi dell'esercizio

Considerazioni

- Command Pattern è usato in (JDK)
  - Runnable interface (java.lang.Runnable)
  - Swing Action(javax.swing.Action)

Interpreter

- Scopo
  - *Dato un linguaggio, definisce la rappresentazione per la sua grammatica insieme ad un interprete che usa la rappresentazione per interpretare frasi del linguaggio*

- Motivazione
  - In particolari tipi di problemi può essere utile esprimere istanze di un problema come frasi di un linguaggio semplice
    - e.g., *pattern string matching*
    - *regular expressions*
  - Esempi
    - *compilatore Java*
    - *Google translator*

Interpreter

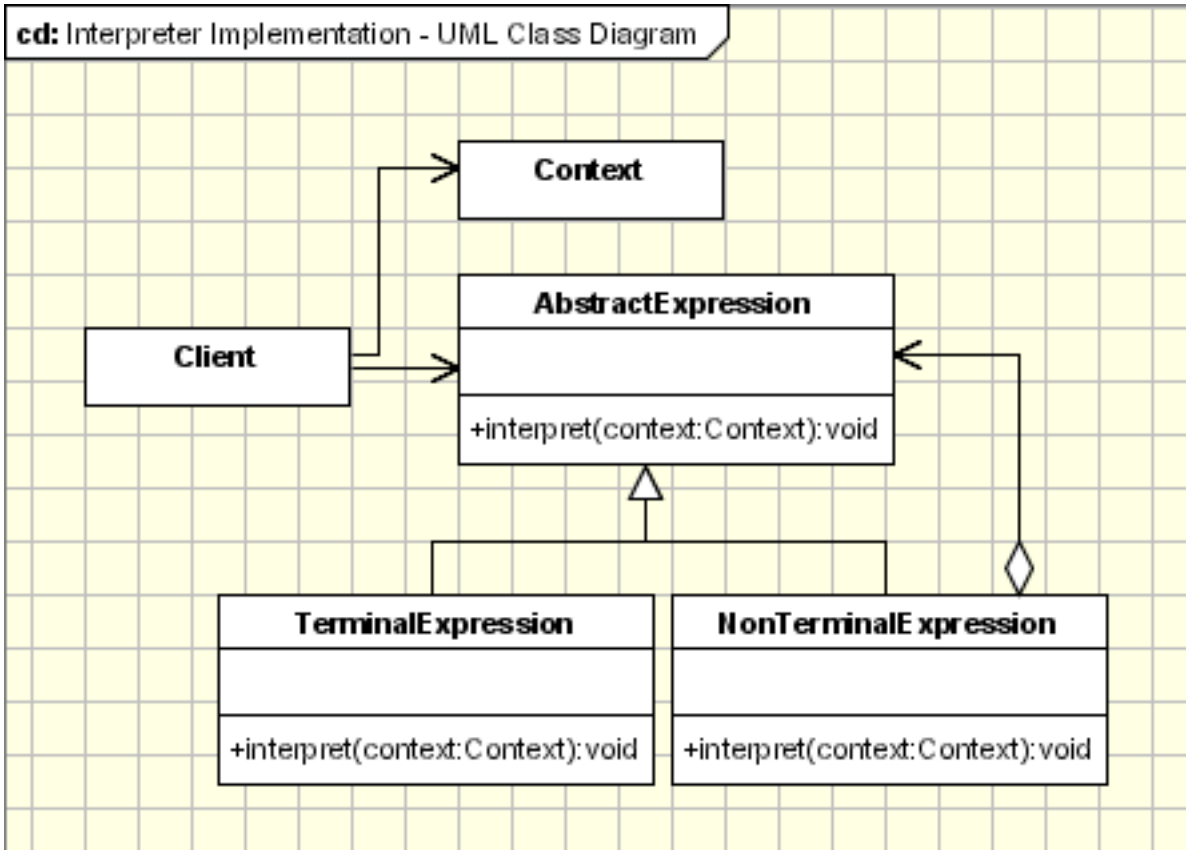
- Applicabilità
  - quando la grammatica è semplice
  - quando non abbiamo vincoli di efficienza
  - può essere applicato ad un'area limitata

Interpreter

```
expression ::= literal | alternation | sequence | repetition |
               '(' expression ')'
alternation ::= expression  '|' expression
sequence ::= expression '&' expression
repetition ::= expression '*'
literal ::= 'a' | 'b' | 'c' | ... { 'a' | 'b' | 'c' | ... }*
```

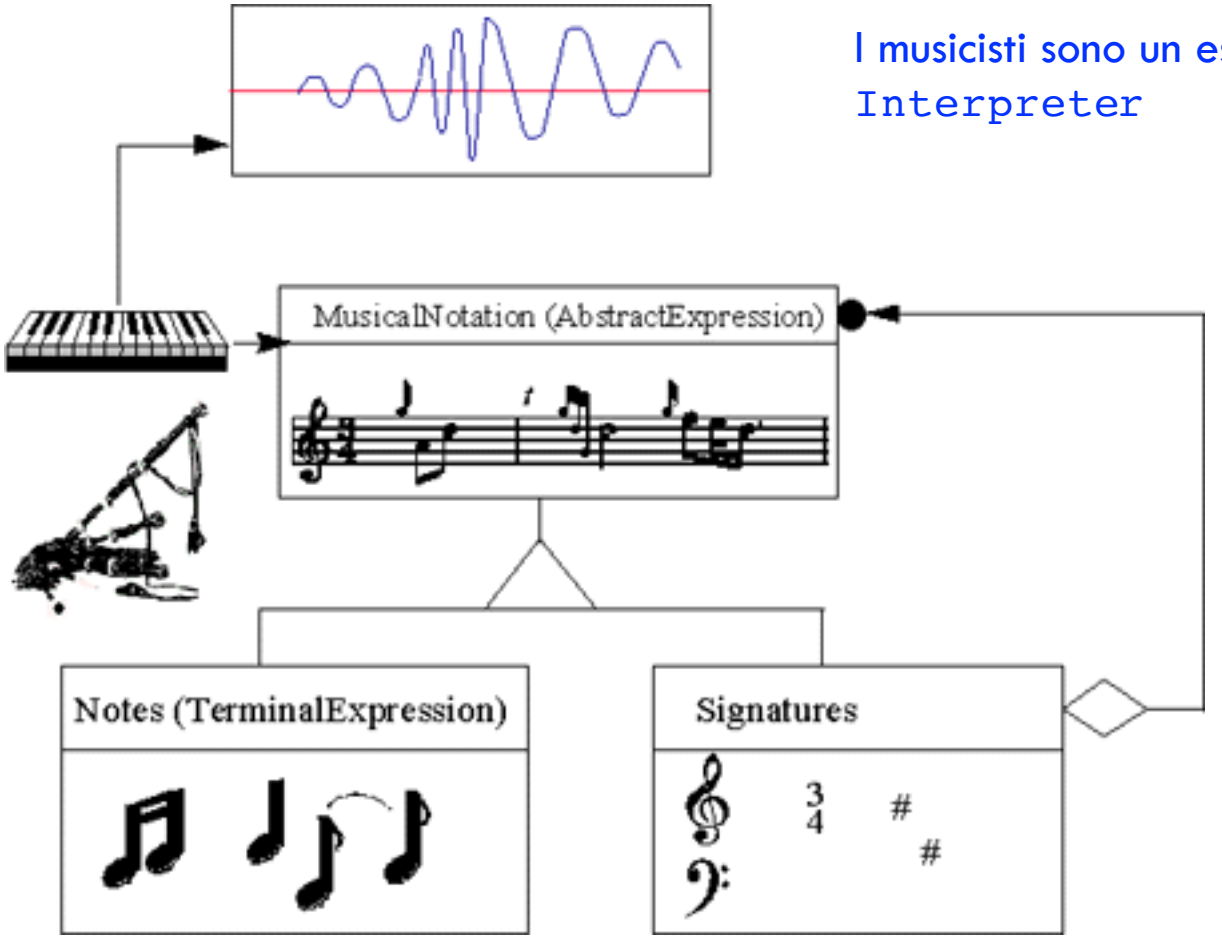
Esempo di grammatica

Interpreter - Struttura



Struttura del pattern `Interpreter`

Interpreter



I musicisti sono un esempio di `Interpreter`

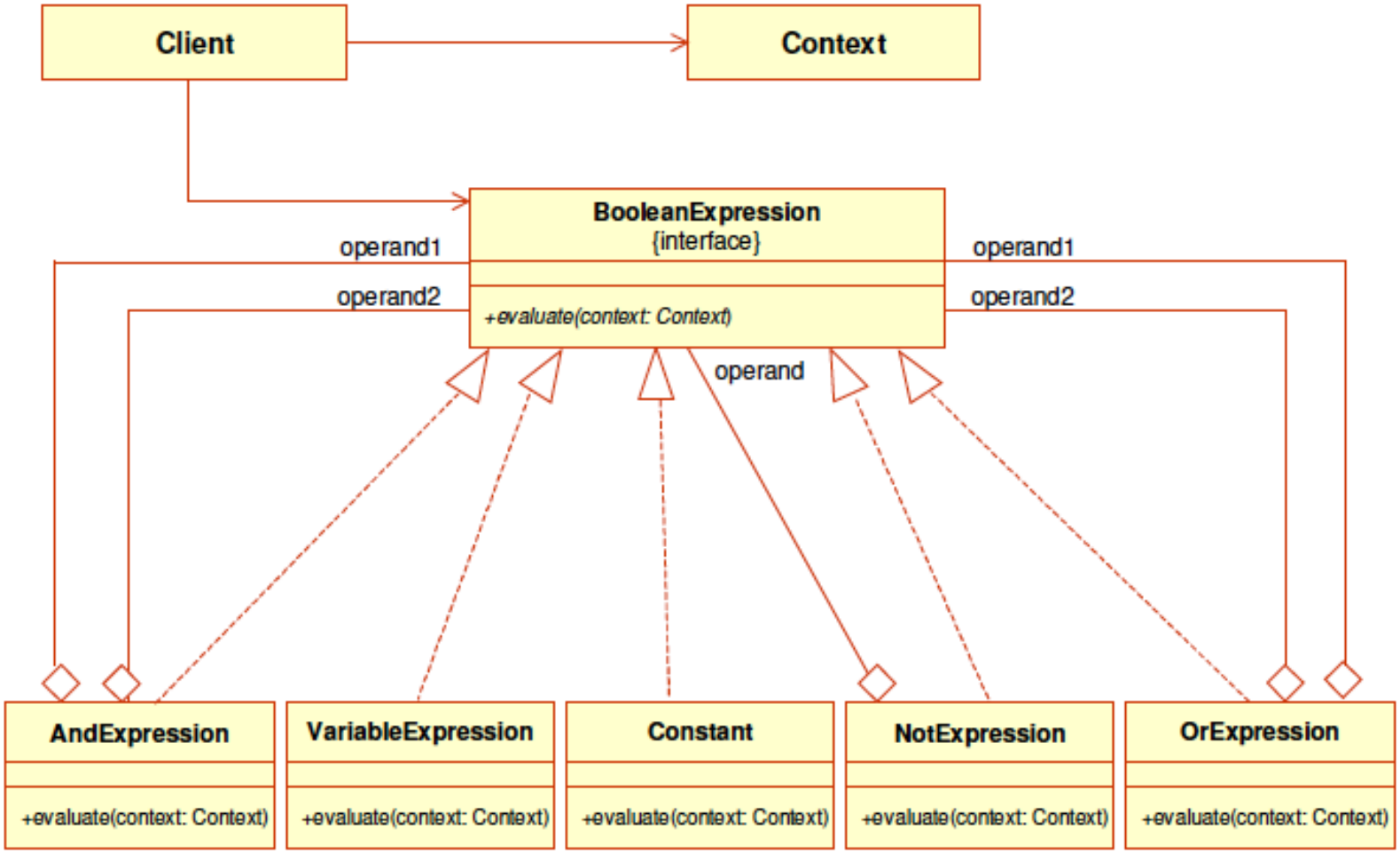
Esemplificazione del pattern `Interpreter`

Esempio Interpreter

```
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'
```

Esempio del pattern `Interpreter`. Grammatica per la generazione di espressioni logiche.

Esempio Interpreter



Esempio del pattern `Interpreter`. Valutazione espressioni logiche.

Considerazioni

- `Interpreter` è usato in (JDK)
  - `java.util.Pattern`
  - `java.text.Format`

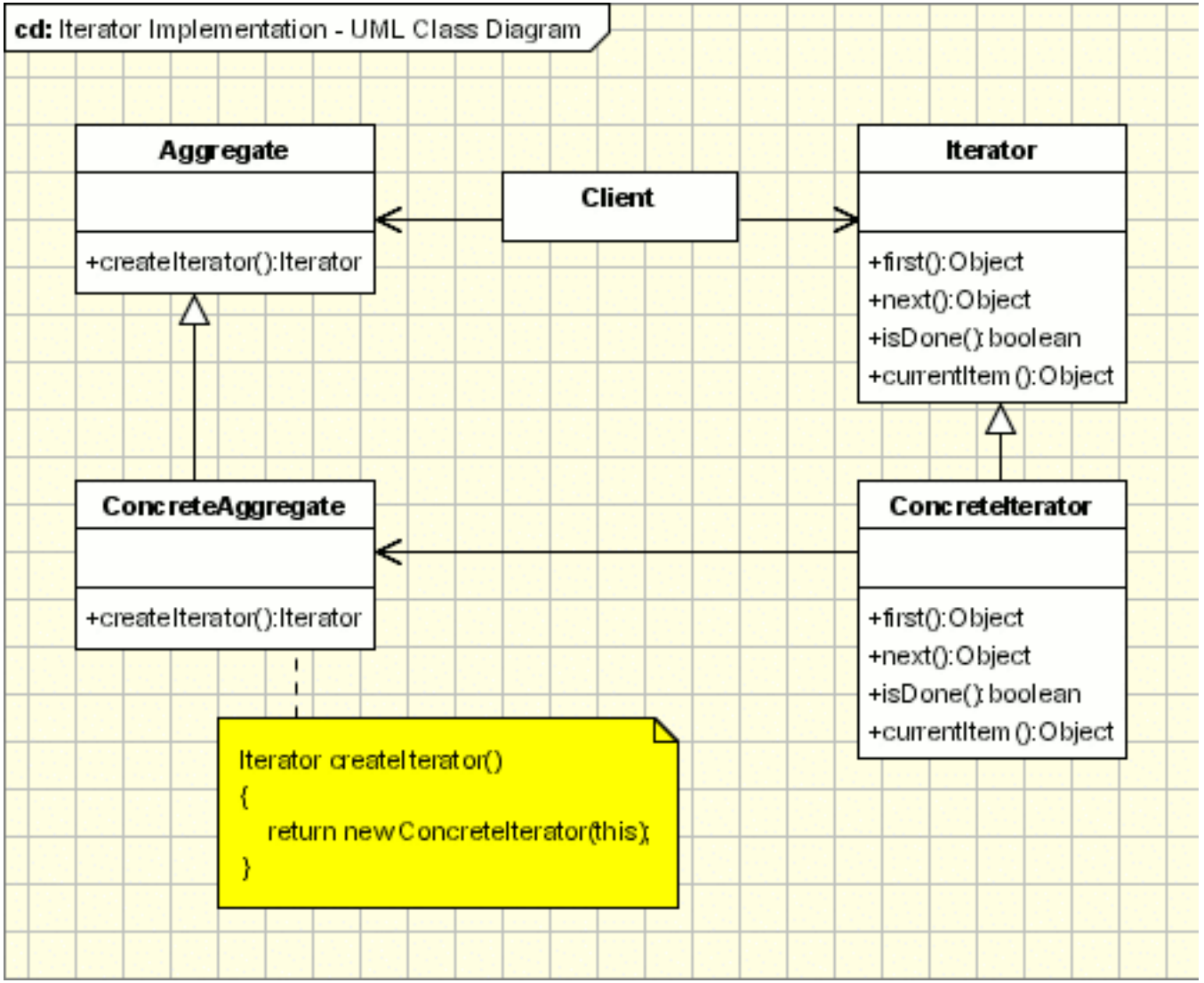
Iterator

- Scopo
  - *Fornisce un modo per accedere agli elementi di un oggetto aggregato nascondendo i dettagli di implementazione.*
- Motivazione
  - Un oggetto aggregato dovrebbe fornire un metodo per accedere agli elementi senza esporre la struttura interna

