

---

ANNO ACCADEMICO 2023/2024

---

# Sviluppo Applicazioni Software

---

## Esercizi

Luna's Notes



**UNIVERSITÀ**  
**DI TORINO**



---

DIPARTIMENTO DI INFORMATICA

---



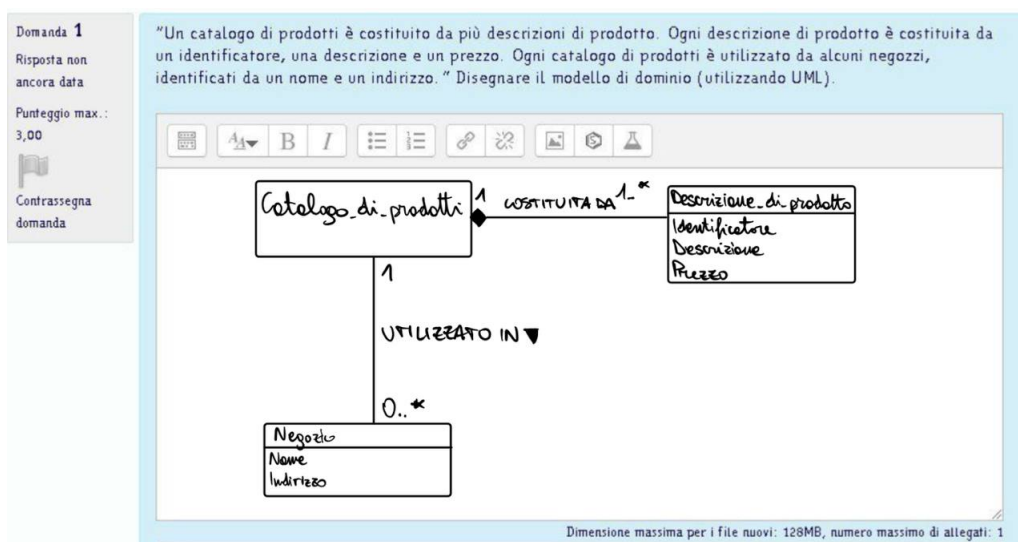
<b>CAPITOLO 1</b>	<b>PRIMO ESERCIZIO</b>	<b>PAGINA 2</b>
1.1	Modello di Dominio	2
1.2	SSD	2
1.3	Contratti	3
1.4	DSD	4
1.5	DCD	6
<b>CAPITOLO 2</b>	<b>SECONDO ESERCIZIO</b>	<b>PAGINA 8</b>
2.1	GRASP	8
2.2	GoF	10
	Decorator — 10 • Composite — 11 • Strategy — 12 • Visitor — 13 • State — 14 • Observer — 15 • Adapter — 16 • AbstractFactory e Singleton — 17	



# 1

## Primo esercizio

### 1.1 Modello di Dominio



### 1.2 SSD

#### Domanda 1.1

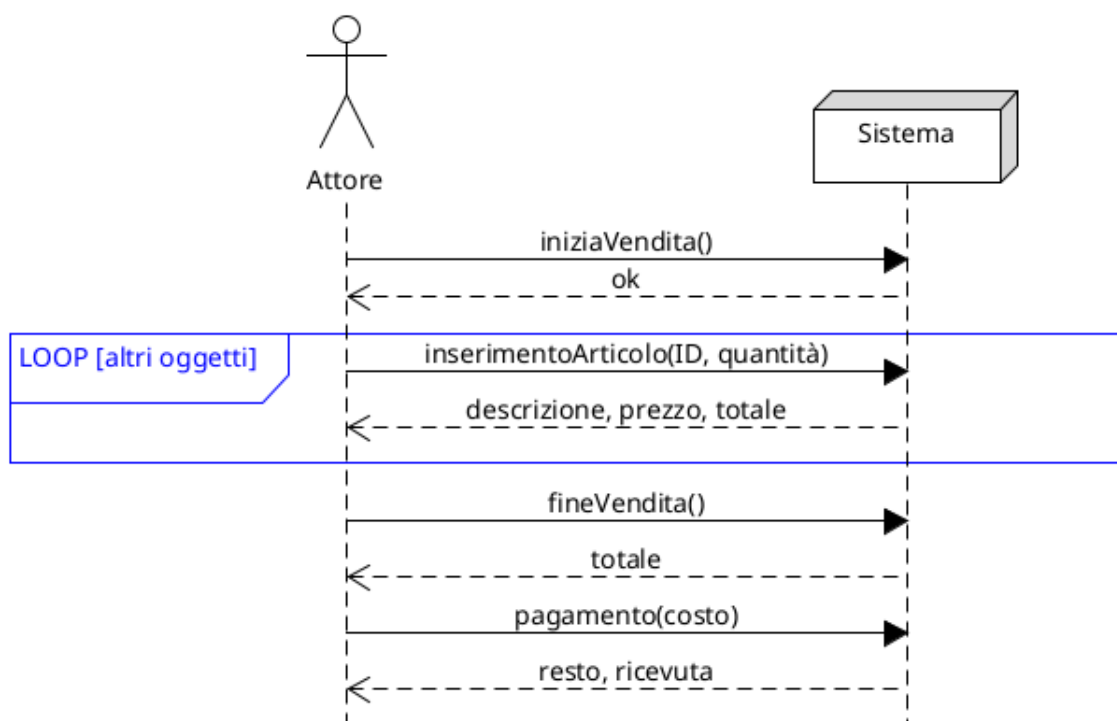
Si consideri il seguente scenario di base di Elabora Vendita:

