UNIVERSITÀ
DEGLI STUDI
FIRENZE

Scuola di Ingegneria
Corso di laurea triennale in
Ingegneria informatica

Elaborato per Ingegneria del Software

10 luglio 2020

Bellini Daniele

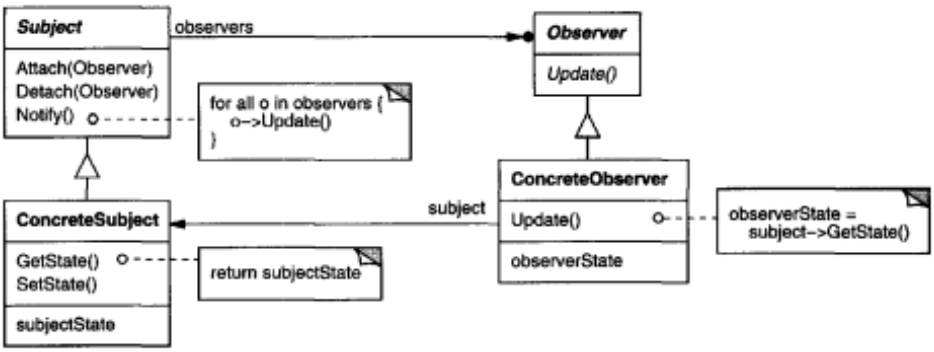
6359592

Scenario di applicazione

Lo scenario che ho scelto di esaminare è quello relativo all'interazione tra un cliente e una piattaforma al momento della scelta dei prodotti da inserire all'interno del proprio ordine.
Per la realizzazione ho utilizzato tre pattern: *Observer, Visitor e State*. Sono pattern **comportamentali** (Behavioural pattern), ovvero definiscono le modalità di comunicazione tra classi ed oggetti.

Breve excursus sui pattern utilizzati

Il design pattern ***Observer*** definisce un collegamento tra oggetti in modo tale che se lo stato di un oggetto viene modificato, tutti gli oggetti che ne dipendono vengono immediatamente aggiornati. Questo, benché permetta la comunicazione tra entità, introduce un leggero accoppiamento tra le parti. Prevede l'esistenza di due attori: un singolo oggetto - conosciuto come Subject - che ha la libertà di effettuare cambiamenti del proprio stato ed una collezione di oggetti - conosciuti come Observer - che adattano il proprio stato in base alla notifica dell'avvenuto cambiamento.
Ogni observer viene settato a run-time e può trovarsi all'interno di una classe che eredita da una classe base oppure implementare una comune interfaccia. La loro effettiva funzionalità ed il loro uso dei dati non deve essere conosciuta dal Subject.
In un'ottica di interfacce potremmo definire il Subject come Publisher, associato ad una collezione di Subscribers, ovvero gli Observers, i quali, una volta eseguito il metodo *attach(Observer)* al Subject, diventano dipendenti dallo stesso e vengono automaticamente aggiornati appena registrano un cambiamento.



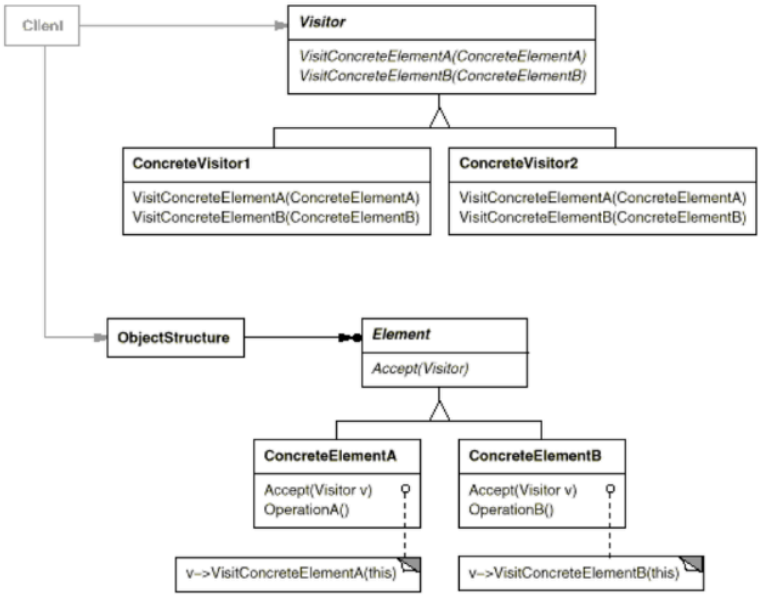
Ricapitolando, i vantaggi noti sono:

- il pattern costituisce un ottimo esempio di applicazione del noto principio "*Fare in modo che l'accoppiamento tra oggetti sia il meno stretto possibile*";
- consente di inviare dati ad altri oggetti in modo efficace senza alcuna modifica nelle classi Subject o Observer;
- Subject e Observer sono indipendenti e la modifica di uno di questi componenti non richiede la modifica dell'altro;
- gli Observers possono essere aggiunti o rimossi in qualsiasi momento.

mentre gli svantaggi che introduce questo pattern sono:

- la classe di Java *Observable* impone l'utilizzo dell'ereditarietà rispetto alla programmazione su interfaccia;
- se non usato con attenzione, tale pattern può aggiungere complessità non necessaria. Occorre tenere presente, infatti, che è delegato al Client l'avvio della notifica di aggiornamento degli observer e che ogni aggiornamento notificherà tutti gli observer; pertanto occorre stare attenti ad invocare una volta sola per aggiornamento per evitare la duplicazione degli aggiornamenti inviati.
- l'ordine delle notifiche di Observer è inaffidabile

Il design pattern ***Visitor*** è un modello di progettazione che permette la separazione di un insieme di dati strutturati dalle operazioni che possono essere eseguite su di esso. Ciò favorisce l'accoppiamento lento e consente l'aggiunta di ulteriori azioni senza modifica delle classi.
Viene infatti creato un modello di dati con funzionalità interne limitate ed un insieme di visitatori che esegue operazioni sui dati; in particolare, consente a ciascuno degli elementi di una struttura di dati di essere visitato a turno senza conoscere in anticipo i dettagli della stessa.
Le classi della struttura dati vengono inizialmente create con l'inclusione di un metodo *accept(Visitor):* questo metodo esegue una chiamata al visitatore, passando se stesso come parametro al metodo *visit(Visitable)* del visitatore, il quale diviene in grado di eseguire operazioni sull'oggetto. Inoltre, l'oggetto visitatore può mantenere lo stato tra le chiamate ai singoli oggetti.



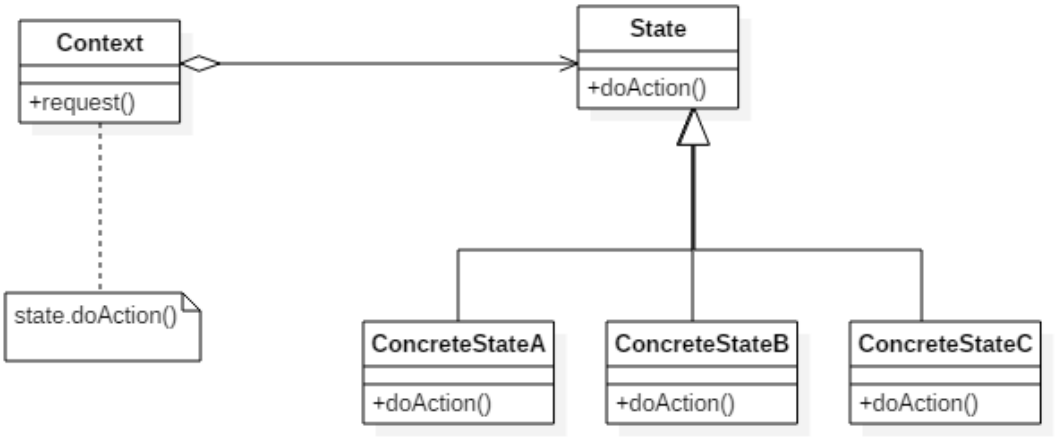
Ricapitolando, i vantaggi noti sono:

- se la logica dell'operazione cambia, sarà sufficiente apportare modifiche solo nell'implementazione del visitatore anziché in tutte le classi oggetto;
- l'aggiunta di un nuovo elemento al sistema è semplice, richiederà modifiche solo nell'interfaccia e nell'implementazione del visitatore e le classi dei ConcreteVisitor esistenti non saranno interessate.

mentre gli svantaggi che introduce questo pattern sono:

- è necessario conoscere il tipo di ritorno dei metodi *visit()* al momento della progettazione, altrimenti si deve modificare l'interfaccia e tutte le sue implementazioni;
- i ConcreteElement devono implementare una particolare interfaccia per essere visitati ed occorre prevedere nell'interfaccia e nell'implementazione del Visitor il metodo di visita per ciascun tipo.

Il design pattern ***State*** è un modello di progettazione che consente ad una classe di cambiare operato in base al proprio stato.
È molto simile al modello strategico ma viene utilizzato per incapsulare comportamenti variabili per lo stesso oggetto in base al suo stato interno. L'oggetto sembrerà cambiare la sua classe.
L'uso corretto del modello di stato può essere un modo più pulito per un oggetto di modificare il suo comportamento in fase di esecuzione senza ricorrere a grandi istruzioni condizionali monolitiche, migliorandone la manutenibilità.
Nel modello di stato, vengono creati oggetti che rappresentano vari stati ed un oggetto di contesto, il cui comportamento varia al variare del suo oggetto di stato. La classe Context è una singola interfaccia con il mondo esterno, mentre una classe di stato è una classe di base astratta per tutti gli stati concreti. ConcreteStateA, ConcreteStateB, ConcreteStateC, ... rappresentano i diversi "stati" della macchina definiti come classi derivate della classe base. I comportamenti specifici dello stato sono definiti nelle classi derivate dallo stato base (classe State).
L'obiettivo è mantenere un puntatore allo "stato" corrente nella classe "contesto", quindi per modificare lo stato della macchina, è necessario modificare il puntatore allo "stato" corrente.