Iterator

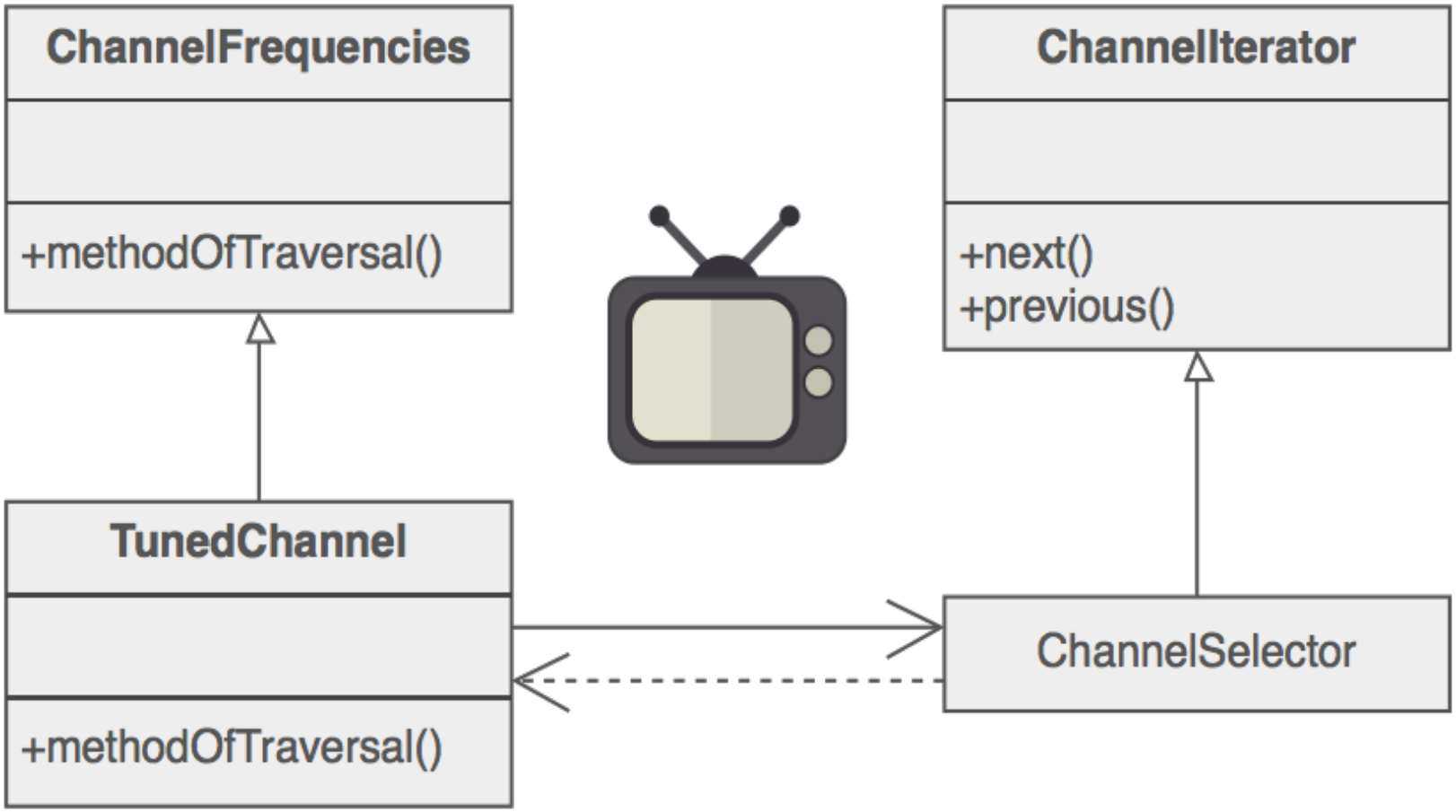
- Applicabilità
  - accedere a contenuti di una collezione senza esporre la sua struttura interna
  - supportare più scorrimenti simultanei di una collezione
  - fornire un'interfaccia uniforme per lo scorrimento di collezioni differenti

Iterator - Struttura



Struttura del pattern `Iterator`

Iterator



Esemplificazione del pattern `Iterator`

Iterator

```
interface IIterator
{
        public boolean hasNext();
        public Object next();
}

interface IContainer
{
        public IIterator createIterator();
}
```

Esempio di implementazione del pattern `Interator`. Collezione di libri.

Iterator

```
class BooksCollection implements IContainer
{
        private String m_titles[] = {"Design
Patterns","1","2","3","4"};

        public IIterator createIterator()
        {
                BookIterator result = new BookIterator();
                return result;
        }


        private class BookIterator implements IIterator
        {
                ...
        }
}
```

Esempio di implementazione del pattern `Interator`. Collezione di libri.

Iterator

```
private class BookIterator implements IIterator
        {
                private int m_position;

                public boolean hasNext()
                {
                        if (m_position < m_titles.length)
                                return true;
                        else
                                return false;
                }
                public Object next()
                {
                        if (this.hasNext())
                                return m_titles[m_position++];
                        else
                                return null;
                }
        }
```

Esempio di implementazione del pattern `Interator`. Collezione di libri.

Iterator

- Iterator
  - interfaccia del framework Collections che permette di iterare su una collezione

```
List<String> strings = new ArrayList<String>();
strings.add("Autoboxing & Auto-Unboxing");
strings.add("Generics");
strings.add("Static imports");
strings.add("Enhanced for loop");
//. . .
Iterator<String> i = strings.iterator();
while (i.hasNext()) {
String string = i.next();
System.out.println(string);
}
```


Considerazioni

- `Interator` è usato in (JDK)
  - `Java Collection Framework`
  - `java.util.Scanner`

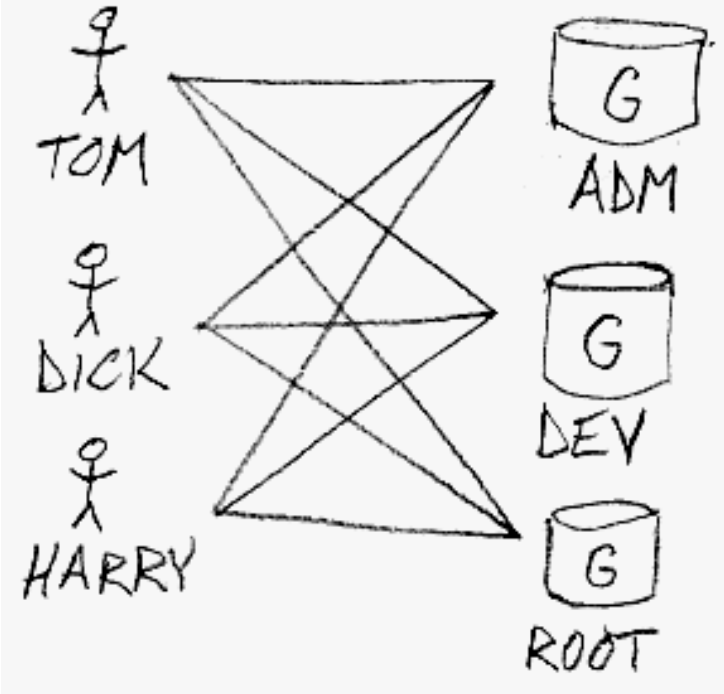
Mediator

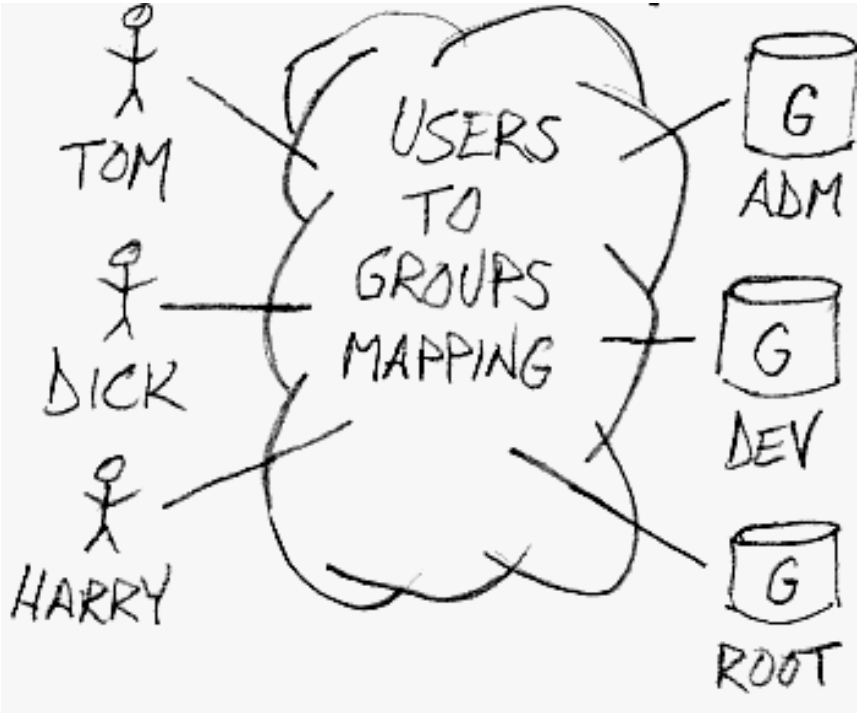
- Scopo
  - *Definire un oggetto che incapsula il meccanismo di interazione di oggetti, consentendo il loro disaccoppiamento in modo da variare facilmente le interazioni tra di loro.*
- Motivazione
  - permettere di modificare agilmente le politiche di interazione, poiché le entità coinvolte devono fare riferimento al loro interno solamente al mediatore

Mediator

- Applicabilità
  - Un insieme di oggetti comunicano in un modo ben definito ma complesso. Le interdipendenze non strutturate e difficili da capire
  - Il riuso di un oggetto è difficile perché si riferisce e relaziona molti altri oggetti
  - Un comportamento distribuito tra diverse classi dovrebbe essere customizzabile senza l'uso di molte sottoclassi