1. il Cliente arriva alla cassa POS con gli articoli e/o i servizi da acquistare;
2. il Cassiere inizia una nuova vendita;
3. il Cassiere inserisce il codice identificativo di un articolo;
4. il Sistema registra la riga di vendita per l'articolo e mostra una descrizione dell'articolo, il suo prezzo e il totale parziale;
5. il Cassiere ripete i passi 3-4 fino a che non indica che ha terminato;
6. il Sistema mostra il totale;

7. il Cassiere riferisce il totale al Cliente e richiede il pagamento;
8. il Cliente paga (in contanti) e il sistema gestisce il pagamento;
9. il sistema registra la vendita completata;
10. il Sistema genera la ricevuta;
11. il Cliente va via con la ricevuta e gli articoli acquistati.

**Note:-**

Fare attenzione al rumore bianco: non tutti i passi forniscono informazioni utili per la realizzazione del SSD.



## 1.3 Contratti

### Domanda 1.2

Definire in modo preciso la pre-condizione e la post-condizione di un'operazione di Sistema. Fare un esempio di operazione con le sue pre-condizioni e le sue post-condizioni.

### Definizione 1.3.1: Pre-condizioni

Ipotesi significative sullo stato del sistema o degli oggetti nel Modello di Dominio prima dell'esecuzione dell'operazione. Si tratta di ipotesi non banali, che dovrebbero essere comunicate al lettore.

### Definizione 1.3.2: post-condizioni

È la sezione più importante. Descrive i cambiamenti di stato degli oggetti nel Modello di Dominio dopo il completamento dell'operazione.

#### Esempio 1.3.1 (enterItem)

##### Pre-condizioni:

⇒ è in corso una vendita  $s$ .

##### Post-condizioni:

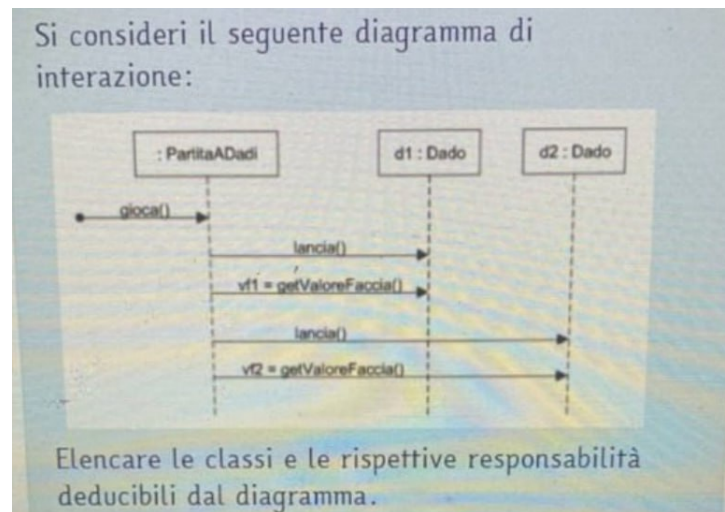
⇒ è stata creata un'istanza  $sli$  di SalesLineItem;

⇒  $sli$  è stata associata con la Sale corrente  $s$ ;

⇒  $sli$  è stata associata con una ProductDescription, in base alla corrispondenza con ItemID;

⇒  $sli.quantity$  è diventata  $quantity$ .

## 1.4 DSD



Le classi coinvolte sono:

⇒ PartitaADadi;

⇒ Dado.

PartitaADadi è il GRASP controller perchè si occupa di chiamare i metodi (delegandoli) sulle altre classi per coordinarle. PartitaADadi ha la responsabilità di conoscere le classi Dado. Ogni Classe Dado ha la responsabilità di conoscere il valore uscito sulla propria faccia dopo il lancio (il Dado è l'*esperto delle informazioni* rispetto a quel valore).

Si consideri l'applicazione *NextGen* vista a lezione e presentata sul libro di testo. Si supponga che abbia una finestra (realizzata mediante la classe *SaleJFrame*) che visualizza le informazioni sulla vendita e che cattura le operazioni del cassiere, la classe *SaleJFrame* è inoltre ascoltatore degli eventi generati dalla pressione dei bottoni della finestra (cioè implementa il metodo *actionListener*). Sono inoltre presenti le classi *Register*, un'astrazione dell'unità fisica, e la classe *Sale* che rappresenta la vendita in corso.

