Photo Editor

Giacomo Vitangeli

Indice

1 Introduction **2**

2 Idea **3**

3 Class Diagram **3**

4 Code **4**

5 Test **12**

6 About Design Patterns **14**
6.1 Memento 14
6.2 Observer 15
6.3 Proxy 16
6.4 Strategy 16

7 Conclusion **16**

8 Sources **17**

1 Introduction

L'intento è quello di creare un software che soddisfi alcune funzioni dei più diffusi editor di immagini come *Adobe Photoshop Lightroom*.
Le funzionalità richieste sono:

- **undo** e **redo**: rispettivamente per annullare la modifica di un parametro e ripristinarne lo stato precedente, oppure per annullare l'operazione undo.
- Visualizzare una **preview** dell'immagine, che viene aggiornata automaticamente dopo ogni modifica dei parametri in fase di editing.
- Visualizzare una **preview before-after**, la quale è più articolata e complessa che mette a confronto la stessa immagine senza le modifiche dei parametri applicate e con.
- La possibilità di decidere se esportare l'immagine editata a **lossless quality**, senza perdita di qualità o meno, richiamando due algoritmi di compressione diversi.

Figura 1: Adobe Photoshop Lightroom.

2 Idea

Per realizzare il sistema in questione ho utilizzato un design pattern ciascuna delle funzionalità richieste:

1. **Memento** è un behavioural pattern che permette di ripristinare lo stato precedente dell'oggetto, senza violare l'incapsulamento. Per la funzionalità *undo* spesso viene preferito l'utilizzo del design pattern **Command**, che avvalendosi di più metodi, permette di trattare il ripristino di stati più complessi; mentre Memento richiede un uso più oneroso dal punto di vista della memoria, ma per oggetti leggeri come semplici interi, risulta più performante in termini di tempo rispetto a Command, poiché dispone direttamente dell'oggetto da ripristinare invece di chiamare una funzione che lo ricalcola.
   Al fine di risparmiare memoria, dopo ogni modifica non viene memorizzato l'intero set di parametri ma solo l'ultimo modificato.

2. **Observer** è il pattern ideale per inseguire lo stato di un oggetto, in questa soluzione per una maggiore flessibilità nei tipi di stati da inseguire ho deciso di optare per un Observer di tipo **Pull**, poiché delegando al monitor concreto il compito di recuperare, mediante i metodi getter, lo stato del subject concreto, risulta più semplice adattare il monitor ad eventuali modifiche nel caso in cui dovesse aver bisogno di dati diversi dal subject.
   L'obiettivo dell'observer in questo caso è quello di aggiornare la preview della foto appena viene apportata una modifica.

3. **Proxy** è utile per gestire la visualizzazione sul monitor della preview before-after che ci aspettiamo essere più complessa, per tale motivo questa modalità verrà instanziata solo se richiesta esplicitamente dall'utente.
   Nel programma reale è presente un bottone per abilitare questa funzionalità, per semplicità farò uso di una variabile booleana con il medesimo scopo.

4. **Strategy** sarà il pattern che mi permetterà di fornire all'utente la scelta su che tipo di esportazione effettuare, dal momento che devo eseguire la medesima operazione con due algoritmi di compressione diversi, gli algoritmi sono quello per l'esportazione di file di tipo TIFF (senza compressione) e JPEG (con compressione); così che l'utente possa privilegiare la qualità del file o la sua leggerezza a seconda del fine per cui sta esportando.

3 Class Diagram

Come si può notare dall'UML in Figura 2 i design pattern Memento e Observer si vanno ad unire in corrispondenza della classe Originator che ricopre il ruolo di Subject concreto del pattern Observer e di oggetto che detiene lo stato di interesse per il pattern Memento.
I design pattern Observer e Proxy trovano il punto di contatto nella classe SmartPreview che ricopre il ruolo di Observer concreto e gestisce l'uso dei metodi di visualizzazione standard della preview o della tipologia before-after.