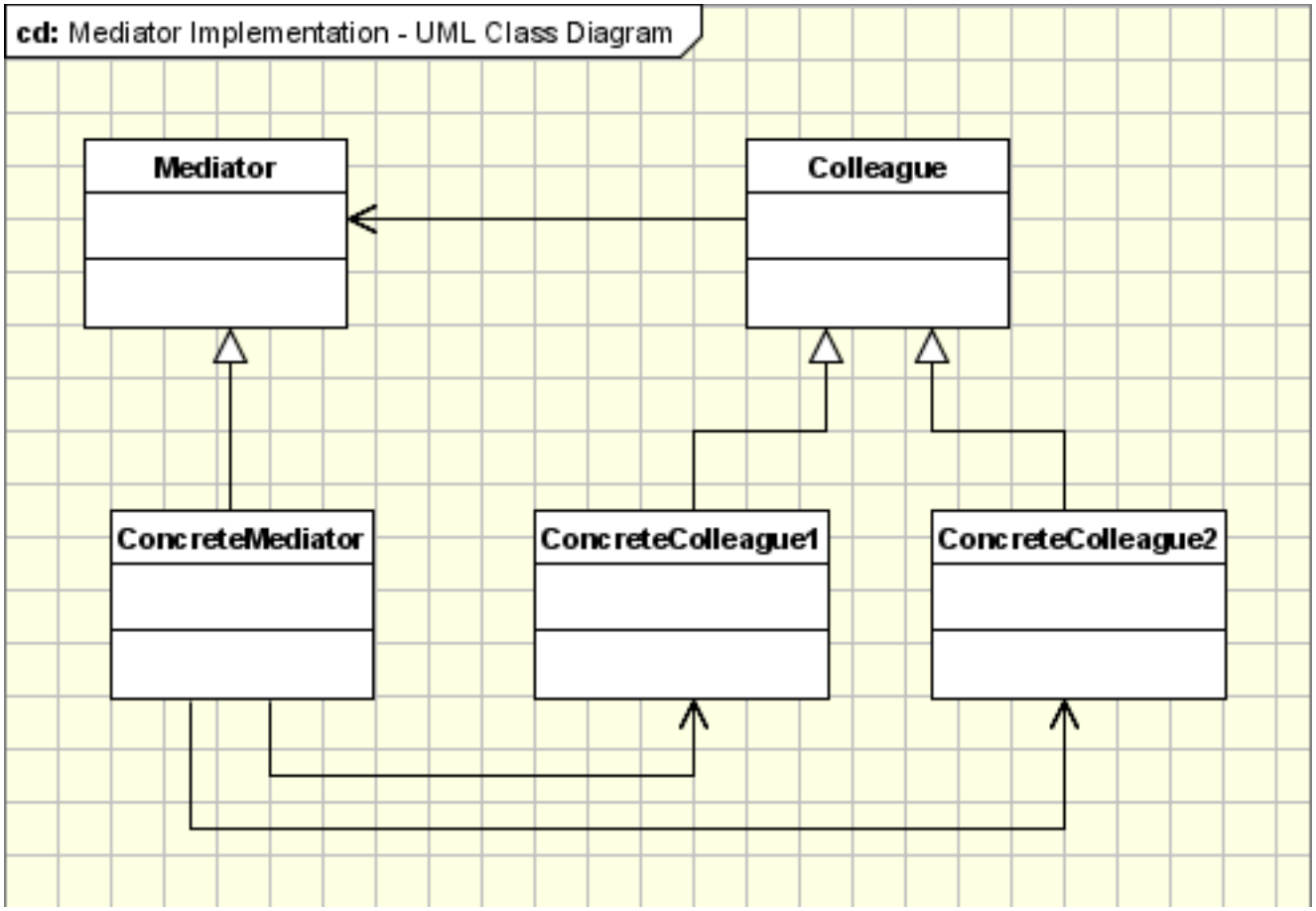
Mediator





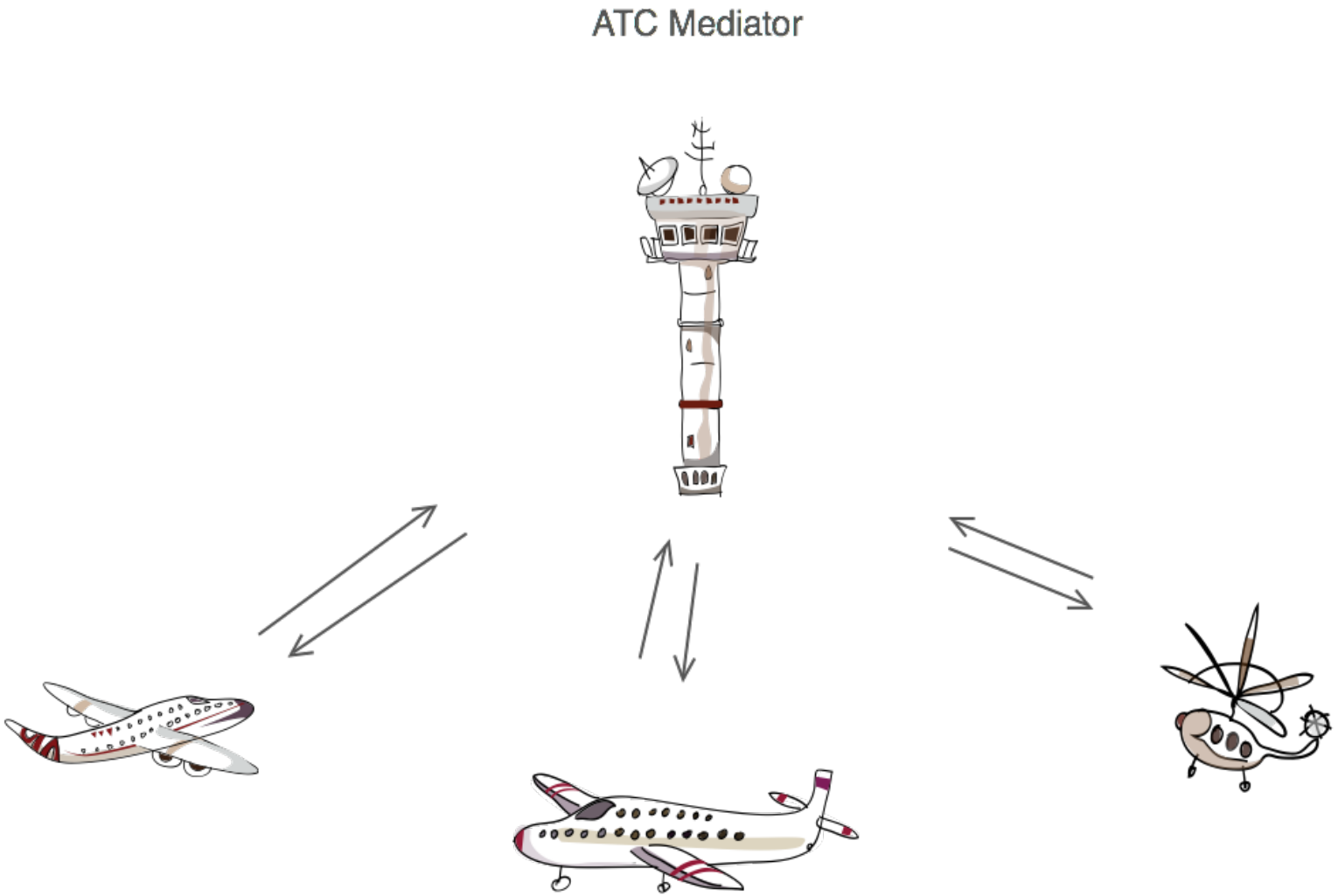
Esemplificazione del pattern `Mediator`

Mediator - Struttura



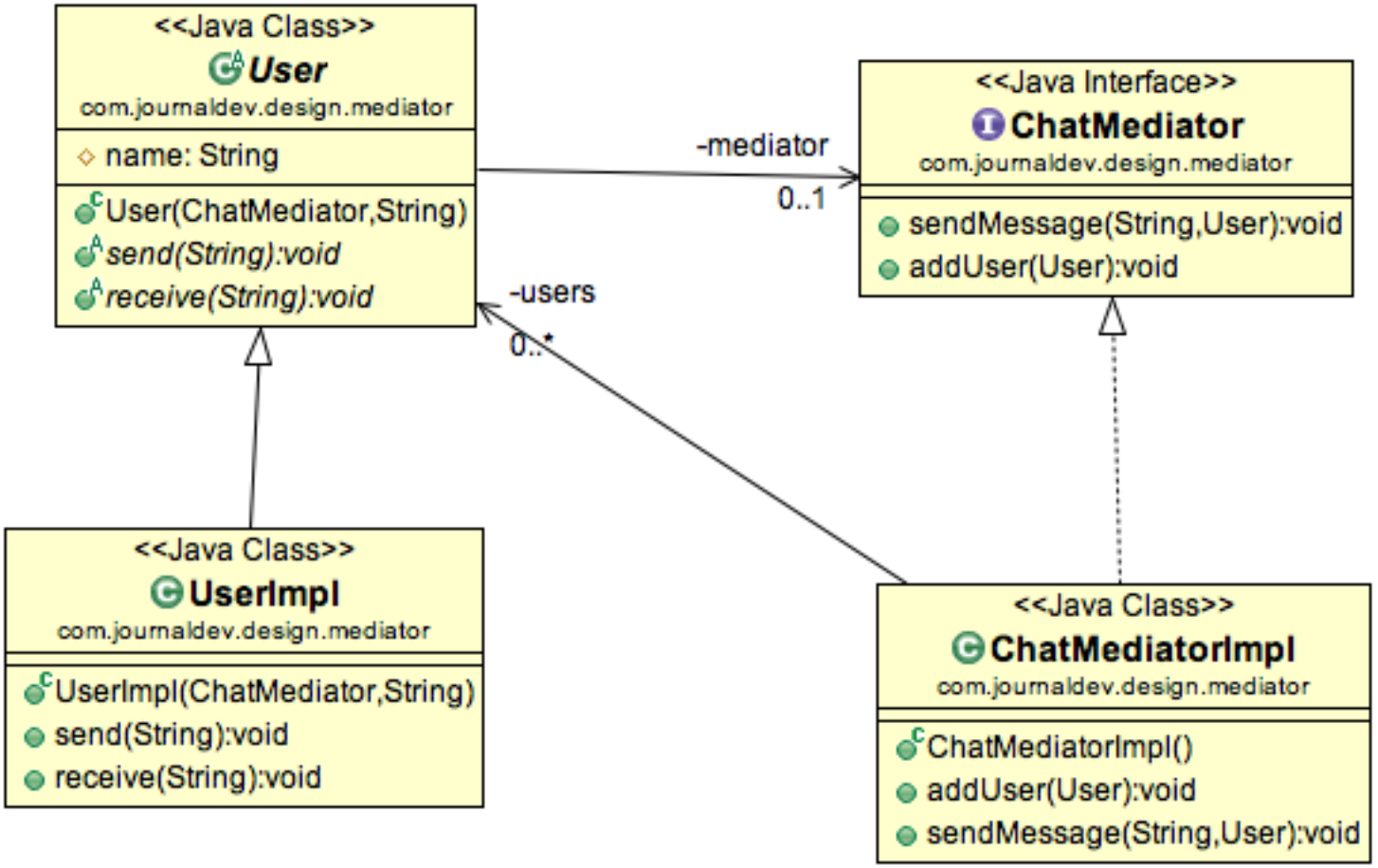
Struttura del pattern `Mediator`

Mediator



Esemplificazione del pattern `Mediator`

Esempio Mediator



Esempio di implementazione del pattern `Mediator`. Applicazione per Chat.

Esempio Mediator

```
public interface ChatMediator {
public void sendMessage(String msg, User user);
void addUser(User user);
}

public abstract class User {
protected ChatMediator mediator;
protected String name;
public User(ChatMediator med, String name){
this.mediator=med;
this.name=name;
}
public abstract void send(String msg);
public abstract void receive(String msg);
}
```


Esempio di implementazione del pattern `Mediator`. Applicazione per Chat.

Esempio Mediator

```
import java.util.ArrayList;
import java.util.List;
public class ChatMediatorImpl implements ChatMediator {
private List<User> users;
  public ChatMediatorImpl(){
  this.users=new ArrayList<>();
  }
  @Override
  public void addUser(User user){
  this.users.add(user);
  }
  @Override
  public void sendMessage(String msg, User user) {
        for(User u : this.users){
        //message should not be received by the user sending it
                if(u != user){
                u.receive(msg);
                }
        }
  }
}
```

Esempio di implementazione del pattern `Mediator`. Applicazione per Chat.

Esempio Mediator

```
public class UserImpl extends User {
  public UserImpl(ChatMediator med, String name) {
  super(med, name);
  }
  @Override
  public void send(String msg){
  System.out.println(this.name+": Sending Message="+msg);
  mediator.sendMessage(msg, this);
  }

  @Override
  public void receive(String msg) {
  System.out.println(this.name+": Received Message:"+msg);
  }
}
```

Esempio di implementazione del pattern `Mediator`. Applicazione per Chat.

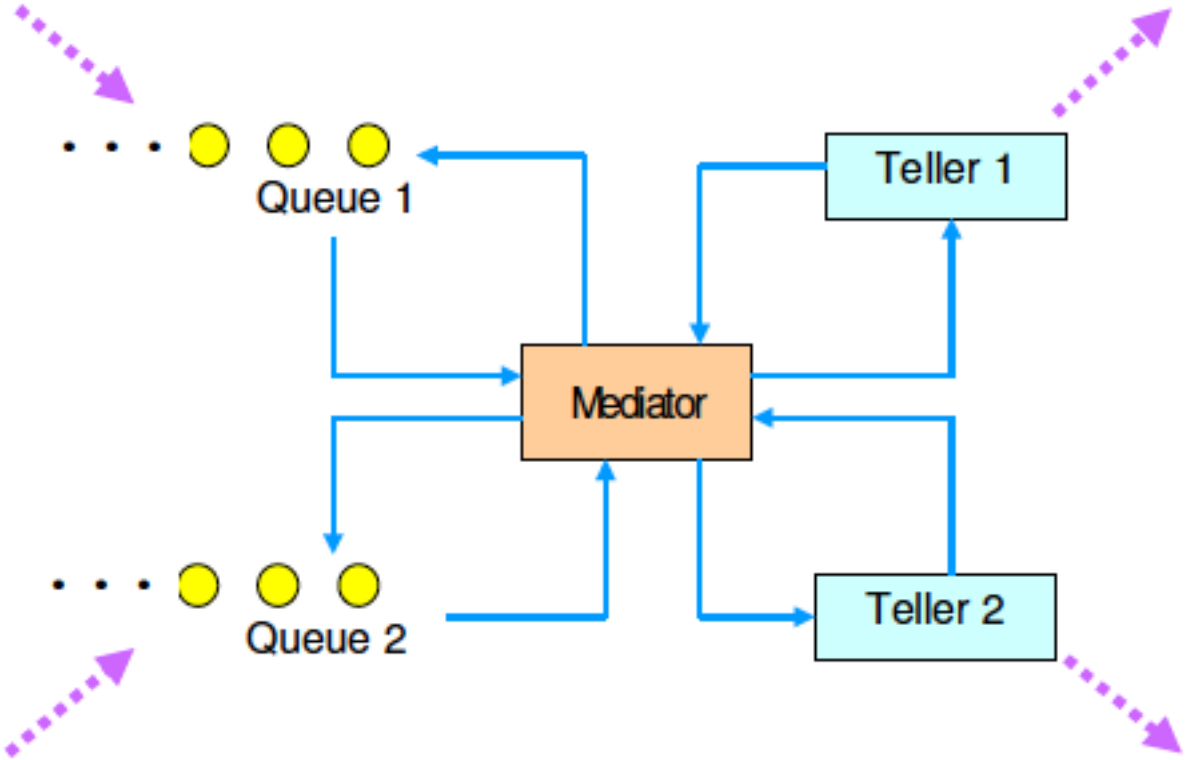
Esempio Mediator

```
public class ChatClient {
  public static void main(String[] args) {
    ChatMediator mediator = new ChatMediatorImpl();
    User user1 = new UserImpl(mediator, "Pankaj");
    User user2 = new UserImpl(mediator, "Lisa");
    User user3 = new UserImpl(mediator, "Saurabh");
    User user4 = new UserImpl(mediator, "David");
    mediator.addUser(user1);
    mediator.addUser(user2);
    mediator.addUser(user3);
    mediator.addUser(user4);
    user1.send("Hi All");
   }
}
```

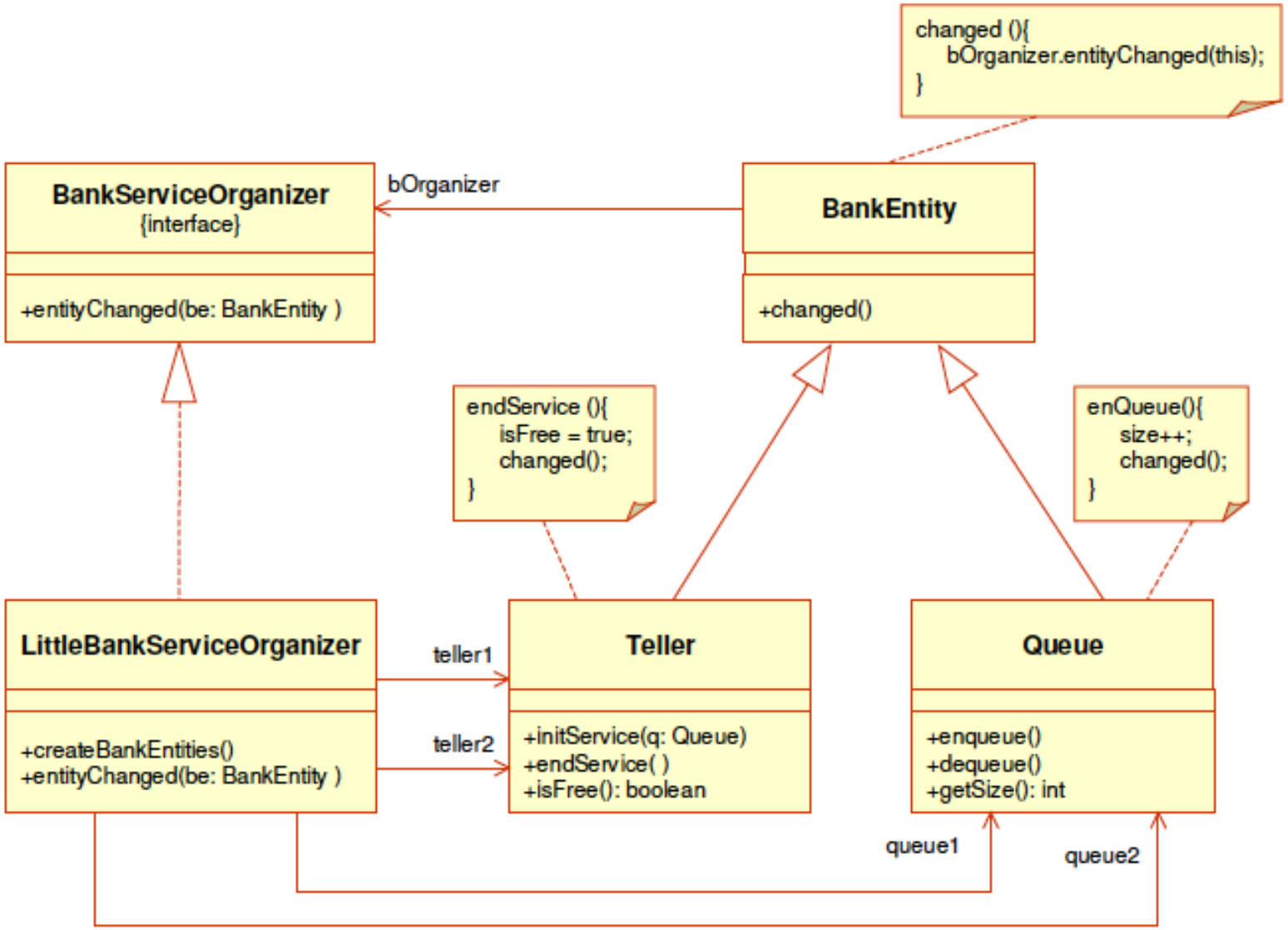
Esempio di implementazione del pattern `Mediator`. Applicazione per Chat.

Esercizi

- Simulatore di code per un sistema bancario



Esercizi



Esercizio pattern `Mediator`. Simulatore di code per un sistema bancario.

Considerazioni

- Mediator è usato in (JDK)
  - java.util.Timer (scheduleXXX())
  - Java Concurrency Executor (execute())
  - java.lang.reflect.Method (invoke())
  - Java Message Service (JMS) API
  - Molto usato per interfacce utente (GUI framework)

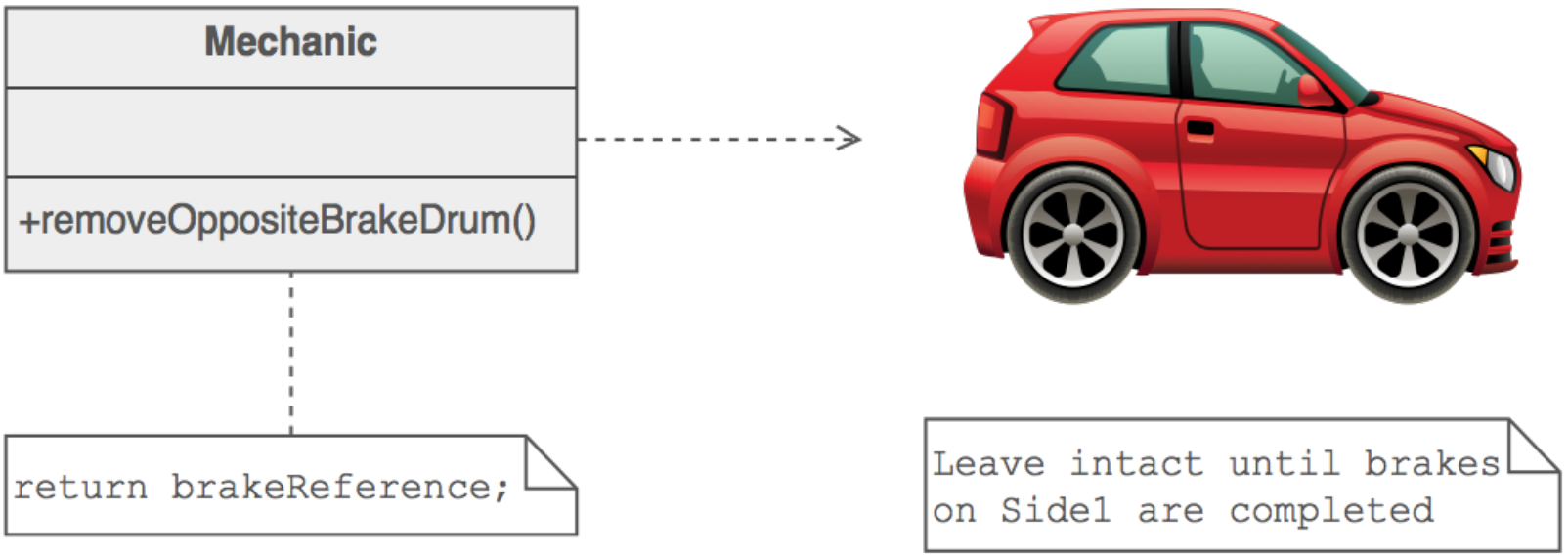
Memento

- Scopo
  - *L'intento di questo modello è di catturare lo stato interno di un oggetto senza violare incapsulamento e fornendo così un mezzo per ripristinare l'oggetto allo stato iniziale quando necessario.*
- Motivazione
  - A volte si rende necessaria l'acquisizione dello stato interno di un oggetto in un certo istante e ripristinare successivamente l'oggetto a quello stato
  - Il processo è utile in caso di errori
    - e.g., calcolatrice che mantiene la lista delle operazioni precedenti