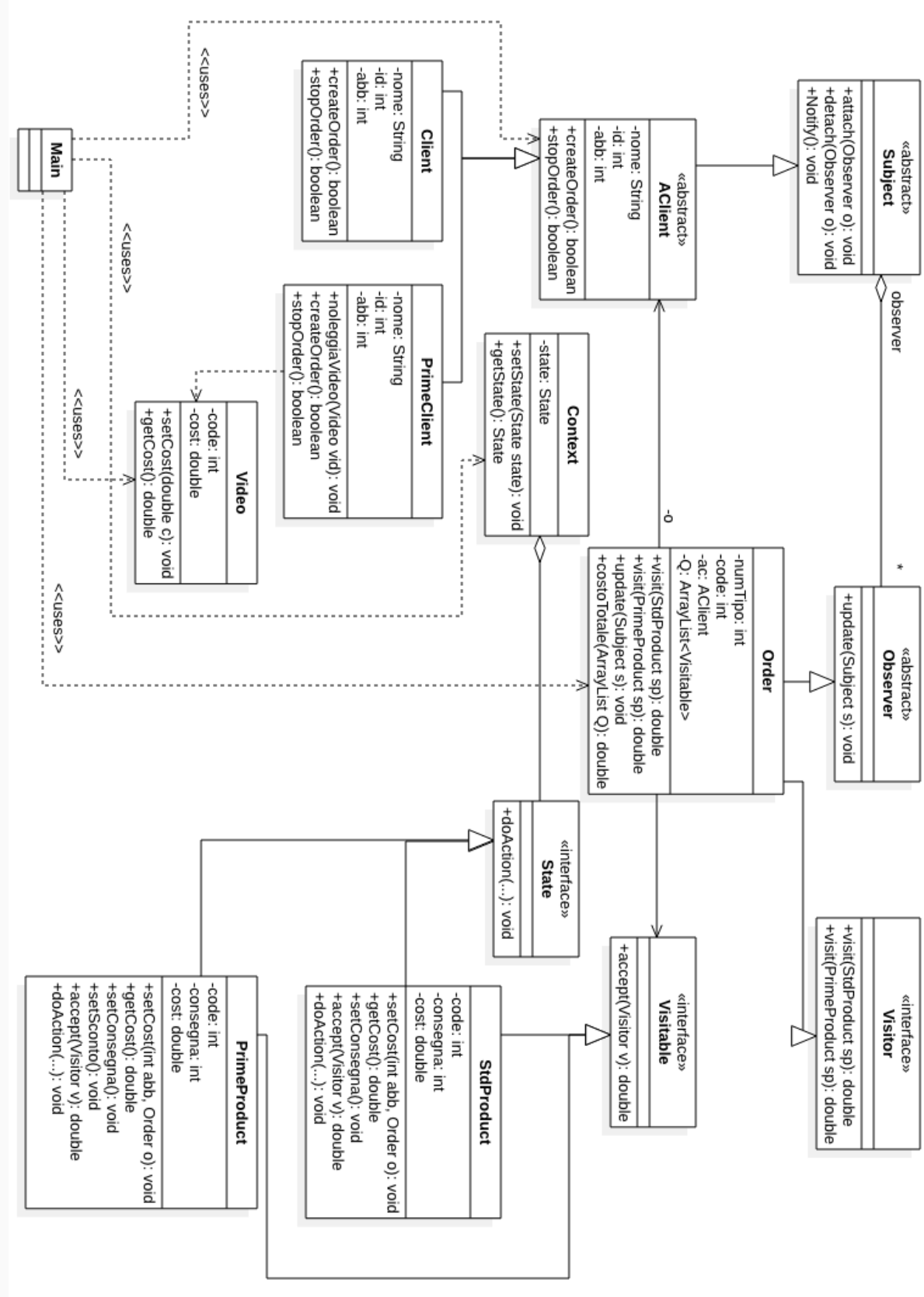
Ricapitolando, i vantaggi noti sono:

- maggiore modularità, il comportamento dell'intero sistema è la composizione dei comportamenti dei singoli stati;
- maggiore elasticità, infatti si possono aggiungere nuovi stati senza modificare pesantemente il codice;
- maggiore flessibilità, è possibile modificare il comportamento di uno stato senza alterare il sistema.

mentre lo svantaggio principale che introduce questo pattern è:

- l'incremento del numero di classi.

Class Diagram



AClient rappresenta il **ConcreteSubject** ed è anche la classe astratta di Client e di PrimeClient, mentre **Order** gioca il ruolo sia di **ConcreteObserver** sia di **ConcreteVisitor**.

Nel momento in cui il cliente dichiara che ha terminato la spesa, ovvero quando diviene true il metodo *stopOrder()* (altro non è che un'azione sequenziale al termine dei vari *add()* all'ordine), viene invocata la *Notify()* e quindi l'*update(AClient)* nella classe Order.

```
if (c.getAbbonamento() == 0){
    if (c.createOrder()) {
        System.out.println("CLIENTE BASE "+c.getId()+", CODICE DELL'ORDINE "+oc.getCode() );
        int consegnaC = 0;

        StdProduct p1 = new StdProduct( code: 1425);
        p1.doAction(context,c.getAbbonamento(), oc);
        if (consegnaC < p1.getConsegna())
            consegnaC = p1.getConsegna();
        StdProduct p2 = new StdProduct( code: 2355);
        p2.doAction(context,c.getAbbonamento(), oc);
        if (consegnaC < p2.getConsegna())
            consegnaC = p2.getConsegna();
        StdProduct p3 = new StdProduct( code: 3325);
        p3.doAction(context,c.getAbbonamento(), oc);
        if (consegnaC < p3.getConsegna())
            consegnaC = p3.getConsegna();
        StdProduct p4 = new StdProduct( code: 4767);
        p4.doAction(context,c.getAbbonamento(), oc);
        if (consegnaC < p4.getConsegna())
            consegnaC = p4.getConsegna();
        StdProduct p5 = new StdProduct( code: 5836);
        p5.doAction(context,c.getAbbonamento(), oc);
        if (consegnaC < p5.getConsegna())
            consegnaC = p5.getConsegna();
        PrimeProduct perr = new PrimeProduct( code: 6134);
        perr.doAction(context,c.getAbbonamento(), oc);
        StdProduct p15 = new StdProduct( code: 15836);
        p15.doAction(context,c.getAbbonamento(), oc);
        if (consegnaC < p15.getConsegna())
            consegnaC = p15.getConsegna();
        PrimeProduct perr1 = new PrimeProduct( code: 64839);
        perr1.doAction(context, c.getAbbonamento(), oc);

       if (c.stopOrder()){
            double TOT = oc.costoTotale(oc.getQ());
            TOT = Math.floor(TOT * 100.0) / 100.0;
            System.out.println("- Il costo totale è: " + TOT + "€.");
            System.out.println("- Tempo di consegna: " + consegnaC);
        }
    }
}
c.detach(oc);
```