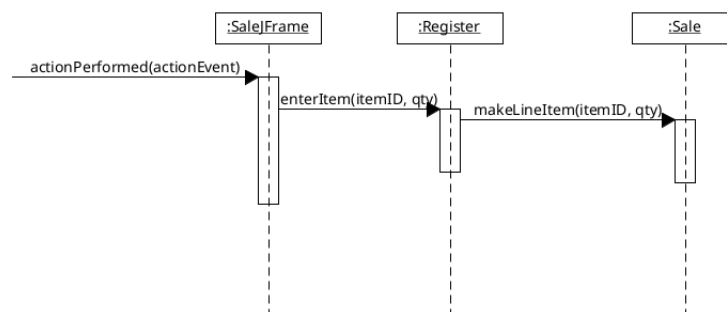
- Si completi il diagramma di sequenza sottostante (si assuma la figura della finestra come rappresentativa dell'IU senza preoccuparsi dei suoi dettagli ma come fosse un'entità unica) inserendo i messaggi *actionPerformed(actionEvent)*, *enterItem(itemID, qty)* e *makeLineItem(itemID, qty)*. Si noti che il messaggio *enterItem(itemID, qty)* è anche il nome del messaggio individuato dall'operazione di sistema che si sta realizzando.
- Si indichi quali classi appartengono allo strato UI e quali allo strato di dominio.
- Si indichi la classe che rappresenta il controllore GRASP.



**Note:-**

- Strato UI: *SaleJFrame*;
- Strato di Dominio: *Register* e *Sale*.

Il GRASP controller è *Register*.

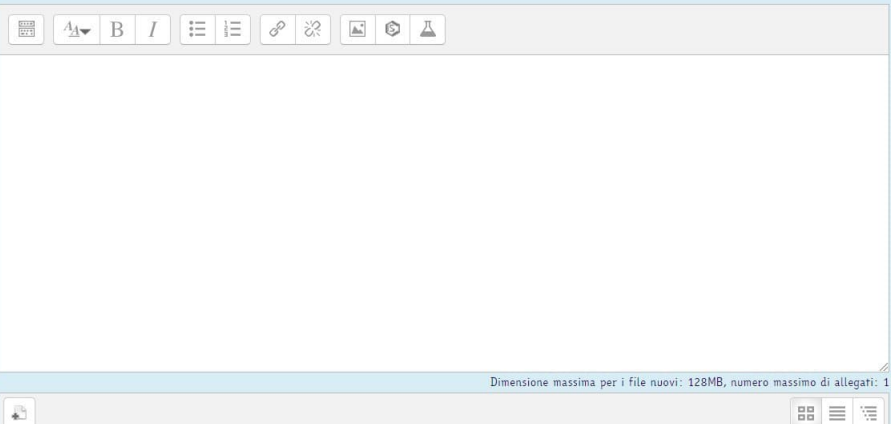




## 1.5 DCD

Rappresentare mediante un diagramma delle classi (modello di dominio) le seguenti affermazioni:

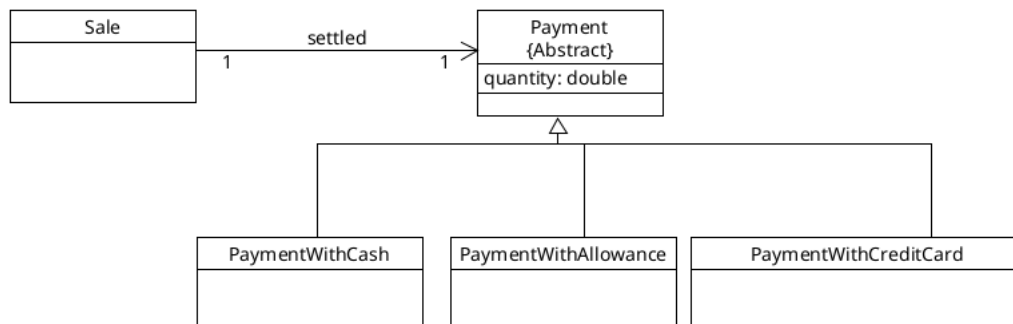
- una vendita è saldata da un pagamento;
- un pagamento è necessariamente associato ad una vendita;
- una vendita è necessariamente saldata;
- un pagamento per contanti, con assegno o con carta di credito, è un pagamento;
- un pagamento è caratterizzato da un ammontare di denaro.



Dimensione massima per i file nuovi: 128MB, numero massimo di allegati: 1

**Note:-**

Prestare attenzione alle cardinalità. Poiché una vendita è obbligatoriamente saldata l'unica cardinabilità disponibile è 1. Per lo stesso motivo, dato che un pagamento salda una sola vendita anche 1





# 2

## Secondo esercizio

### 2.1 GRASP

#### Domanda 2.1

Problema e soluzione di Creator

#### **Pattern 2.1.1** (*Creator - Creatore*):

**Problema:** Chi crea un oggetto A? Chi deve essere responsabile della creazione di una nuova istanza di una classe?

**Soluzione:** Assegnare alla classe B la responsabilità di creare un'istanza della classe A se una delle seguenti condizioni è vera:

- ⇒ B aggrega o contiene A;
- ⇒ B registra A<sup>a</sup>;
- ⇒ B utilizza strettamente A;
- ⇒ B ha i dati necessari per inizializzare A.

---

<sup>a</sup>Registrare significa salvare un riferimento.

#### Domanda 2.2

Problema e soluzione di Information Expert

#### **Pattern 2.1.2** (*Information Expert - Esperto delle informazioni*):

**Problema:** Qual è un principio di base, generale, per l'assegnazione delle responsabilità agli oggetti?

**Soluzione:** Assegnare la responsabilità a un oggetto che ha le informazioni necessarie per soddisfarla, all'esperto delle informazioni, ovvero alla classe che possiede le informazioni necessarie per soddisfare la responsabilità.