Memento

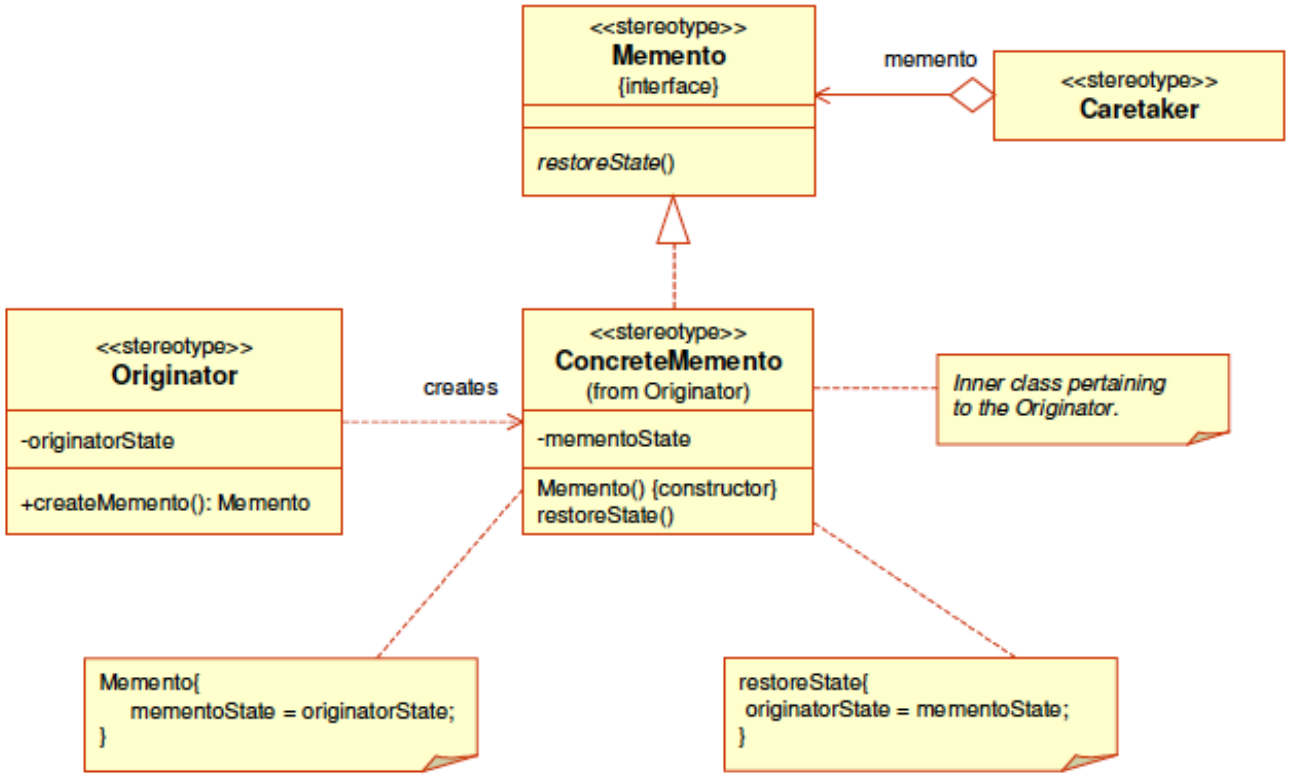
- Applicabilità
  - viene utilizzato quando uno stato di un oggetto deve essere catturato in modo che possa essere ripristinato a quello stato più tardi
  - in situazioni in cui il passaggio in modo esplicito dello stato dell'oggetto violerebbe incapsulamento

Memento



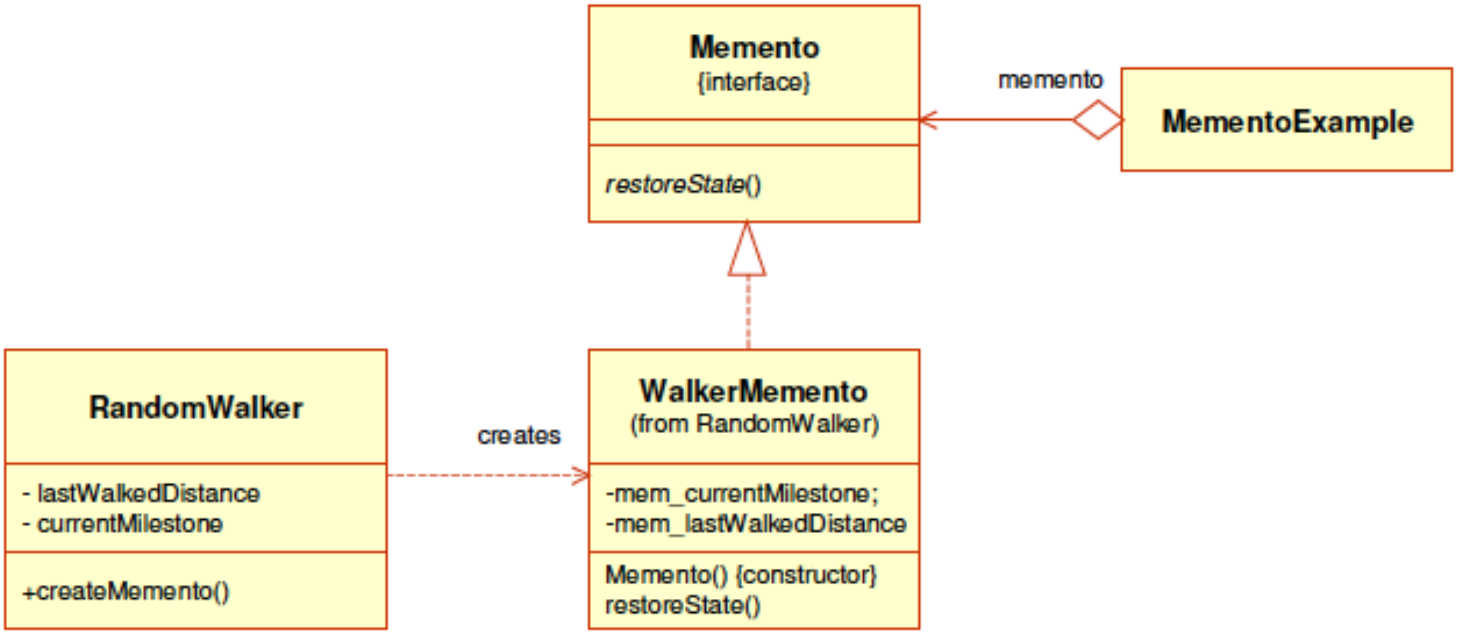
Esemplificazione del pattern `Memento`

Memento - Struttura



Struttura del pattern `Memento`

Memento



Esempio di utilizzo del pattern `Memento`. Viaggiatore e tappe casuali del viaggio.

Esempio Memento

```
public interface Memento {
public void restoreState();
}
```

Esempio di implementazione del pattern `Mediator`. Applicazione Viaggiatore.

Esempio Memento

```
public class RandomWalker {
private int currentMilestone;
private int lastWalkedDistance;
private RandomWalker randomWalker;
public RandomWalker() {
randomWalker = this;
}
public void randomWalk() {
lastWalkedDistance = (int) ( Math.random() * 100 );
currentMilestone = currentMilestone + lastWalkedDistance;
}
public int getCurrentMilestone() {
return currentMilestone;
}
public int getLastWalkedDistance() {
return lastWalkedDistance;
}
public Memento createMemento( ) {
return new WalkerMemento( );
}
```


Esempio Memento

```
class WalkerMemento implements Memento{
private int mem_currentMilestone;
private int mem_lastWalkedDistance;
public WalkerMemento() {
mem_currentMilestone = currentMilestone;
mem_lastWalkedDistance = lastWalkedDistance;
}
public void restoreState() {
currentMilestone = mem_currentMilestone;
lastWalkedDistance = mem_lastWalkedDistance;
}
} //End of class WalkerMemento
} //End of class RandomWalker
```

Esempio di implementazione del pattern `Mediator`. Applicazione Viaggiatore.

Esempio Memento

```
public class MementoExample {
public static void main (String[] arg) {
RandomWalker luke = new RandomWalker();
// Creates a Memento that saves the original state
Memento tripStop = luke.createMemento();
for(int i=1; i<=4 ;i++) {
System.out.println("Starting trip...");
luke.randomWalk();
whereIs( luke );
System.out.println("Do you like this place?" );
if(Math.random() < .4) {
System.out.println("-No!");
// Restores the last saved state
tripStop.restoreState();
whereIs( luke );
}else {
System.out.println("-Yes!");
// Creates a new Memento to save the new state
tripStop = luke.createMemento();
}
}
System.out.println("You reach the km " +luke.getCurrentMilestone());
}
```


Esempio Memento

```
public static void whereIs(RandomWalker rw) {
System.out.print ( "You are now stopped at km " +
rw.getCurrentMilestone()+ ". " );
System.out.println( "This place is "+
rw.getLastWalkedDistance()
+ " kms far from your last stop. ");
}
```

Esempio di implementazione del pattern `Mediator`. Applicazione Viaggiatore.

Esercizio

- Gestione file
  - leggere e scrivere file
  - possibilità di recuperare l'ultimo stato salvato
  - Codice Java
    - `FileWriterUtil.java` (`Memento` classe nascosta)
    - `FileWriterCaretaker.java`
    - `FileWriterClient.java`
  - Scrivere il diagramma delle classi

Esercizio

- Semplice calcolatrice con operazioni di *undo*
  - Addizione di due numeri
  - Possibilità di recuperare l'ultima operazione e ripristino del precedente risultato
- Operazioni di rollback sui database

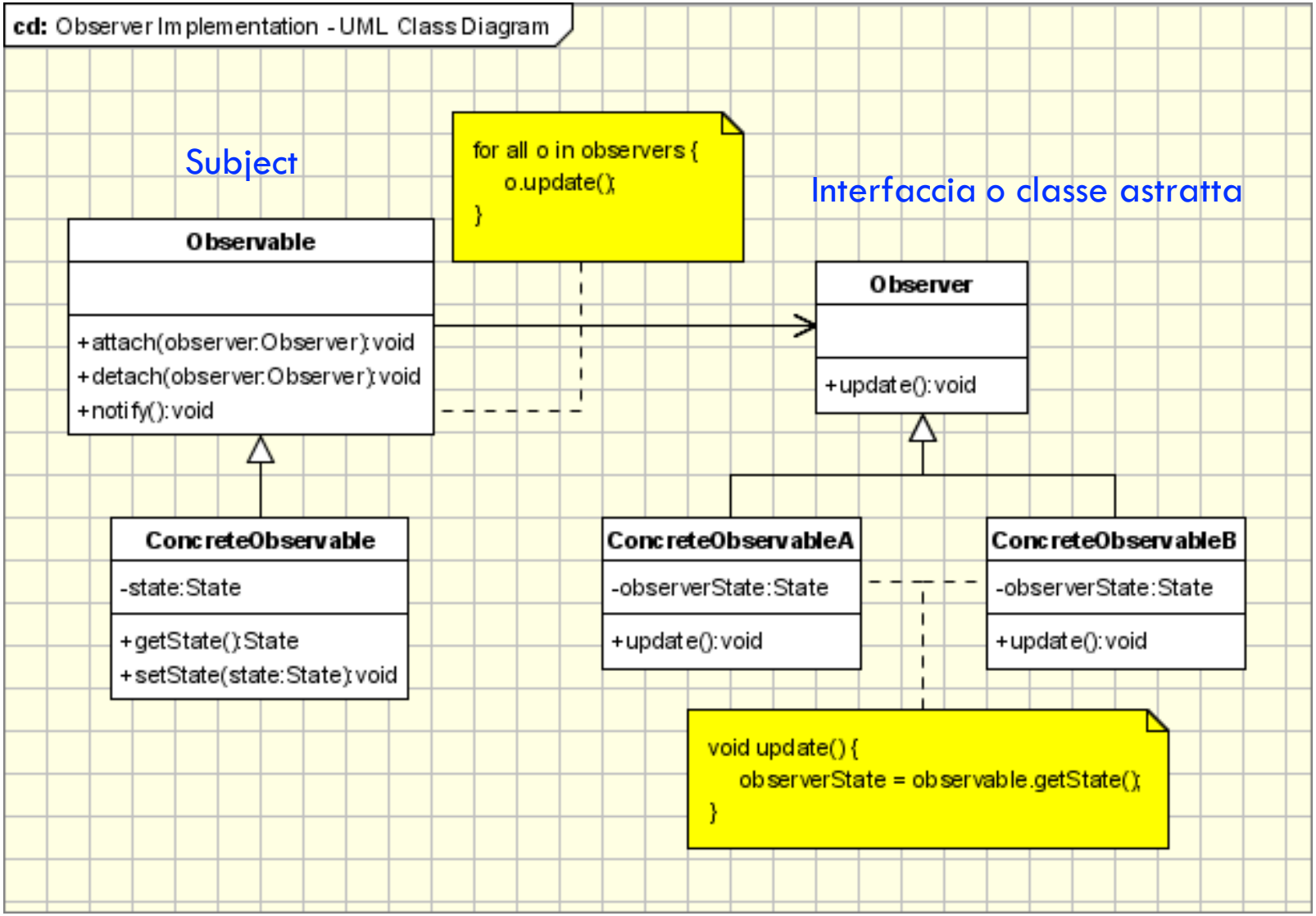
Observer

- Scopo
  - *Definisce una dipendenza una a molti tra oggetti, tale che se un oggetto cambia stato, tutte le sue dipendenze sono notificate e aggiornate automaticamente*
- Anche conosciuto come
  - *Publish-Subscribe*
- Motivazione
  - Un effetto del partizionamento di un sistema in una collezione di classi cooperanti è la necessità di mantenere le consistenze tra gli oggetti
  - Le classi non devono essere fortemente accoppiate perché riducono la loro riusabilità

Observer

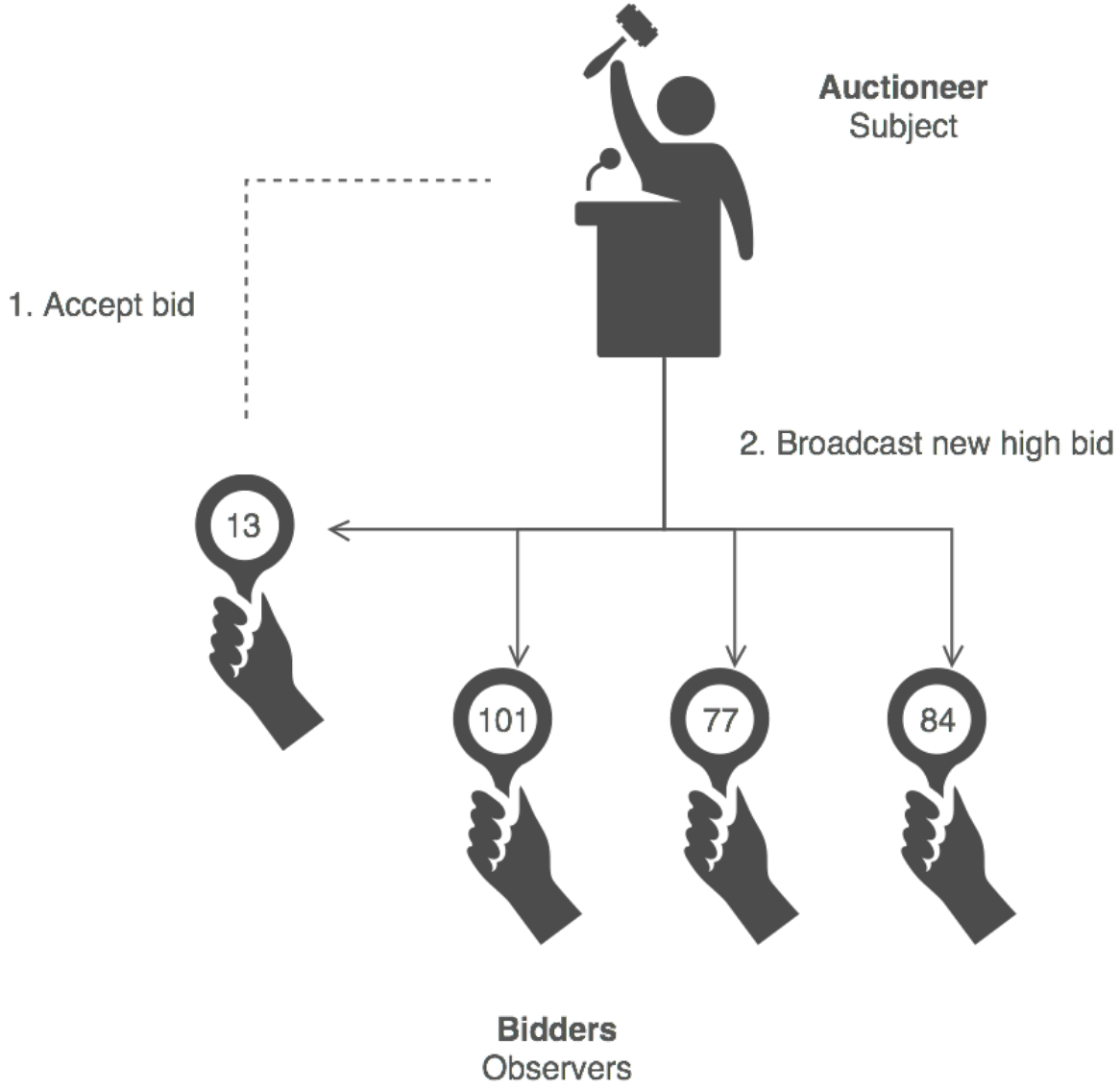
- Applicabilità
  - quando un'astrazione ha due aspetti che dipendono l'uno dall'altro. L'incapsulamento di questi aspetti in oggetti separati permette il loro riuso in modo indipendente
  - quando il cambiamento di un oggetto richiede il cambiamento di altri e non si conoscono quanti oggetti hanno bisogno di essere cambiati

Observer – Struttura



Struttura del pattern `Observer`

Observer



Esemplificazione del pattern `Observer`

Esempio Observer

```
class Subject {  // 1. The "independent" abs
   private Observer[] observers = new Observer[9];
                  // 3. Coupled to base class
   private int totalObs  = 0;
   private int state;
   public void attach( Observer o ) {
    observers[totalObs++] = o; } // 3. Coupled
   public int  getState(){ return state; }
   public void setState( int in ) {
      state = in;
      notifyObservers(); }
   private void notifyObservers() {
      for (int i=0; i < totalObs; i++)
         observers[i].update();
               // 3. Coupled to base class
}  }           // 5. Broadcast events to observers
```

Esempio di implementazione del pattern `Observer`. Conversione numeri.