A questo punto, viene tenuto traccia del numero di tipologia di articoli da pagare, aggiungendo (+1) per ogni elemento che viene inserito nella lista **Q** (attributo della classe Order):

```
public void update(Subject s) {
    for (Visitable product:Q)
        numTipo++;
    if (numTipo == 1)
        System.out.println("\n- Esiste una sola tipologia di articoli da pagare.");
    else System.out.println("\n- Le tipologie di articoli da pagare sono: " + numTipo + ".");
}
```

e, attraverso un metodo aggiuntivo *costoTotale(Order.Q)*, viene restituito il prezzo che il cliente è tenuto a pagare. Tale metodo scansiona la lista di articoli ed invoca su ognuno di essi il metodo *accept(Order)* proprio del pattern Visitor, che a sua volta chiama il metodo *visit(Prodotto)* della classe Order. Nel nostro caso, è necessario definirne due versioni: una per gli articoli senza sconto, definiti dalla classe StdProduct, e una per gli articoli a cui è applicabile, definiti da PrimeProduct.

```
@Override
public double visit(StdProduct sp) { return sp.getCost(); }

public double visit(PrimeProduct pp) { return pp.getCost(); }

public double costoTotale( ArrayList<Visitable> Q){
    double totale = 0.0D;
    for (Visitable product: Q)
        totale += product.accept( visitor: this);
    return totale;
}
```

Considerando uno scenario in cui si hanno due clienti, uno Base, ovvero non ha diritto a godere di nessun vantaggio, ed uno Prime, il quale beneficia di sconti, velocità di consegna e possibilità di noleggiare video, allora si ha che l'output risulta essere:

-Output di un cliente Base

```
CLIENTE BASE 2681, CODICE DELL'ORDINE 3528

Prodotto Standard 1425:
15.24€. Inserito nell'ordine
Prodotto Standard 2355:
43.77€. Inserito nell'ordine
Prodotto Standard 3325:
19.44€. Inserito nell'ordine
Prodotto Standard 4767:
29.49€. Inserito nell'ordine
Prodotto Standard 5836:
14.8€. Inserito nell'ordine
Prodotto Prime 6134:
<<ERRORE: Cliente non abilitato all'offerta su questo prodotto, non inseribile nell'ordine>>
Prodotto Standard 15836:
32.68€. Inserito nell'ordine
Prodotto Prime 64839:
<<ERRORE: Cliente non abilitato all'offerta su questo prodotto, non inseribile nell'ordine>>
- Le tipologie di articoli da pagare sono: 6.
- Il costo totale è: 155.41€.
- Tempo di consegna: 9
```