**Domanda 2.3**

Problema e soluzione di Low Coupling

**Pattern 2.1.3** (*Low Coupling - Accoppiamento basso*):

**Problema:** Come ridurre l'impatto dei cambiamenti? Come sostenere una dipendenza bassa, un impatto dei cambiamenti basso e una maggiore opportunità di riuso?

**Soluzione:** Assegnare le responsabilità in modo tale che l'accoppiamento (non necessario) rimanga basso.

**Domanda 2.4**

Problema e soluzione di High Cohesion

**Pattern 2.1.4** (*High Cohesion - Coesione alta*):

**Problema:** Come mantenere gli oggetti focalizzati, comprensibili e gestibili e, come effetto collaterale, sostenere Low Coupling?

**Soluzione:** Assegnare le responsabilità in modo tale che la coesione rimanga alta.

**Domanda 2.5**

Problema e soluzione di Controller

**Pattern 2.1.5** (*Controller - Controllore*):

**Problema:** Qual è il primo oggetto oltre lo strato UI che riceve e coordina ("controlla") un'operazione di sistema?

**Soluzione:** Assegnare le responsabilità a un oggetto che rappresenta una delle seguenti scelte:

- ⇒ Un oggetto che rappresenta il sistema (facade controller);
- ⇒ Un oggetto che rappresenta uno scenario di un Caso d'Uso (controller di Caso d'Uso o controller di sessione).

## 2.2 GoF

### 2.2.1 Decorator

```
public class TestPaint {
    public static void main(String[] args) {
        Figure[] figure = new Figure[5];

        figure[0] = new Cerchio();
        figure[1] = new Rettangolo();

        FiguraColorata cerchioBordoRosso = new ColoreBordo(new Cerchio(), "rosso");
        FiguraColorata cerchioRosso = new ColoreSfondo(cerchioBordoRosso, "rosso");

        figure[2] = cerchioRosso;

        FiguraColorata cerchioSfondoBlu = new ColoreBordo(new Cerchio(), "blu");
        FiguraColorata cerchioBlu = new ColoreSfondo(cerchioSfondoBlu, "blu");

        figure[3] = cerchioBlu;

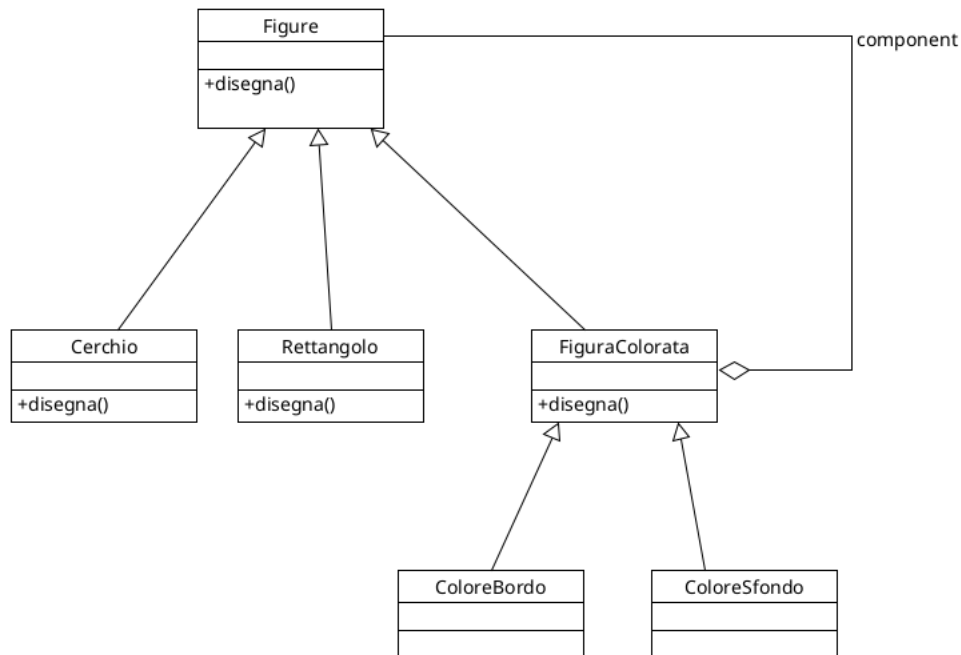
        FiguraColorata rettangoloSfondoBlu = new ColoreSfondo(new Rettangolo(), "blu");

        figure[4] = rettangoloSfondoBlu;
        for (int i = 0; i < 5; i++) {
            figura[i].disegna();
        }
    }
}
```

Questo utilizza un insieme di classi che realizzano un programma di disegno utilizzando un noto pattern GoF. Si dica di quale pattern si tratta e disegnare il diagramma UML delle classi coinvolte.