Esempio Observer

```
abstract class Observer {
// 2. Root of the "dependent" hierarchy
   protected Subject subj;
   public abstract void update(); }

class HexObserver extends Observer {
// 2. Concrete class of the "dependent"
   public HexObserver( Subject s ) {    // hierarchy
      subj = s;  subj.attach( this ); }
// 4. Observers register themselves
   public void update() {
      System.out.print( "  " + Integer.toHexString(
subj.getState() ) );
}  }
// 6. Observers "pull" information


class OctObserver extends Observer { ... }
class BinObserver extends Observer { ... }
```


Esempio Observer

```
public class ObserverDemo {
   public static void main( String[] args ) {
      Subject sub = new Subject();
 // 7. Client configures the number and type of Observers

new HexObserver( sub );  new OctObserver( sub );  new
BinObserver( sub );
      while (true) {
         System.out.print( "\nEnter a number: " );
         sub.setState( legge un intero );
}  }  }
```

Esempio di implementazione del pattern `Observer`. Conversione numeri.

Esercizio

- Comunicazione numeri
  - ad un oggetto (`Subject`) vengono comunicati diversi numeri
    - decide in modo casuale di cambiare il suo stato interno, memorizzando il numero ad esso proposto
  - due oggetti incaricati del monitoraggio dell'oggetto descritto (un `Watcher` e un `Psychologist`),
    - devono avere notizie di ogni suo singolo cambio di stato, per eseguire i propri processi di analisi
  - usare i costruttori di base forniti dalle Java API

Esercizio

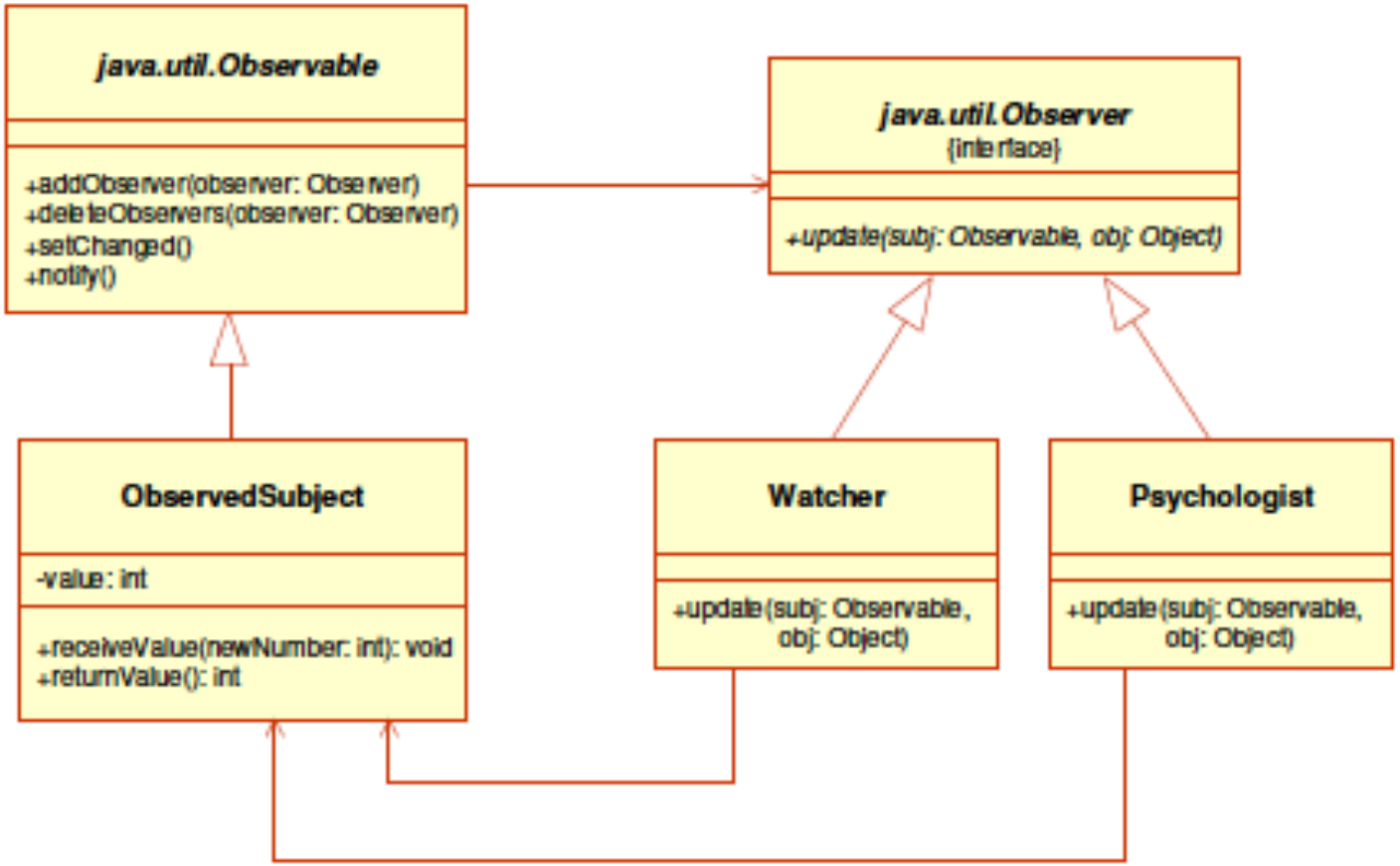


Diagramma delle classi per l'esercizio "comunicazione numeri".

Considerazioni

- Applicazioni in Java
  - usato in `Java Message Serivece` (JMS) insieme al pattern `Mediator`
  - Il framework `Model-View-Controller` (MVC) usa `Observer`
    - `Model` è un `Subject`
    - `Views` sono `Observers`
  - Piattaforme in Java per implementare `Observer`
    - `java.util.Observable`
    - `java.util.Observer`

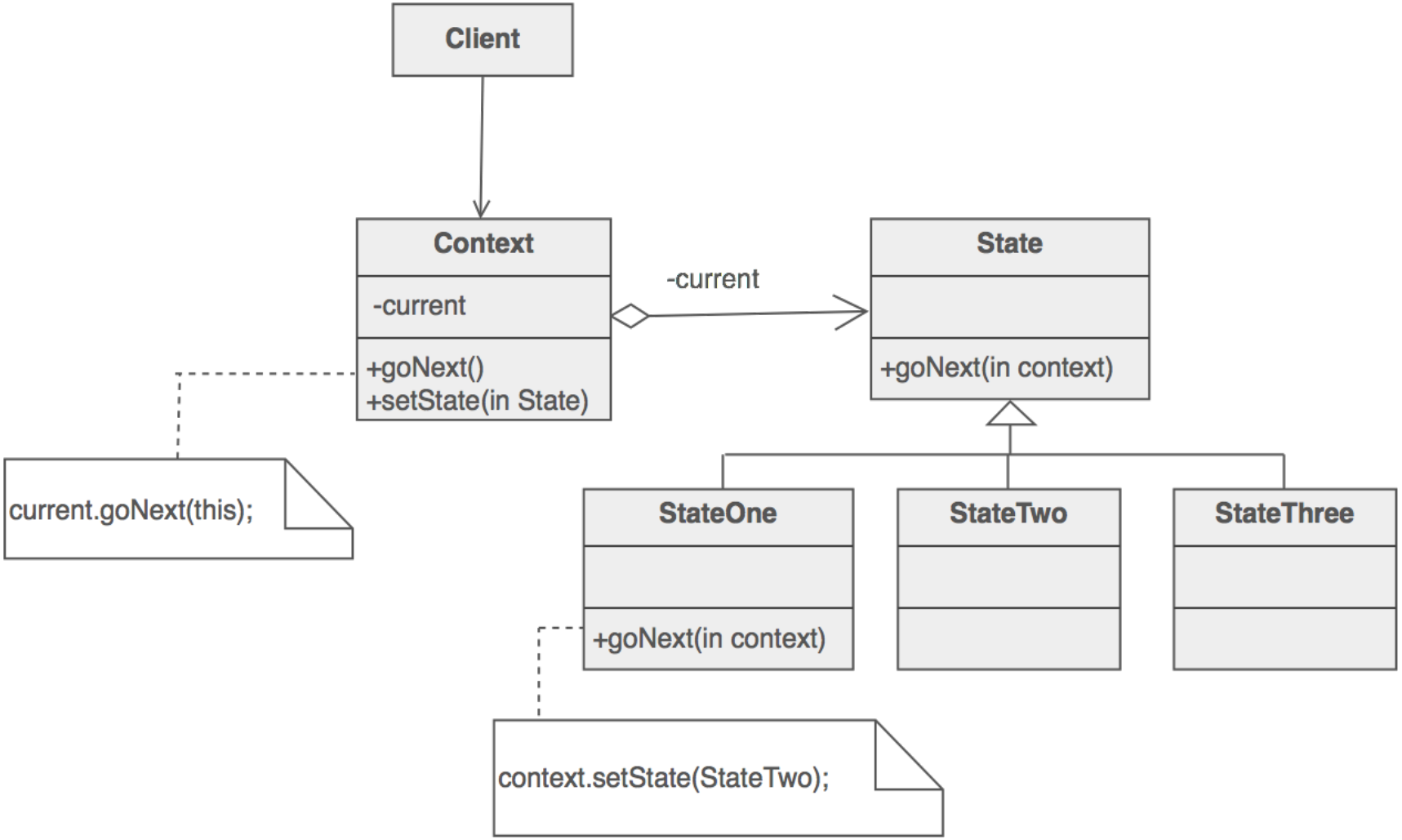
State

- Scopo
  - *Permette ad uno oggetto di cambiare il proprio comportamento quando cambia il suo stato interno. L'oggetto sembrerà cambiare la sua classe.*
- Anche conosciuto come
  - *Objects for States*
- Motivazione
  - soluzione al problema di come rendere il comportamento dipendente dallo stato
  - consente ad un oggetto di cambiare il proprio comportamento a run-time in funzione dello stato in cui si trova

State

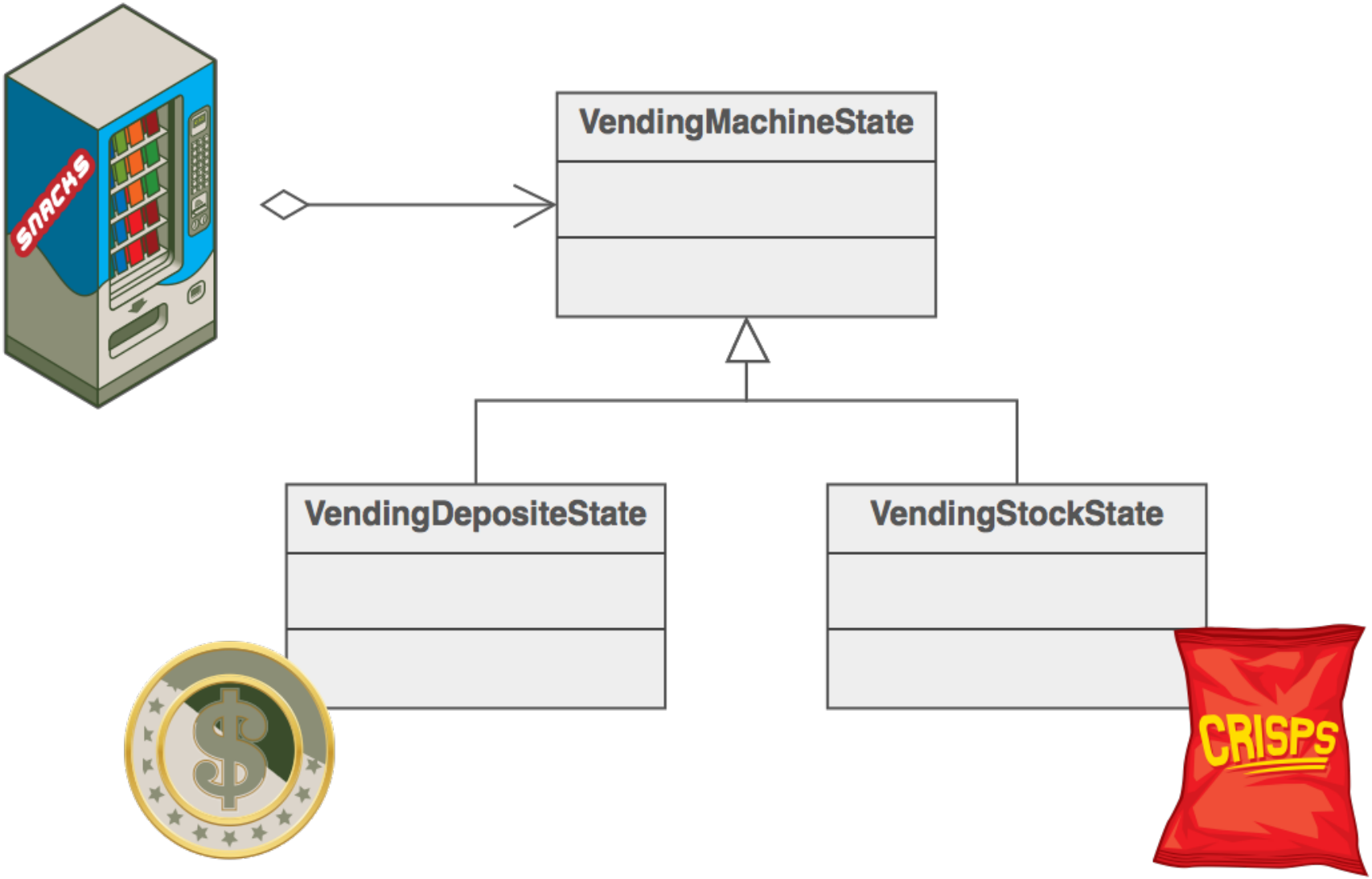
- Applicabilità
  - il comportamento associato ad uno stato dipende solo da una classe
  - la logica che implementa il cambiamento di stato viene implementata in una sola classe piuttosto che con istruzioni condizionali nella classe che implementa il comportamento
  - *evita stati incoerenti*

State - Struttura



Struttura del pattern `State`

State



Esempòlificazione del pattern `State`

Esempio State

```
public interface State {
public void doAction();
}

public class TVStartState implements State {
@Override
public void doAction() {
System.out.println("TV is turned ON");
 }
}

public class TVStopState implements State {
@Override
public void doAction() {
 System.out.println( "TV is turned OFF");
}
}
```


Esempio di implementazione del pattern `State`. Telecomando.

Esempio State

```
public class TVContext implements State {
private State tvState;
public void setState(State state) {
this.tvState=state;
}
public State getState() {
return this.tvState;
}
@Override
public void doAction() {
this.tvState.doAction();
}
}
```


Esempio di implementazione del pattern `State`. Telecomando.

Esempio State

```
public class TVRemote {
public static void main(String[] args) {
TVContext context = new TVContext();
State tvStartState = new TVStartState();
State tvStopState = new TVStopState();
context.setState(tvStartState);
context.doAction();
context.setState(tvStopState);
context.doAction();
}
}
```


Esempio di implementazione del pattern `State`. Telecomando.