-Output di un cliente Prime

```
CLIENTE PRIME 6392, CODICE DELL'ORDINE 3548

Prodotto Prime 6134:
22.48€ ---> 16.41€, sconto del 27%. Inserito nell'ordine
Prodotto Prime 7324:
38.24€ ---> 29.44€, sconto del 23%. Inserito nell'ordine
Prodotto Prime 8582:
8.44€ ---> 6.07€, sconto del 28%. Inserito nell'ordine
Prodotto Prime 9396:
20.84€ ---> 17.08€, sconto del 18%. Inserito nell'ordine
Prodotto Standard 11283:
7.46€. Inserito nell'ordine
Noleggio film al costo di 4.99
- Le tipologie di articoli da pagare sono: 5.
- Il costo totale è: 81.44€.
- Tempo di consegna: 7
```

Come si nota, il cliente che non beneficia di sconti non può comprare prodotti a sconto (PrimeProduct) mentre il cliente Prime può liberamente comprare prodotti su cui non è applicabile lo sconto (StdProduct).
La distinzione tra cliente Base e cliente Prime viene eseguita tramite l'uso di una variabile **int abbonamento**, settata a 0 e 1 rispettivamente.
Si è deciso di utilizzare il metodo *Math.floor(double)* del pacchetto Java.Math per limitare il numero di cifre decimali dopo la virgola.
Inoltre il tempo di consegna finale di tutto l'ordine segue il massimo tempo di consegna tra i prodotti all'interno dell'ordine stesso.

Il calcolo del costo per ogni prodotto viene eseguito tramite la funzione *setCost(int abbonamento, Order o)*, che nel caso di StdProduct è implementata come segue:

```
public void setCost(int abbonamento, Order o) {
    if (abbonamento == 0 || abbonamento == 1)
        cost = 5 + Math.random() * 45;
        cost = Math.floor(cost * 100.0) / 100.0;
        System.out.print(cost + "€.");
        o.getQ().add(this);
        System.out.print(" Inserito nell'ordine");
}
```

mentre nel caso di PrimeProduct:

```
public void setCost(int abbonamento, Order o) {
    if (abbonamento == 0){
        System.out.print("<<ERRORE: Cliente non abilitato all'offerta su questo prodotto, non inseribile nell'ordine>>");
    }
    else {
        double ct = 5 + Math.random() * 45;
        ct = Math.floor(ct * 100.0) / 100.0;
        cost = ct - ct * ((sconto) / 100.0);
        cost = Math.floor(cost * 100.0) / 100.0;
        System.out.print(ct + "€ ---> " + cost + "€, sconto del " + sconto + "%.");
        o.getQ().add(this);
        System.out.print(" Inserito nell'ordine");
    }
}
```

Nel calcolo del costo per prodotti Prime si controlla il tipo di abbonamento del cliente: se risulta essere di tipo Base, non è abilitato all'offerta sul prodotto scelto, quindi all'acquisto; se invece il cliente è di tipo Prime, si prosegue a calcolare il costo a cui si applica lo sconto.
Tale sconto viene calcolato in modo casuale in modo da averne uno compreso tra il 10% e il 30%.

```
public void setSconto(){
    sconto = 10 + ((int)(Math.random() * 20));
}
```

Il tempo di consegna è scandito dalla variabile **int consegna** nelle classi StdProduct e PrimeProduct. Come detto in precedenza, la tipologia Prime garantisce un valore minore di tale variabile. Infatti la funzione *setConsegna()* è definita come:

```
public void setConsegna(){
    consegna = 1 +((int)(Math.random() * 3));
}
```

mentre la funzione relativa alla tipologia Std:

```
public void setConsegna() {
    consegna = 5 + ((int) (Math.random() * 5));
}
```

Infine per i clienti Prime vi è la possibilità di noleggiare video. Tale operazione viene eseguita dalla funzione *noleggioVideo(Video vid)* nella classe PrimeClient dove l'oggetto vid può essere di tipo serie tv o di tipo film grazie alla variabile **int code**, settata a 0 o 1 rispettivamente.

```
public void noleggioVideo(Video vid){
    if (vid.getCode() == 0){
        System.out.println("\nNoleggio serie tv al costo di "+ vid.getCost());
    } else if(vid.getCode() == 1){
        System.out.println("\nNoleggio film al costo di "+ vid.getCost());

    }
}
```

Use Case Diagram