**Note:-**

Quando vengono aggiunte nuove funzionalità, di solito è Decorator



### 2.2.2 Composite

Si consideri il seguente codice:

```
public abstract class MenuElement {
    private String name;
    private String url;

    public void add(MenuElement component) {
        throw new UnsupportedOperationException();
    }
    public abstract void displayMenu();
}

public class MenuItem extends MenuElement {
    public MenuItem(String name, String url) {
        super(name, url);
    }
    public void displayMenu() {
        System.out.println(getName() + " : " + getUrl());
    }
}

public class Menu extends MenuElement {
    List<MenuElement> subMenus = new ArrayList<>();
    public Menu(String name, String url) {
        super(name, url);
    }
    public void add(MenuElement MenuElement) {
        this.subMenus.add(MenuElement);
    }
    public void displayMenu() {
        System.out.println(getName() + " : " + getUrl() + "\n");
        this.subMenus.forEach(MenuElement::displayMenu);
    }
}
```

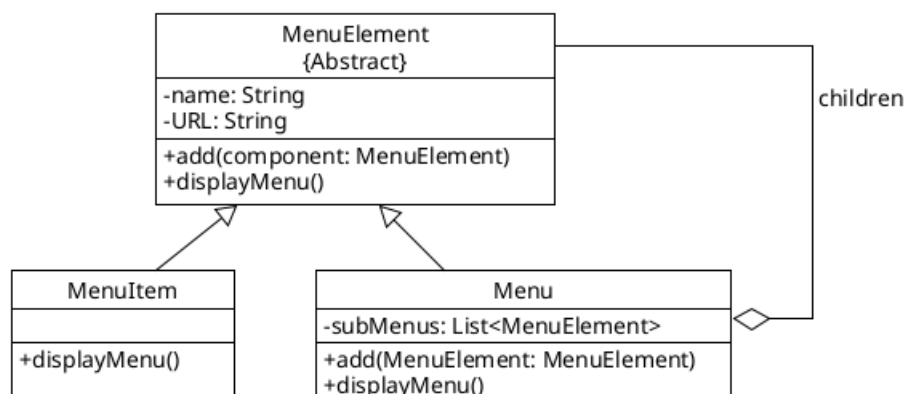
Questo utilizza un insieme di classi che adottano un noto pattern GoF. Si dica di quale pattern si tratta, tenen

```
MenuElement allTutorials = new Menu("Tutorials", "/tutorials");
MenuElement spring = new Menu("Spring", "/spring");
MenuElement versioning
= new Menu("Version Management", "/versioning");
MenuElement java = new MenuItem("Java", "/java");

allTutorials.add(spring);
allTutorials.add(versioning);
allTutorials.add(java);
spring.add(new MenuItem("Spring Core", "/core"));
spring.add(new MenuItem("Spring Boot", "/boot"));
```

#### Note:-

Generalmente i Composite sono alberi o strutture ricorsive. In questo caso si può notare che sia **Menu** che **MenuItem** implementano il `displayMenu()`, ma è possibile aggiungere nuovi elementi solo al Composto (**Menu**). Se si tenta di aggiungere nuovi elementi a **MenuItem**, non essendo ridefinito `add(component: MenuElement)` viene utilizzato il metodo della classe astratta **MenuElement** che lancia un'eccezione.



### 2.2.3 Strategy

Si consideri il seguente codice:

```

public interface CompressionAlgorithm {
    public void compressFiles(ArrayList<File> files);
}

public class ZipCompressionAlgorithm implements CompressionAlgorithm {
    public void compressFiles(ArrayList<File> files) {
        //using ZIP algorithm
    }
}

public class RarCompressionAlgorithm implements CompressionAlgorithm {
    public void compressFiles(ArrayList<File> files) {
        //using RAR algorithm
    }
}

public class CompressionContext {
    private CompressionAlgorithm algorithm;
    //this can be set at runtime by the application preferences
    public void setCompressionAlgorithm(CompressionAlgorithm algorithm) {
        this.algorithm = algorithm;
    }
    public void createArchive(ArrayList<File> files) {
        algorithm.compressFiles(files);
    }
}

public class Client {
    public static void main(String[] args) {
        CompressionContext ctx = new CompressionContext();
        //we could assume context is already set by preferences
        ctx.setCompressionAlgorithm(new ZipCompressionAlgorithm());
        //get a list of files...
        ctx.createArchive(fileList);
    }
}

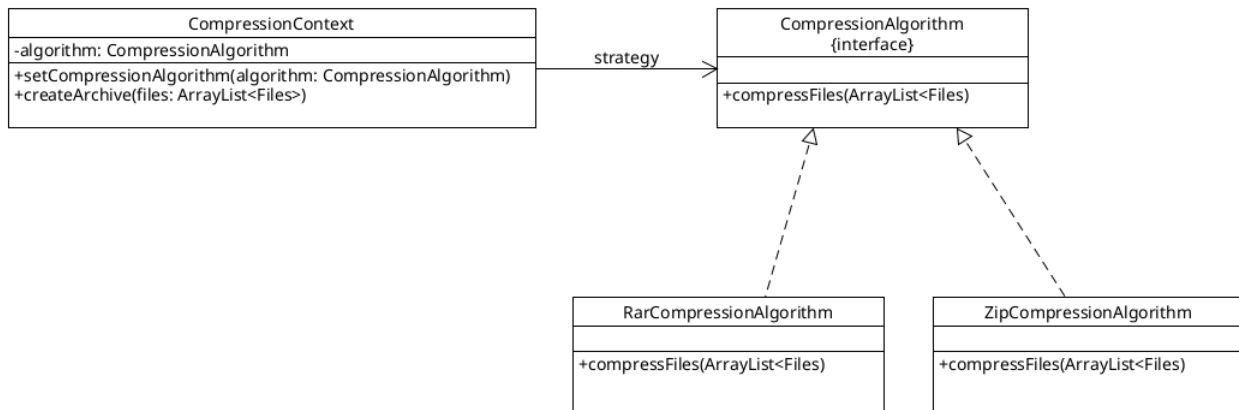
```

Questo utilizza un insieme di classi che adottano un noto pattern GoF. Si dica di quale pattern si tratta.