Esercizio

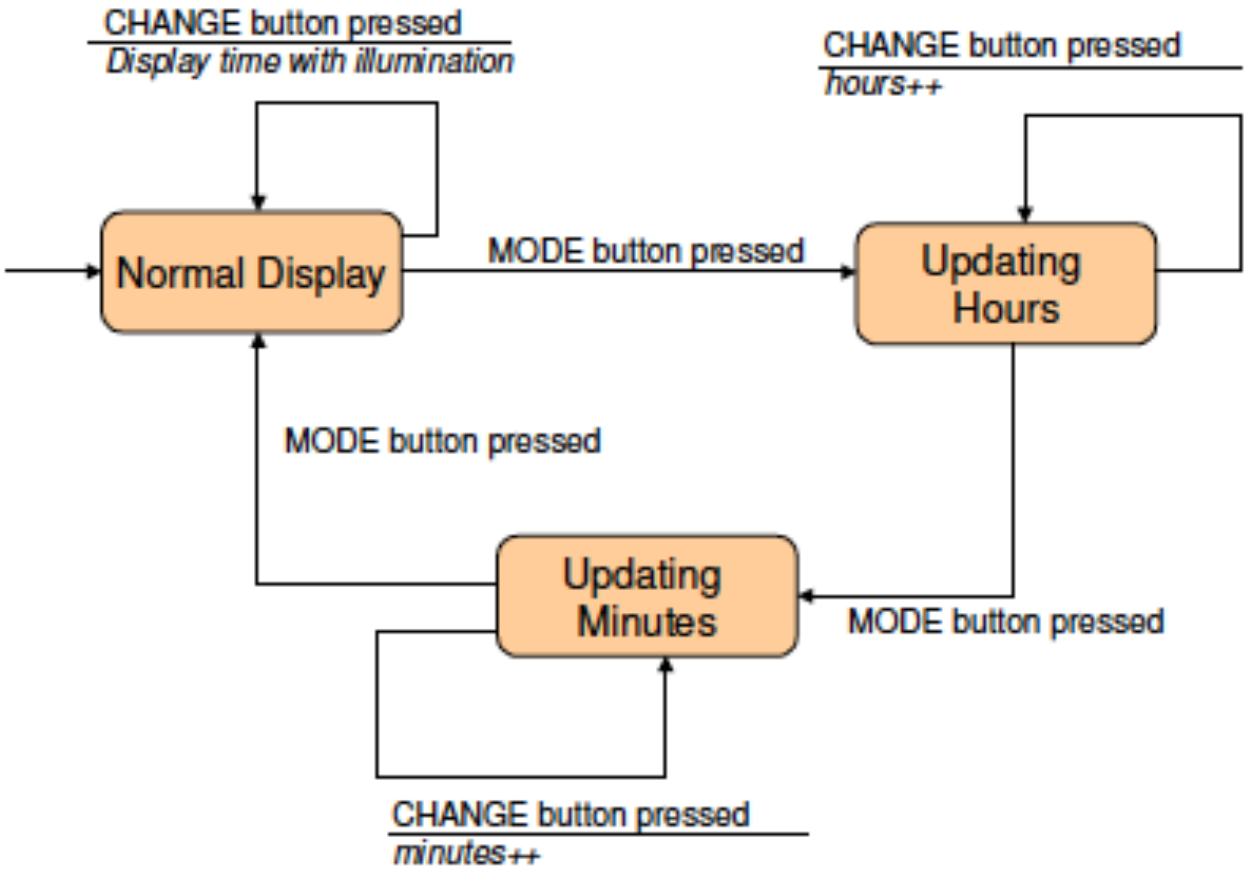


Diagramma di stati per il comportamento di un orologio

Esercizio

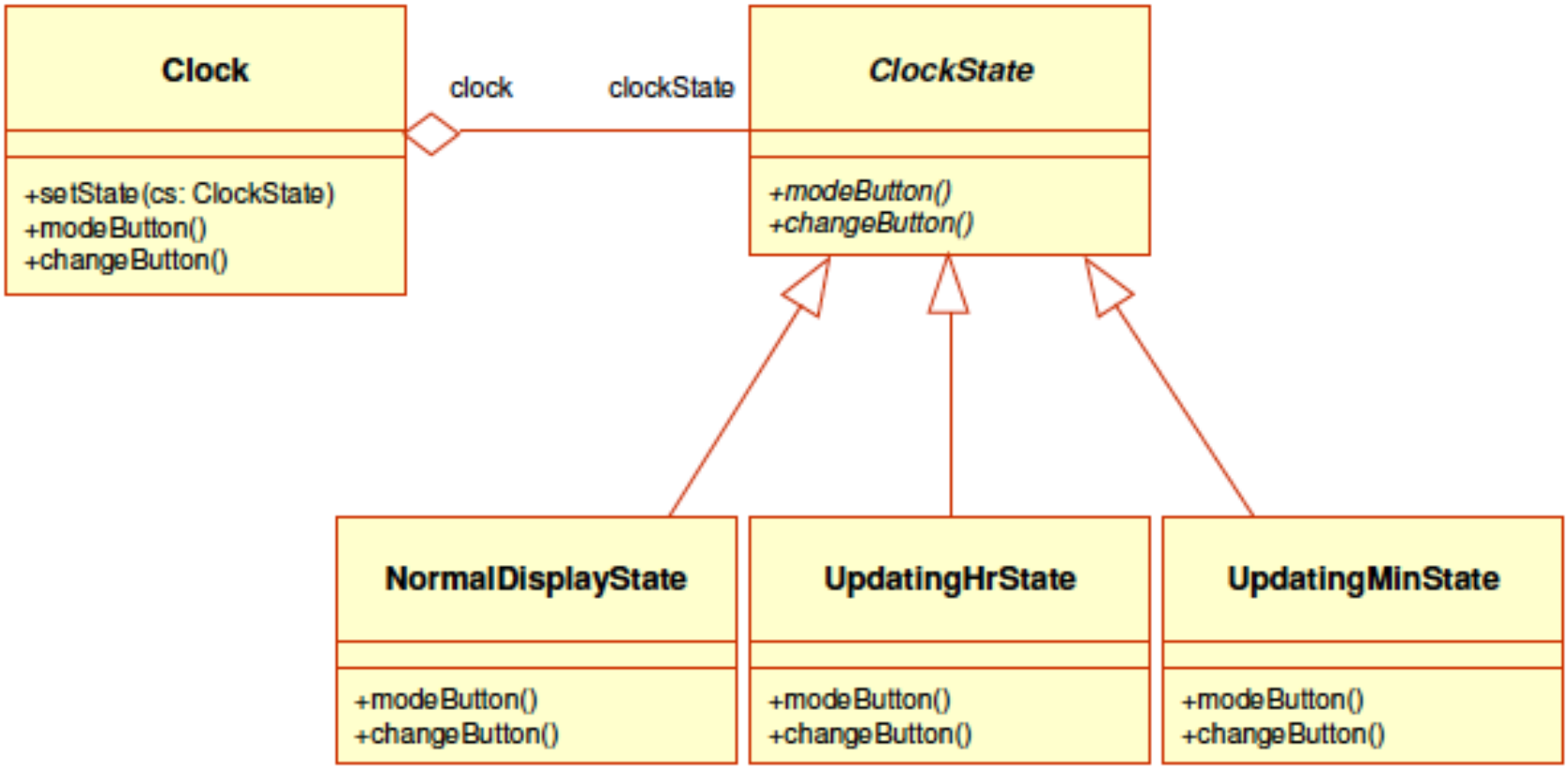


Diagramma delle classi per il comportamento di un orologio

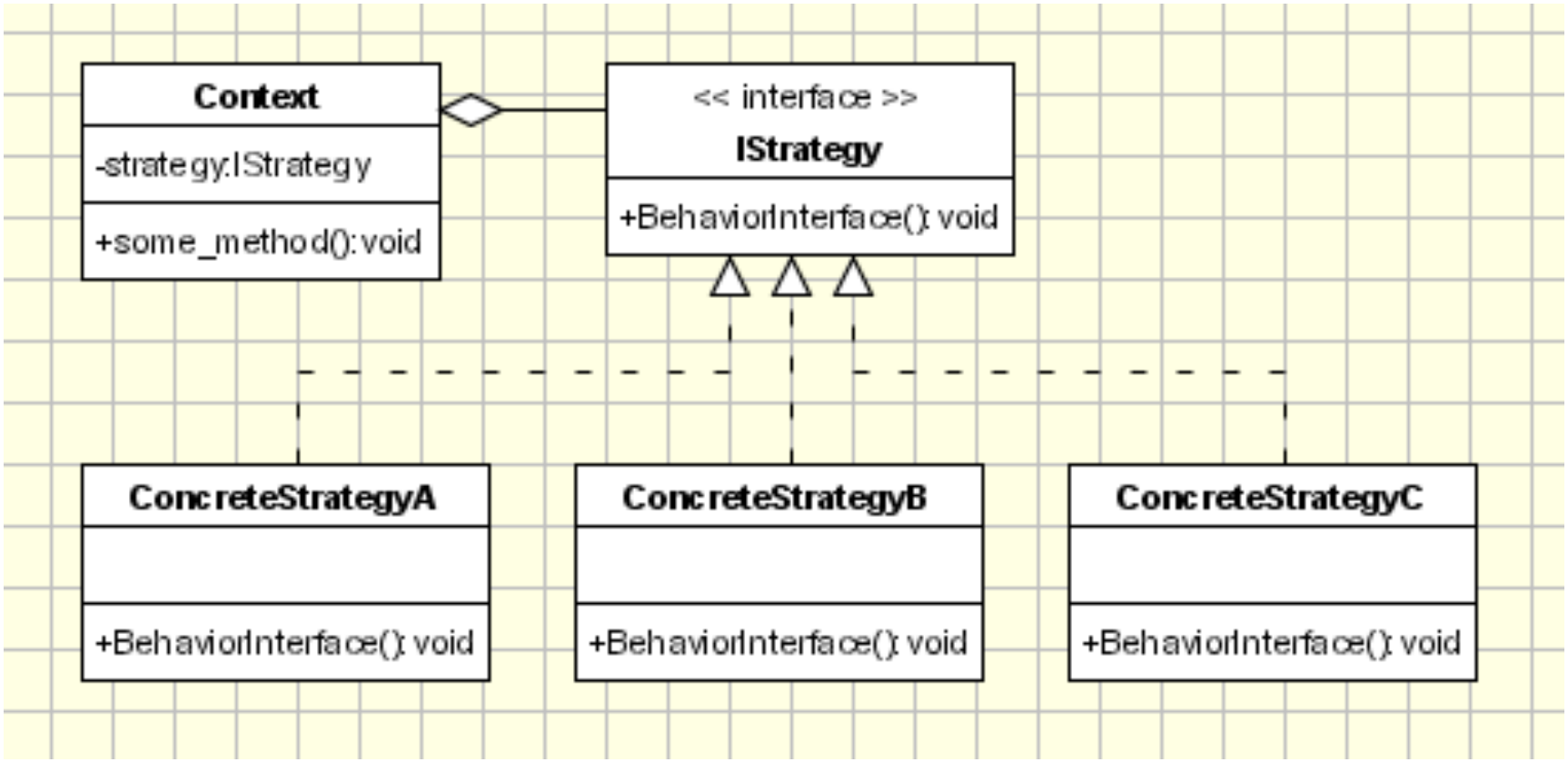
Strategy

- Scopo
  - *Definire una famiglia di algoritmi, incapsularli e renderli intercambiabili. L'algoritmo cambia indipendentemente dai client che lo usano.*
- Anche conosciuto come
  - *Policy*
- Motivazione
  - necessità di modificare dinamicamente gli algoritmi utilizzati da un'applicazione
    - e.g., visite in una struttura ad albero, possibilità di selezionare a tempo di esecuzione una tra le visite

Strategy

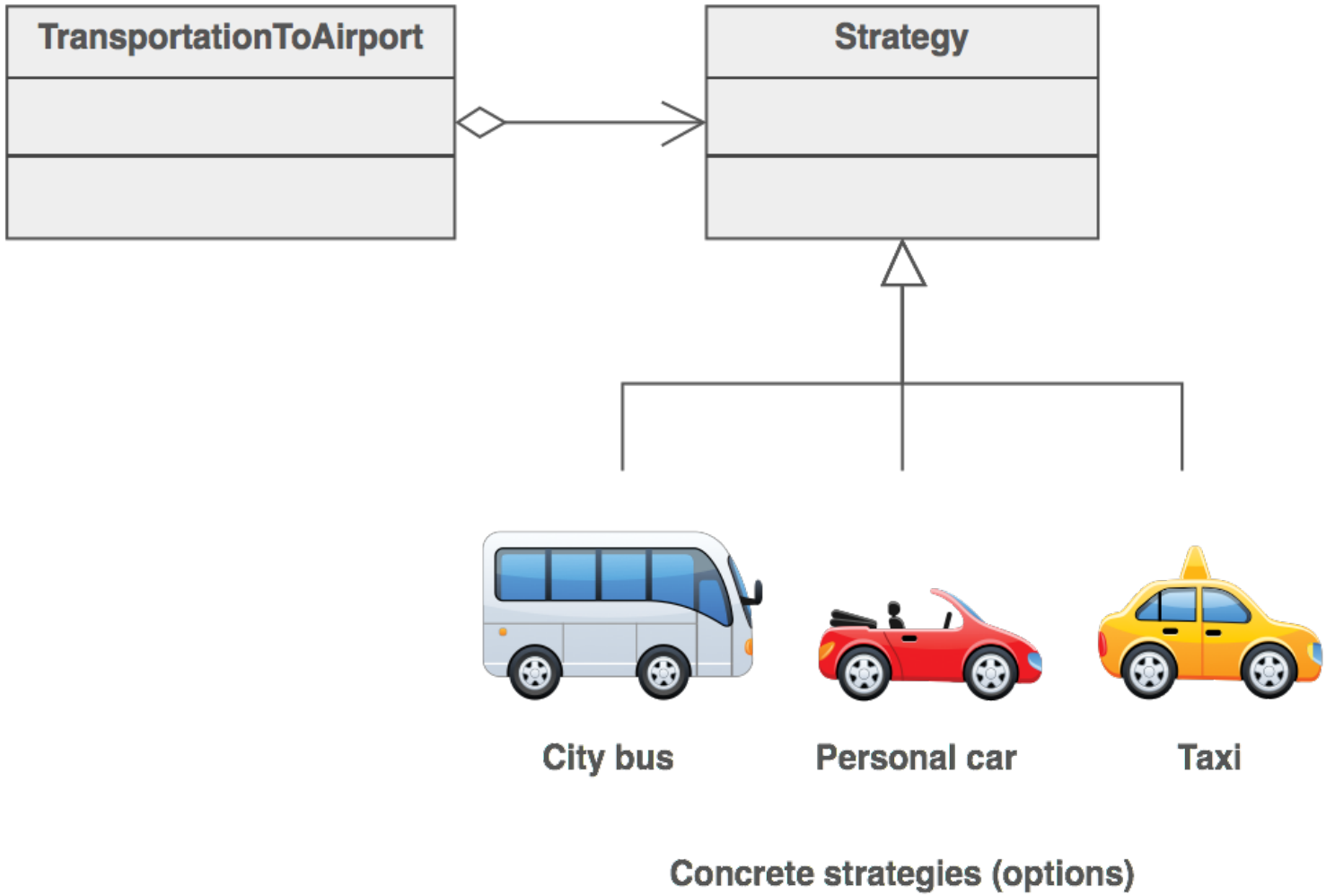
- Applicabilità
  - classi correlate differiscono solo per il loro comportamento
  - abbiamo bisogno di varianti di un algoritmo
  - un algoritmo usa dati che i client non dovrebbero conoscere
  - una classe definisce diversi comportamenti

Strategy - Struttura



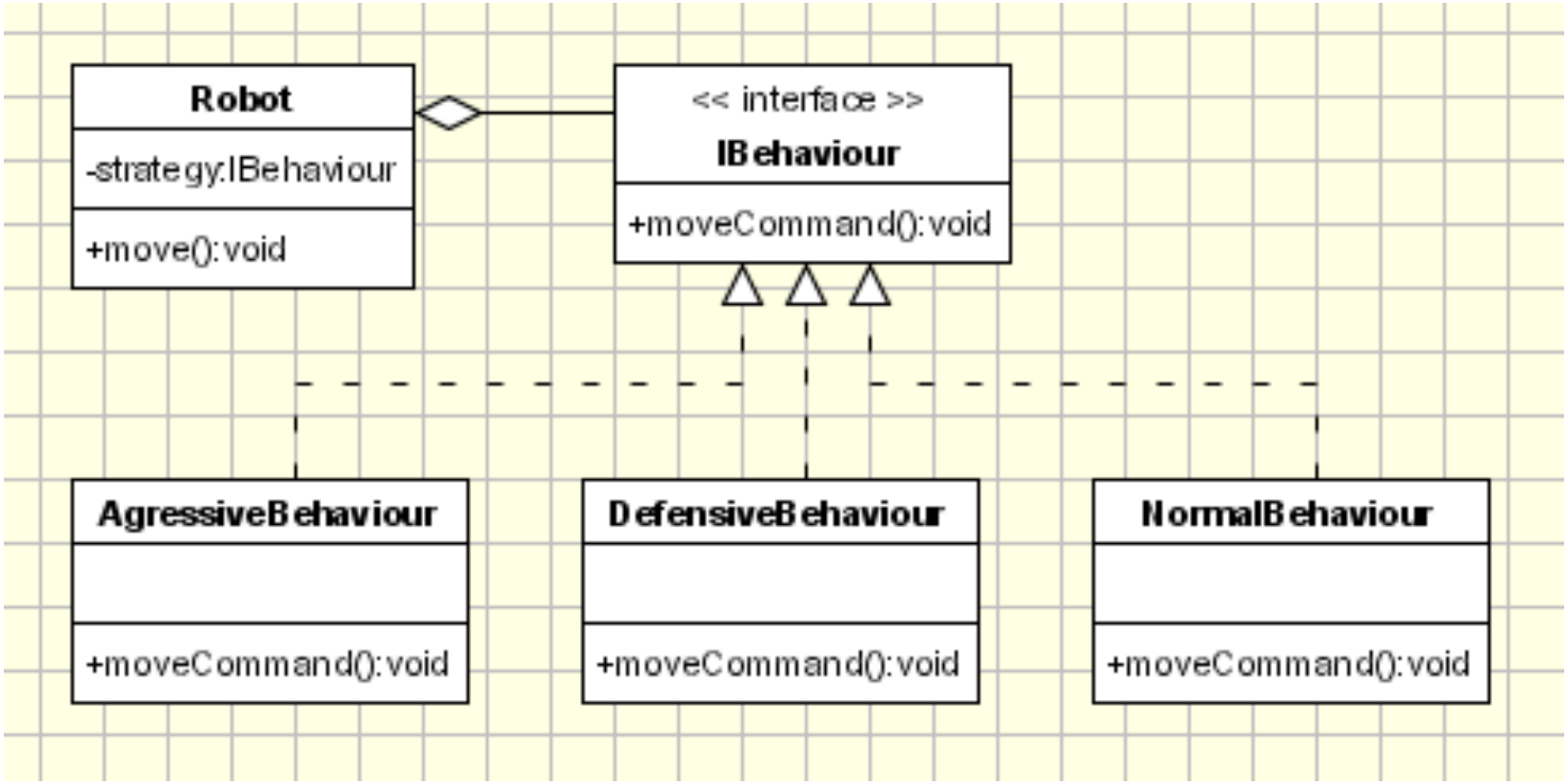
Struttura del pattern `Strategy`

Strategy



Esemplificazione del pattern `Strategy`. Trasporti in Aeroporto.