Scegli un'alternativa:

#### Note:-

Strategy viene utilizzato quando ci sono più possibilità tra cui scegliere. Per esempio tra diversi algoritmi (di ordinamento, compressione, etc.) che fanno tutti la stessa cosa, ma in modo diverso.



### 2.2.4 Visitor

Si consideri il seguente codice:

```
public class ScannerPerCarrello implements Scanner {
    @Override
    public Double scan(MerceVendutaAPeso merce) {
        return merce.getPeso() * merce.getPrezzoAlKg();
    }

    @Override
    public Double scan(MerceVendutaInPezzi merce) {
        return merce.getNumeroDiPezzi() * merce.getPrezzoUnitario();
    }
}

public class Main {

    public static void main(String[] args) {
        List<Merce> carrello = new ArrayList<>();

        MerceVendutaInPezzi p1 = new MerceVendutaInPezzi("C01","Cereali",2.30D,2);
        MerceVendutaInPezzi p2 = new MerceVendutaInPezzi("C02","Quaderno",1.10D,1);
        MerceVendutaAPeso p3 = new MerceVendutaAPeso("C03","Mele",2.50D,2.0D);

        carrello.add(p1);
        carrello.add(p2);
        carrello.add(p3);

        Double totaleSpesa = calcolaTotale(carrello);
        System.out.println("Costo totale " + totaleSpesa + " euro");
    }

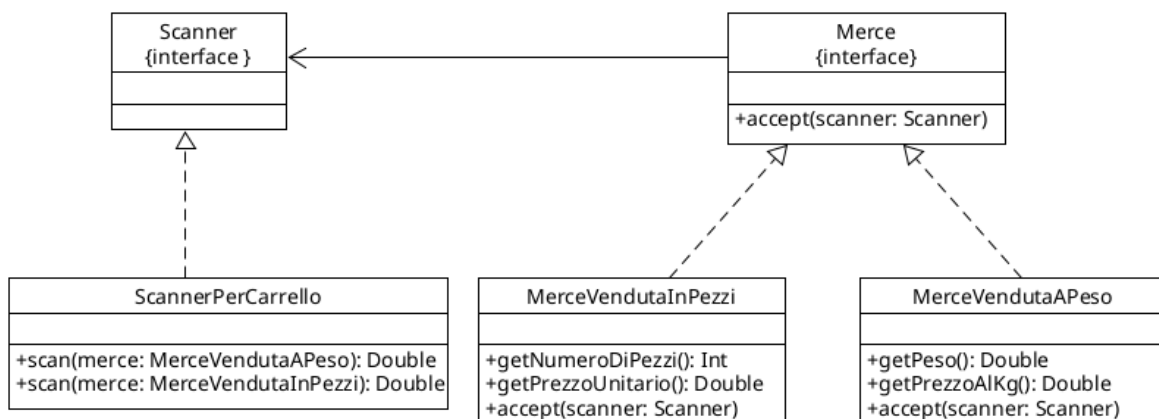
    private static Double calcolaTotale(List<Merce> carrello) {
        Double totale = 0.0D;
        Scanner scanner = new ScannerPerCarrello();

        for(Merce merce : carrello){
            totale = totale + merce.accept(scanner);
        }
        return totale;
    }
}

Questo utilizza un insieme di classi che realizzano uno scanner per carrello della spesa utilizzando un noto pattern GoF.
Si dica di quale pattern si tratta.
```

#### Note:-

Visitor lo si ha quando si tenta di effettuare un'operazione (in questo caso un calcolo) su elementi che possono essere diversi e devono essere trattati in modo diverso (in questo caso `MerceVendutaInPezzi` e `MerceVendutaAPeso`).





## 2.2.5 State

Si consideri il seguente codice:

```
public interface Behavior {
    void onEnter();
    void observe();
}

public class PeacefulBehavior implements Behavior {
    private final Mammoth mammoth;

    public PeacefulBehavior(Mammoth mammoth) {
        this.mammoth = mammoth;
    }
    public void observe() {
        LOGGER.info("{} is calm and peaceful.", mammoth);
    }
    public void onEnter() {
        LOGGER.info("{} calms down.", mammoth);
    }
}

public class AngryBehavior implements Behavior {
    private final Mammoth mammoth;

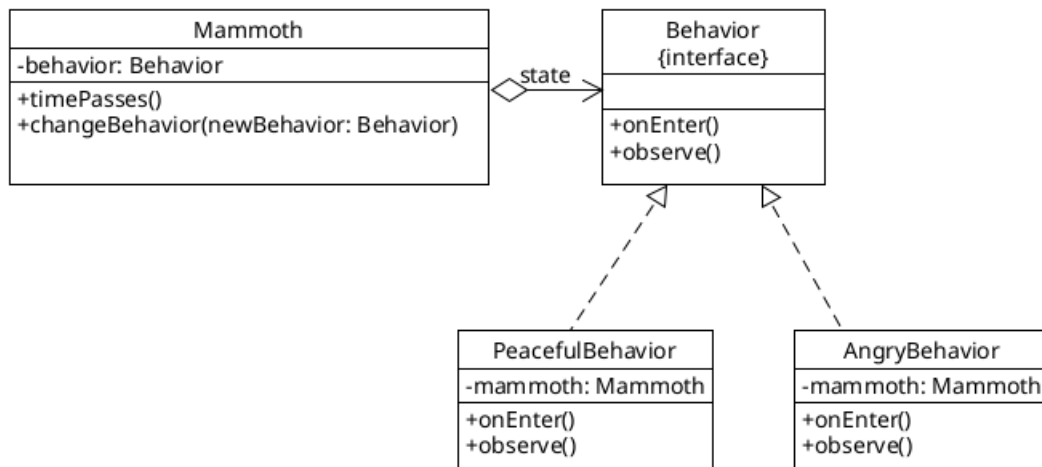
    public AngryBehavior(Mammoth mammoth) {
        this.mammoth = mammoth;
    }
    public void observe() {
        LOGGER.info("{} is furious!", mammoth);
    }
    public void onEnter() {
        LOGGER.info("{} gets angry!", mammoth);
    }
}

public class Mammoth {
    private Behavior behavior;

    public Mammoth() {
        behavior = new PeacefulBehavior(this);
    }
    public void timePasses() {
        if (behavior.getClass().equals(PeacefulBehavior.class)) {
            changeBehaviorTo(new AngryBehavior(this));
        } else {
            changeBehaviorTo(new PeacefulBehavior(this));
        }
    }
    private void changeBehaviorTo(Behavior newBehavior) {
        this.behavior = newBehavior;
        this.behavior.onEnter();
    }
    public String toString() {
        return "The mammoth";
    }
}
```

### Note:-

State si riconosce dal fatto che c'è sempre un "pulsante" per cambiare stato. In questo esempio lo stato cambia a seconda della nuova **Behavior** passata, ma può capitare che lo stato si alterni tra 2 o 3 valori senza necessità di passare esplicitamente un parametro.



## 2.2.6 Observer

Si consideri il seguente codice:

```

public interface ShowMessage {
    public void attach(MessageView o);
    public void detach(MessageView o);
    public void show(Message m);
}

public class MessageContainer implements ShowMessage {
    private List listMessageView = new ArrayList<>();
    public void attach(MessageView o) {
        listMessageView.add(o);
    }
    public void detach(MessageView o) {
        listMessageView.remove(o);
    }
    public void show(Message m) {
        for(MessageView mw: listMessageView) {
            mw.view(m);
        }
    }
}

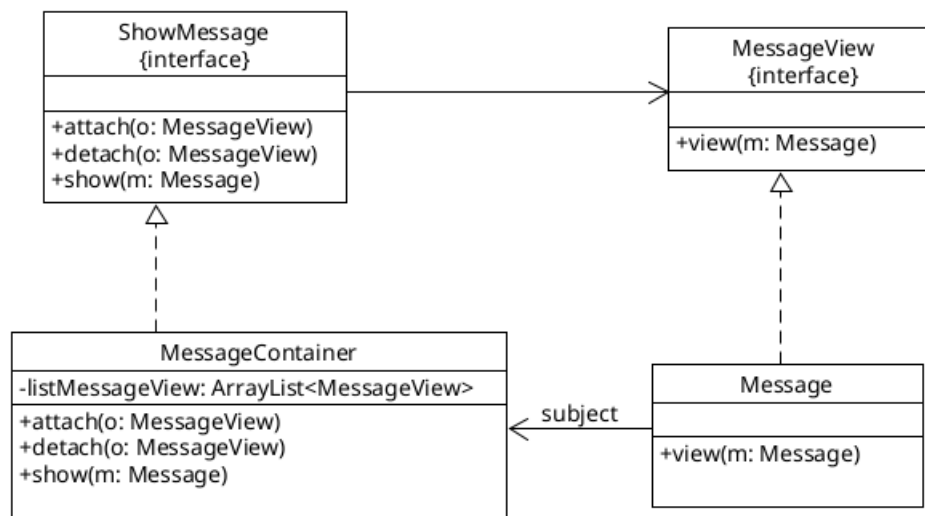
public interface MessageView {
    public void view(Message m);
}

public class MessageViewOne implements MessageView { //similmente per ...Two e ...
    public void view(Message m) {
        System.out.println("MessageViewOne :: " + m.getMessageContent());
    }
}

```

**Note:-**

Observer ha sempre un metodo per attaccarsi/iscriversi e uno per staccarsi/disiscriversi. Inoltre hanno anche un metodo per notificare gli osservatori (in questo caso `show(m: Message)`).