Esempio Strategy

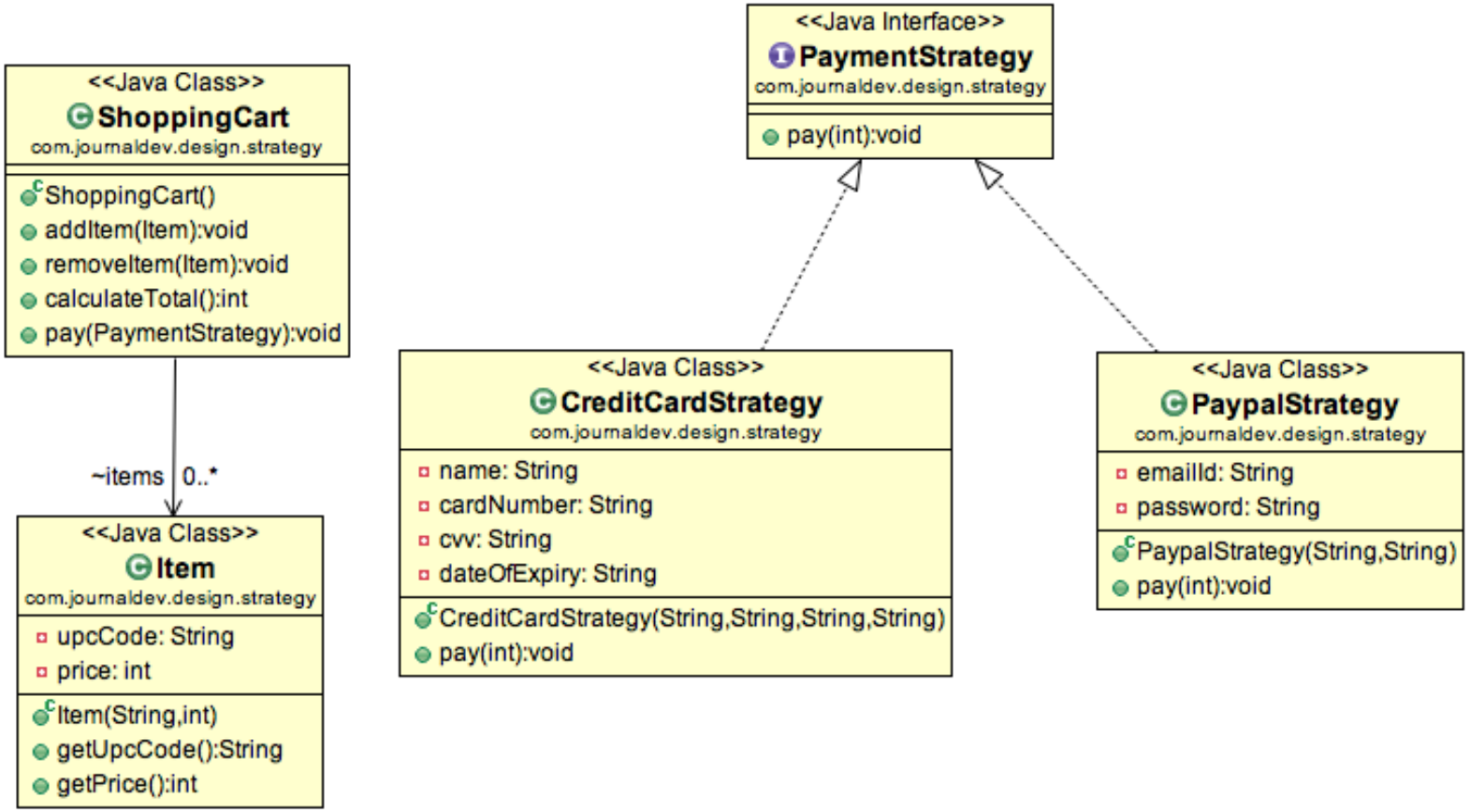


Esempio di implementazione del pattern `Strategy`. Robot.

Codice di riferimento

`Robot (directory)`

Esercizio



Esercizio pattern `Strategy`. Shopping cart.

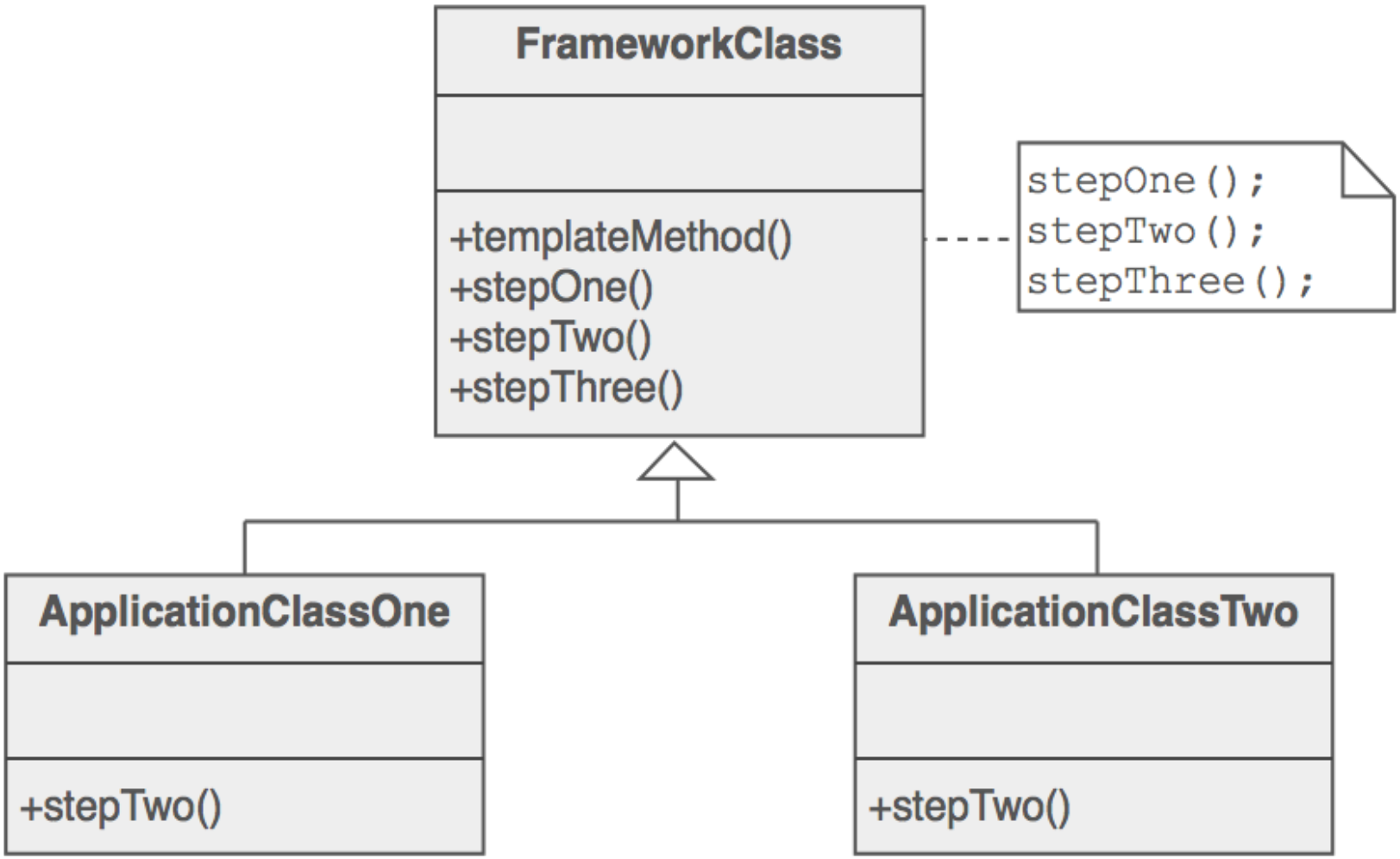
Template method

- Scopo
  - *Definire lo scheletro di un algoritmo in un'operazione, rinviando alcuni passi alle sottoclassi client. Consente alle sottoclassi di ridefinire alcuni passi senza cambiare la struttura dell'algoritmo.*
- Motivazione
  - necessità di modificare dinamicamente gli algoritmi utilizzati da un'applicazione
    - e.g., visite in una struttura ad albero, possibilità di selezionare a tempo di esecuzione una tra le visite

Template Method

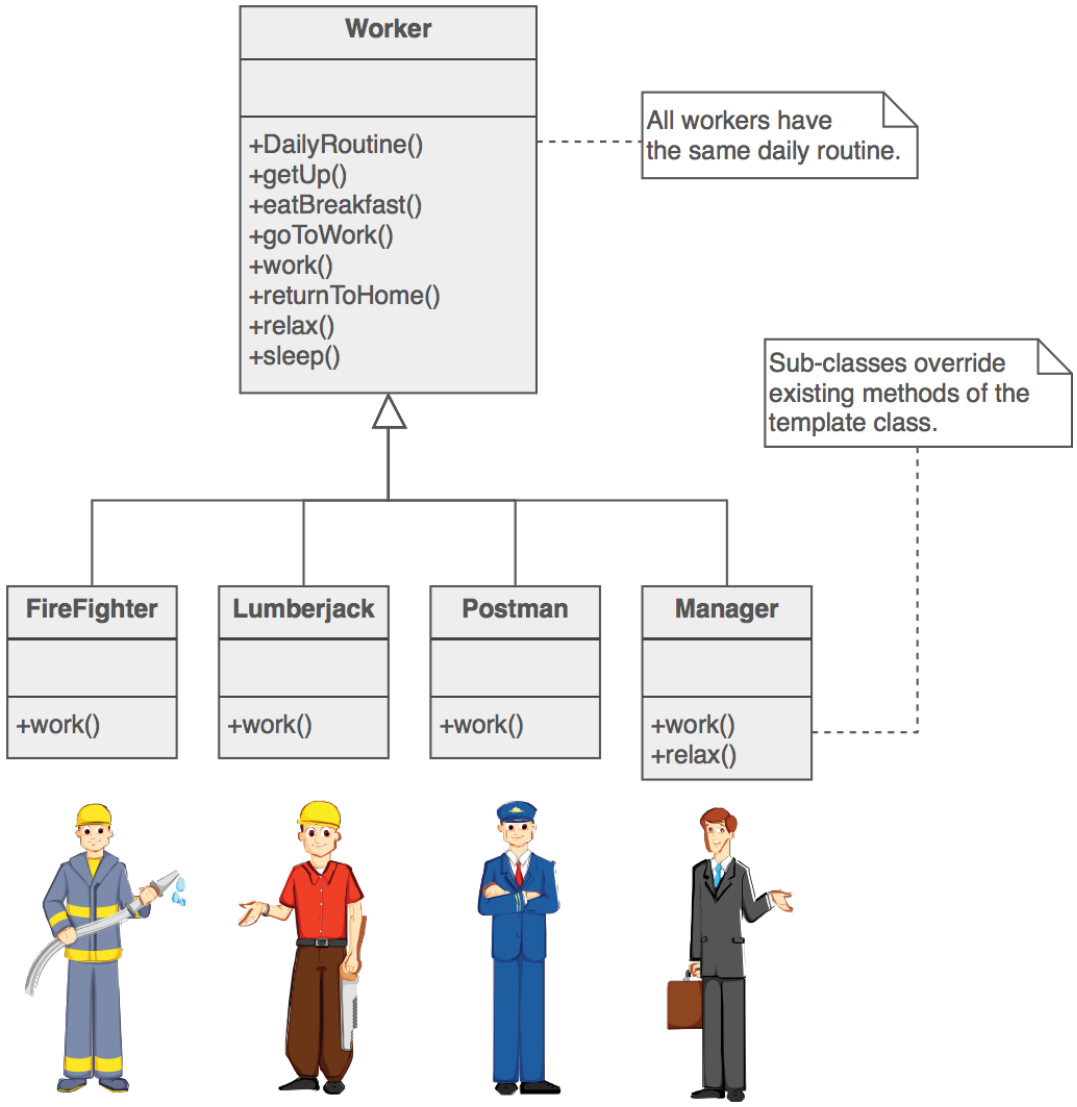
- Applicabilità
  - quando si vuole implementare la parte invariante di un algoritmo una volta sola e lasciare che le sottoclassi implementino il comportamento che può variare
  - quando il comportamento comune di più classi può essere fattorizzato in una classe a parte per evitare di scrivere più volte lo stesso codice
  - per avere modo di controllare come le *sottoclassi ereditano dalla superclasse*, facendo in modo che i metodi template chiamino dei metodi "gancio" (hook) e impostarli come unici metodi sovrascrivibili

Template Method - Struttura



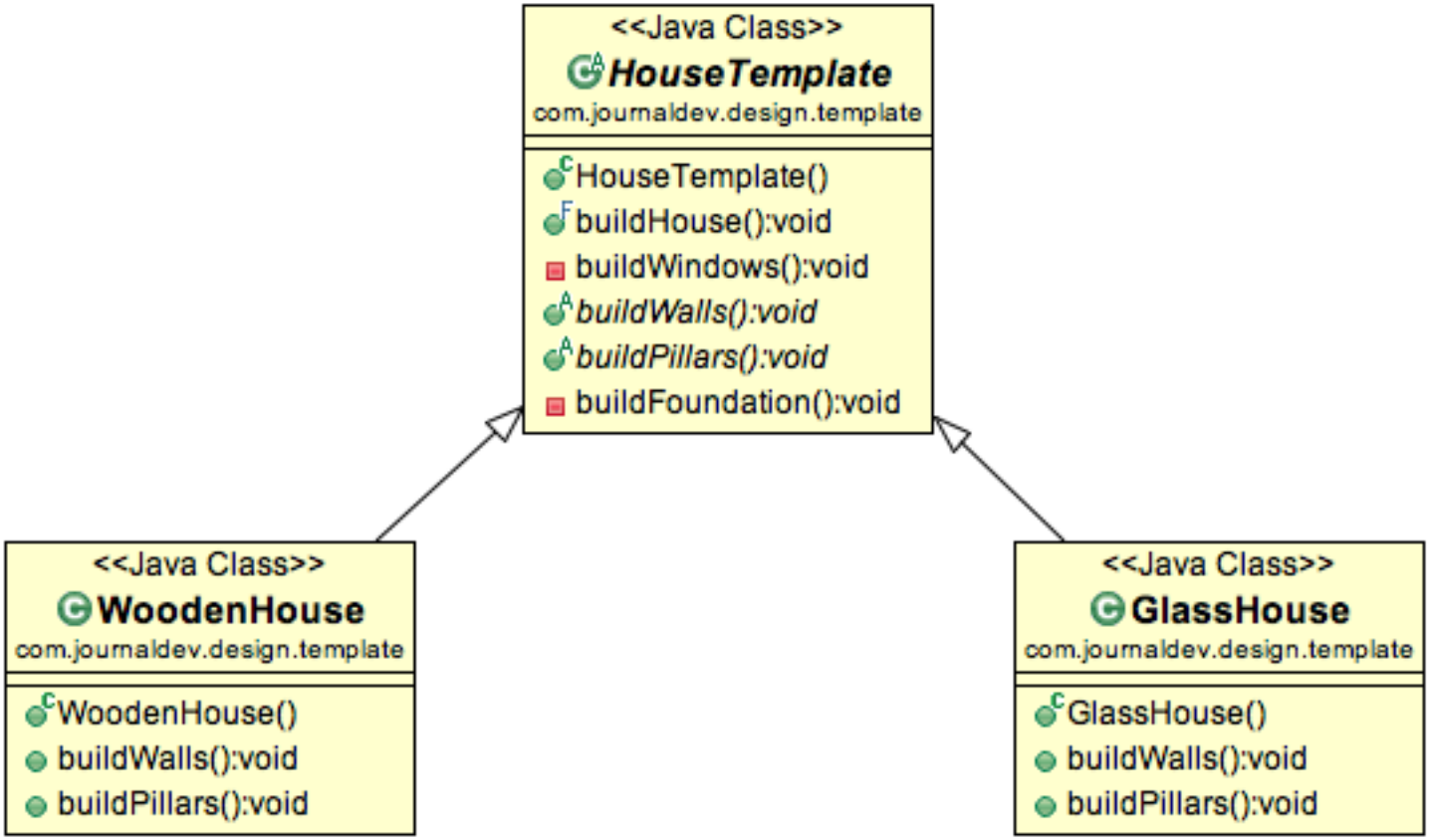
Struttura del pattern `Template Method`

Template Method



Esemplificazione del pattern `Template Method`

Esempio Template Method

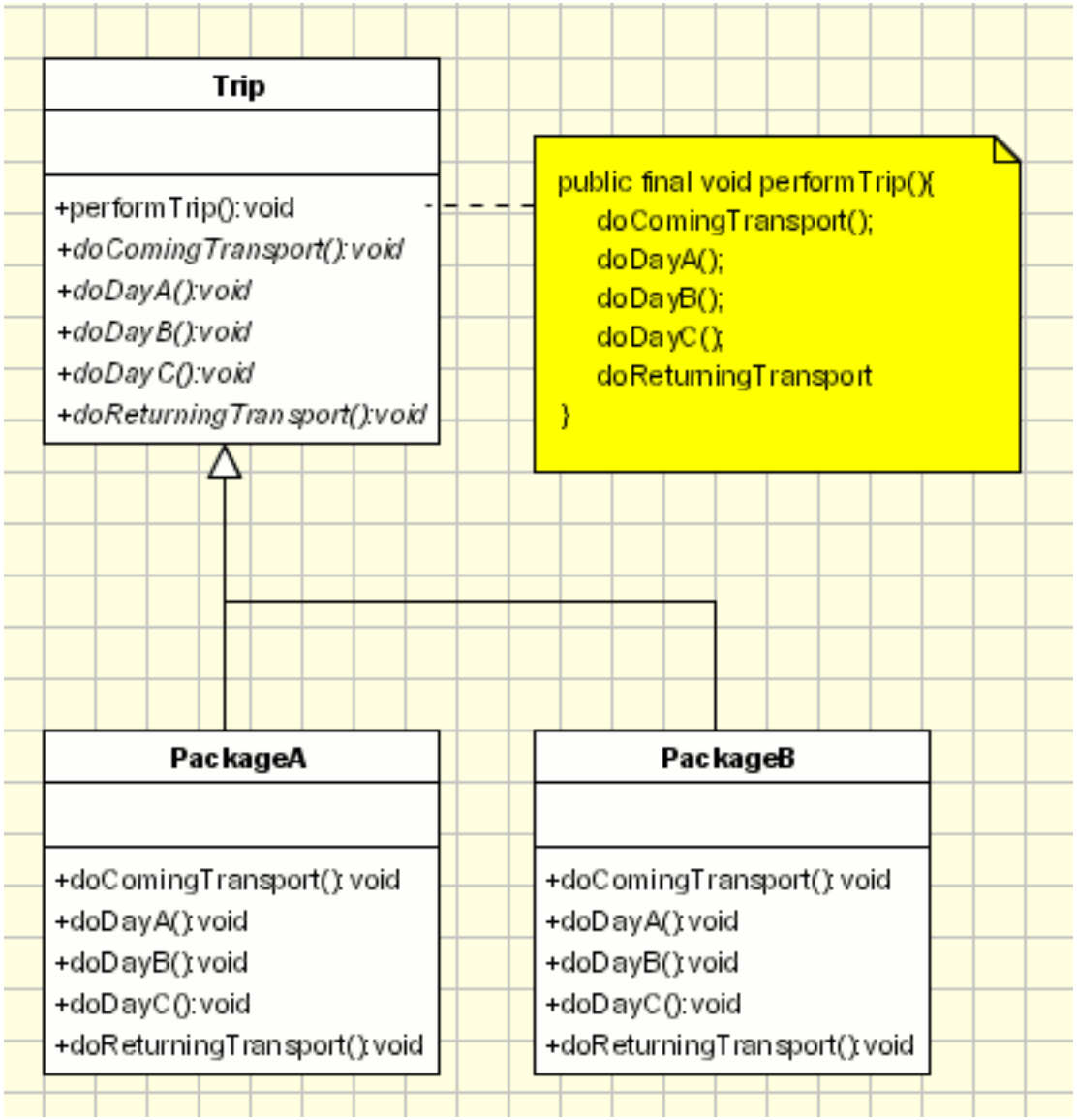


Esempio di implementazione del pattern `Template Method`. Casa.

Codice di riferimento

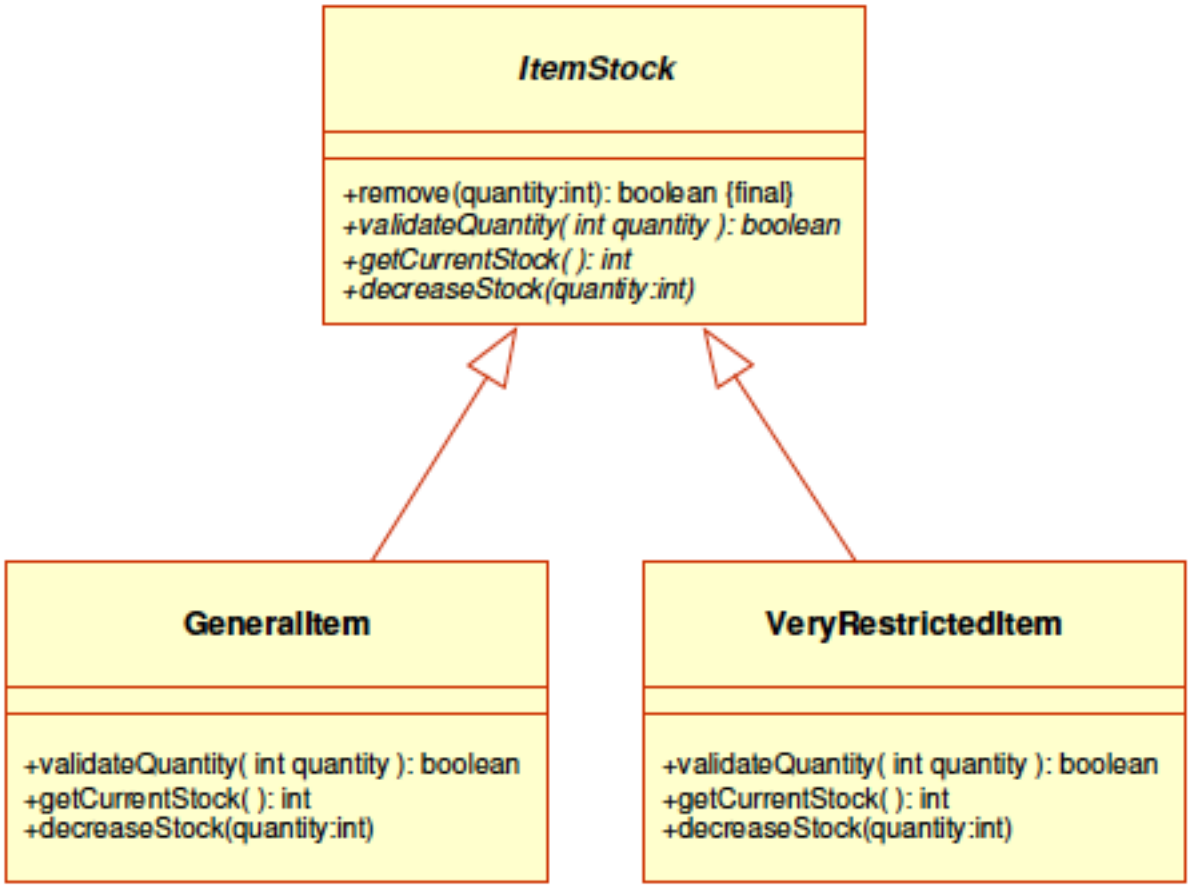
`Casa (directory)`

Esercizi



Esempio di implementazione del pattern `Template Method`. Agenzia di viaggi.

Esercizi



Esempio di implementazione del pattern `Template Method`. Magazzino.

Considerazioni

- `Template Method` è usato in (tutti i metodi non astratti)
  - `java.io.InputStream`
  - `java.io.OutputStream`
  - `java.io.Reader`
  - `java.io.Writer`
  - `java.util.AbstractList`
  - `java.util.AbstractSet`
  - `java.util.AbstractMap`

Visitor

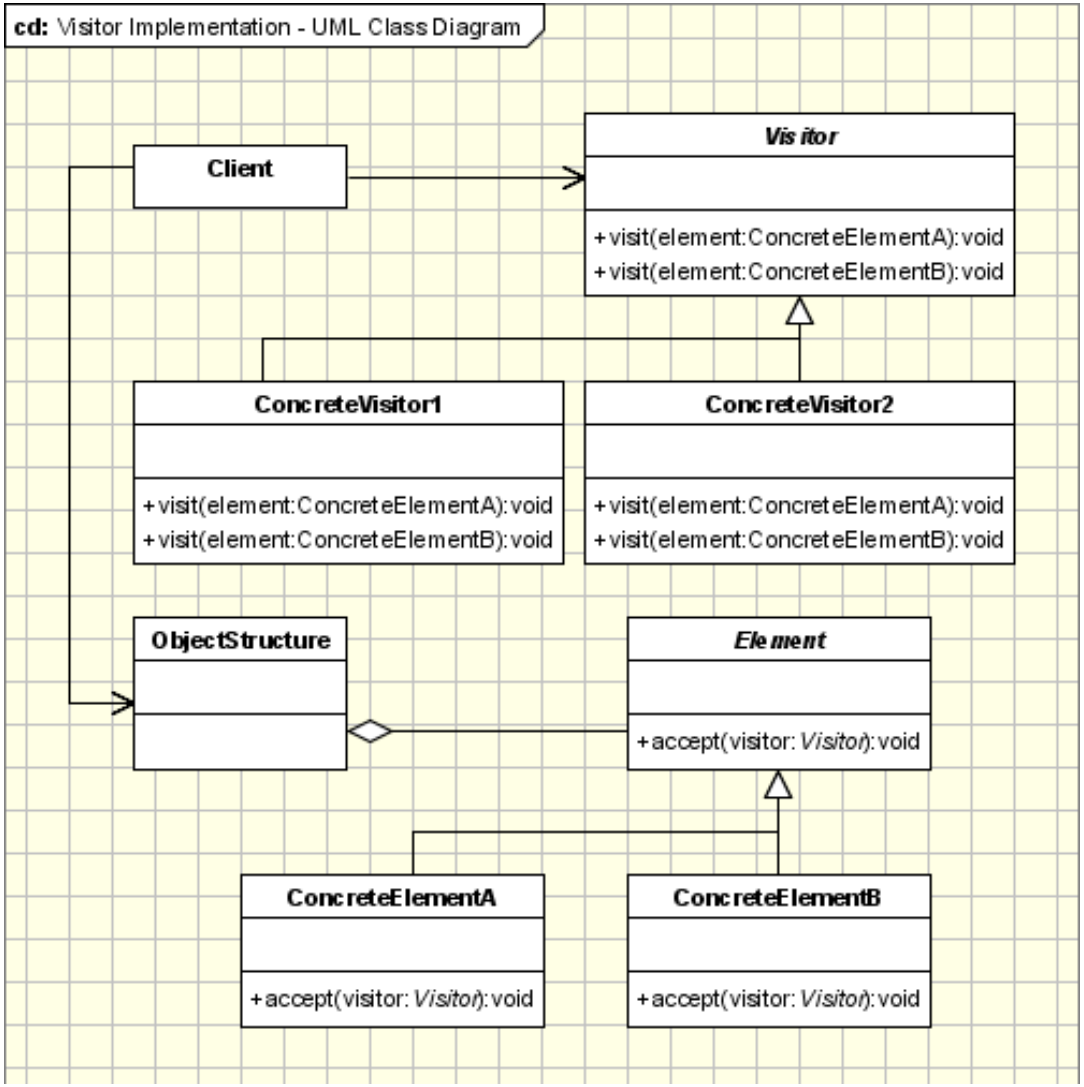
- Scopo
  - *Rappresentare un'operazione da eseguire sugli elementi di una struttura oggetto. Permette di definire una nuova operazione senza cambiare le classi degli elementi su cui opera.*

- Motivazione
  - separare un algoritmo dalla struttura di oggetti composti a cui è applicato
    - aggiungere nuove operazioni e comportamenti senza dover modificare la struttura stessa
  - in collezioni contenere oggetti di tipo diverso
    - operazioni eseguite su tutti gli elementi di raccolta *senza conoscere il tipo*

Visitor

- Applicabilità
  - una struttura di oggetti è costituita da molte classi con interfacce diverse ed è necessario che l'algoritmo esegua su ogni oggetto un'operazione differente a seconda dalla classe concreta dell'oggetto stesso
  - è necessario eseguire svariate operazioni indipendenti e non relazionate tra loro sugli oggetti di una struttura composta, ma non si vuole sovraccaricare le interfacce delle loro classi
    - riunendo le operazioni correlate in ogni `Visitor` è possibile inserirle nei programmi solo dove necessario
  - le classi che costituiscono la struttura composta sono raramente suscettibili di modifica, ma è necessario aggiungere spesso operazioni sui rispettivi oggetti

Visitor – Struttura



Struttura del pattern `Visitor`

Visitor

Cab company dispatcher

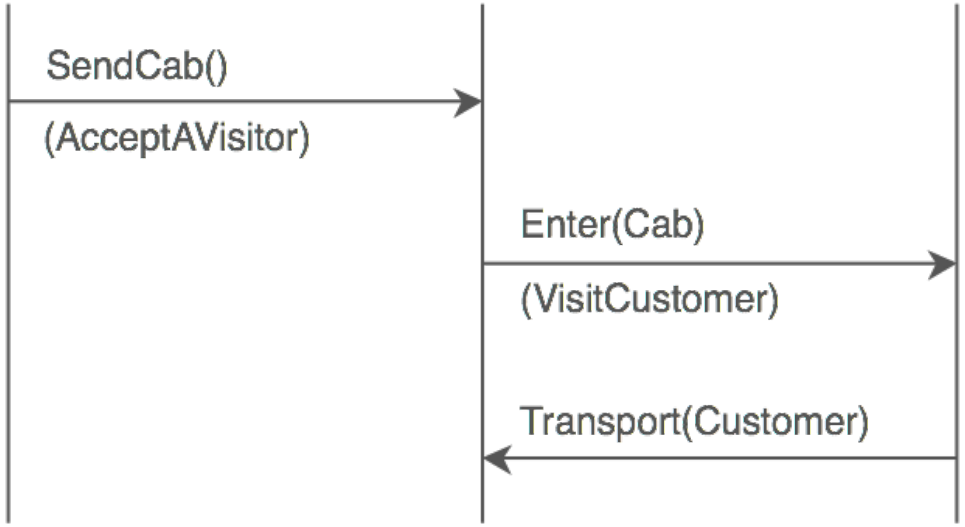
Object structure is list of Customers

Customer

Concrete element of Customers list

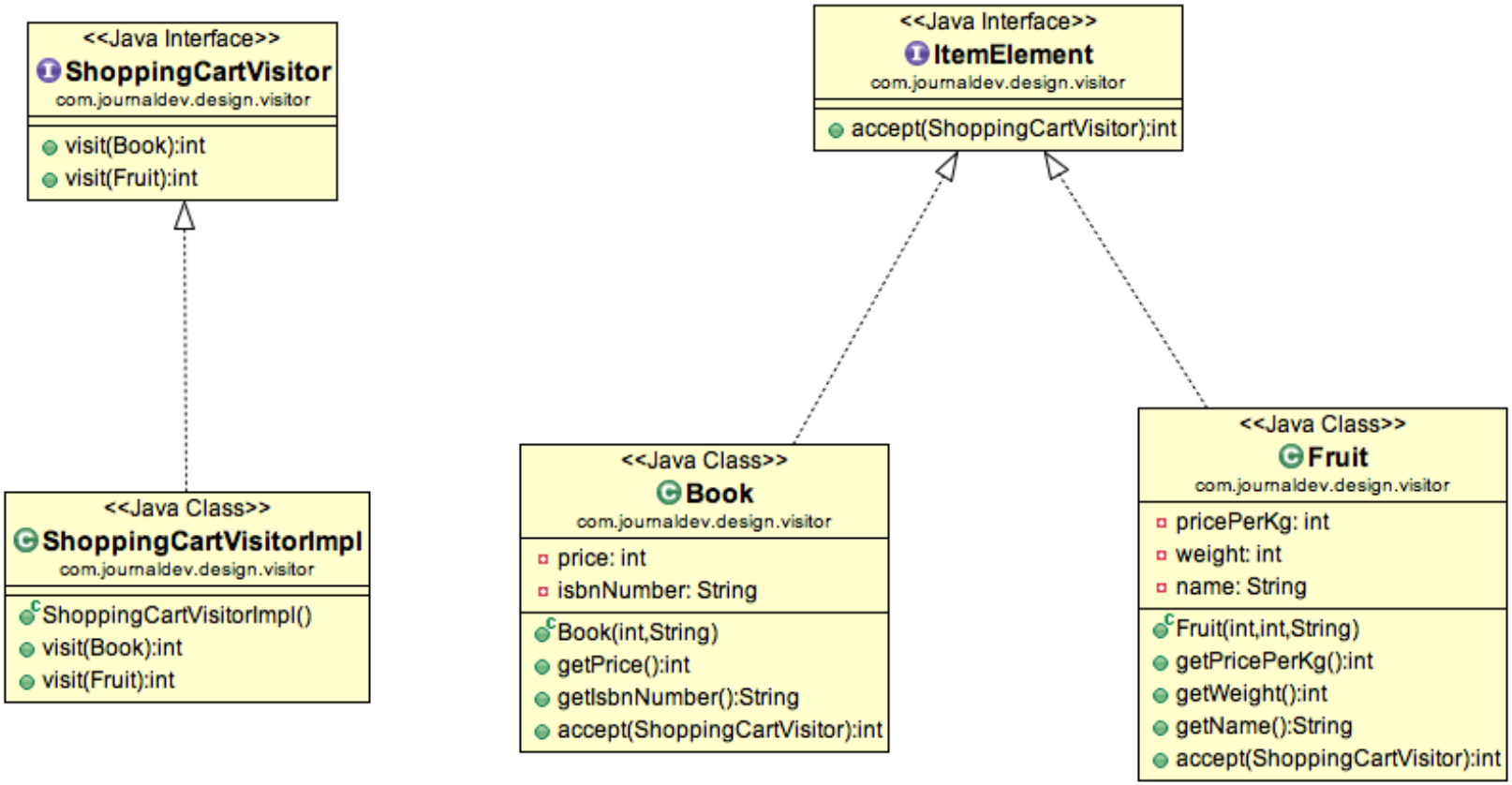
Taxi

Visitor



Esemplificazione del pattern `Visitor`. Compagnia Taxi.

Esempio Visitor



Esempio di implementazione del pattern `Visitor`. Carrello della spesa.

Codice di riferimento

`Shopping_Cart (directory)`