## 2.2.7 Adapter

Si consideri il seguente codice:

```
public class BankDetails{
    private String bankName;
    private String accHolderName;
    private long accNumber;

    public String getBankName() {
        return bankName;
    }
    public void setBankName(String bankName) {
        this.bankName = bankName;
    }
    public String getAccHolderName() {
        return accHolderName;
    }
    public void setAccHolderName(String accHolderName) {
        this.accHolderName = accHolderName;
    }
    public long getAccNumber() {
        return accNumber;
    }
    public void setAccNumber(long accNumber) {
        this.accNumber = accNumber;
    }
}

public interface CreditCard {
    public void giveBankDetails();
    public String getCreditCard();
}

import java.io.BufferedReader;
import java.io.InputStreamReader;

public class BankCustomer extends BankDetails implements CreditCard {
    public void giveBankDetails(){
        try{
            BufferedReader br=new BufferedReader(new InputStreamReader(System.in));

            System.out.print("Enter the account holder name :");
            String customername=br.readLine();
            System.out.print("\n");

            System.out.print("Enter the account number:");
            long accno=Long.parseLong(br.readLine());
            System.out.print("\n");

            System.out.print("Enter the bank name :");
            String bankname=br.readLine();

            setAccHolderName(customername);
            setAccNumber(accno);
            setBankName(bankname);
        }catch(Exception e){
            e.printStackTrace();
        }
    }
    public String getCreditCard() {
        long accno=getAccNumber();
        String accholdername=getAccHolderName();
        String bname=getBankName();

        return ("The Account number "+accno+" of "+accHoldername+" in "+bname+
            " bank is valid and authenticated for issuing the credit card. ");
    }
}

public class PatternDemo {
    public static void main(String args[]){
        CreditCard targetInterface=new BankCustomer();
        targetInterface.giveBankDetails();
        System.out.print(targetInterface.getCreditCard());
    }
}
```

Questo progetto utilizza un noto pattern GoF. Si dica di quale pattern si tratta tenendo conto dell'output fornito:

Enter the account holder name :Matteo Baldoni

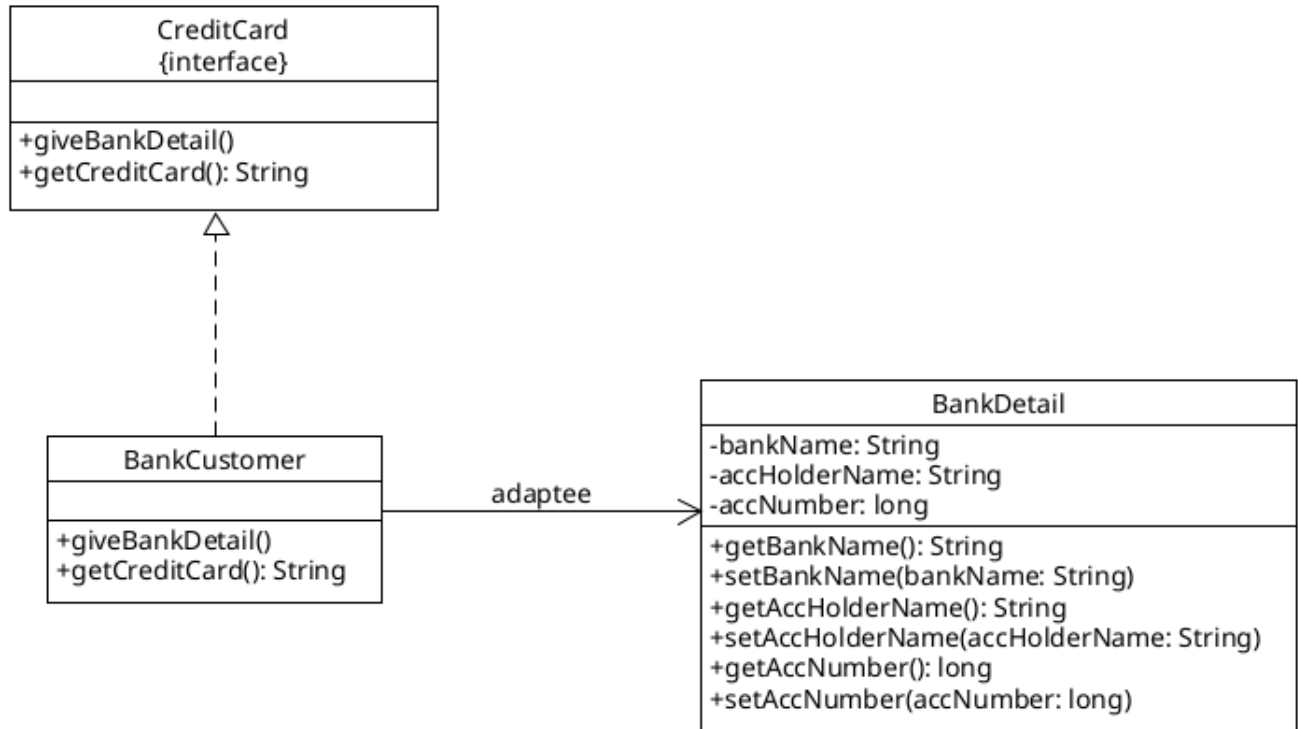
Enter the account number:10100

Enter the bank name :WowBanca

The Account number 10100 of Matteo Baldoni in WowBanca bank is valid and authenticated for issuing the credit card.

**Note:-**

Adapter è abbastanza riconoscibile, ma bisogna stare attenti a quale versione viene messa. Probabilmente se uscirà sarà la versione "oggetto" perché è migliore rispetto alla versione "classe". Ma nulla vieta ai prof. di mettere quest'ultima.



### 2.2.8 AbstractFactory e Singleton

**Note:-**

Singleton è facilmente riconoscibile dal fatto che può apparire come classe statica o con un metodo che restituisce la sua istanza se non è null o una nuova istanza se è null.

**Note:-**

AbstractFactory è evidente dal fatto che viene usato per creare oggetti e non è il Singleton.