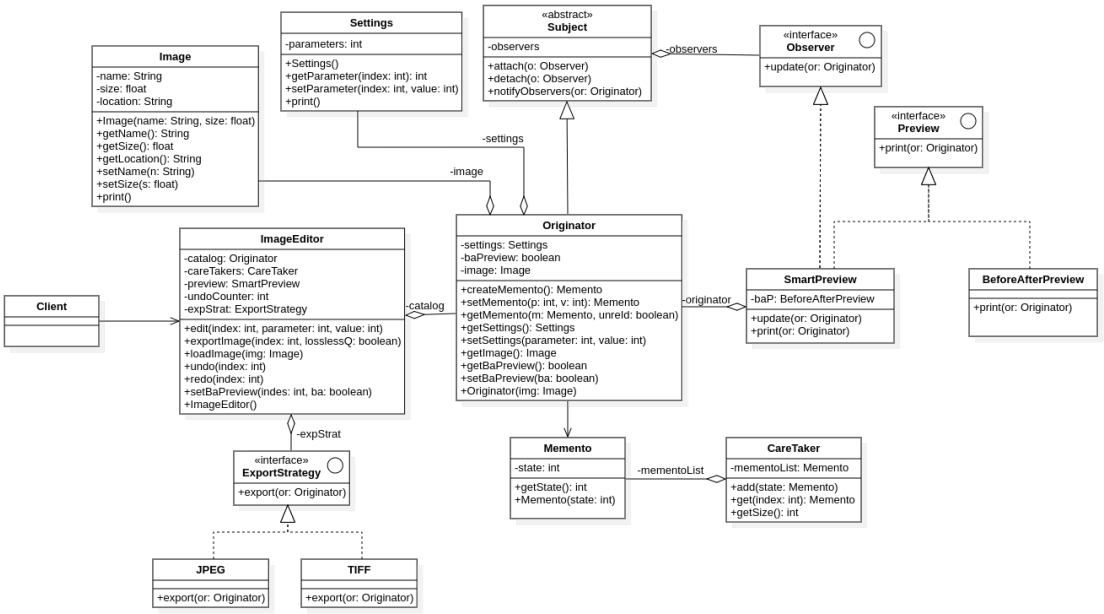

Figura 2: Class Diagram.

4 Code

ImageEditor: l'ImageEditor rappresenta l'interfaccia che il client può utilizzare per editare le immagini ed esportarle, la sua utilità è quella di contenere un catalogo di foto ed associare a queste un Originator che permette di applicare delle modifiche e visualizzarle a schermo tramite la notifica al monitor. Tra gli attributi di questa classe troviamo anche un oggetto di tipo SmartPreview che permette di fare l'attach dell'observer, una lista di CareTaker che per ogni Originator garantiscono l'undo ed infine una strategia per l'esportazione che deve essere istanziata del tipo che interessa all'utente.

```
import java.util.ArrayList;

public class ImageEditor
{
    private ArrayList<Originator> catalog;
    private ArrayList<CareTaker> careTakers;
    private SmartPreview preview;
    private ExportStrategy expStrat;

    public ImageEditor()
    {
        this.catalog = new ArrayList<Originator>();
        this.careTakers = new ArrayList<CareTaker>();
        preview = new SmartPreview();
    }

    public void loadImage(Image img)
    {
        Originator or = new Originator(img);
        or.attach(preview);
```

```
        this.catalog.add(or);
        CareTaker ct = new CareTaker();
        this.careTakers.add(ct);
    }

    public void edit(int index, int parameter, int value)
    {
        this.catalog.get(index).setSettings(parameter, value);
        this.careTakers.get(index).add(this.catalog.get(index).setMemento(parameter,
            value));
        this.catalog.get(index).setUndoCounter(1);
    }

    public void undo(int index)
    {
        int size = this.careTakers.get(index).getSize();
        int count = this.catalog.get(index).getUndoCounter();
        this.catalog.get(index).getMemento(this.careTakers.get(index).get(size-count),
            true);
        this.catalog.get(index).setUndoCounter(count+1);
    }

    public void redo(int index)
    {
        int size = this.careTakers.get(index).getSize();
        int count = this.catalog.get(index).getUndoCounter()-1;
        this.catalog.get(index).setUndoCounter(count);
        this.catalog.get(index).getMemento(this.careTakers.get(index).get(size-count),
            false);
    }

    public void setBaPreview(int index, boolean ba)
    {
        catalog.get(index).setBaPreview(ba);
    }

    public void exportImage(int index, boolean losslessQuality)
    {
        if(losslessQuality)
            expStrat = new TIFF();
        else
            expStrat = new JPEG();
        expStrat.export(this.catalog.get(index));
    }
}
```

Originator: la classe Originator detiene uno stato che è dato da un inseme di parametri (*settings*) e da un'immagine (*Image*) a cui questi parametri sono applicati e quando il suo stato viene modificato invoca il metodo *notifyObservers* passandogli il proprio stato per informare la preview dell'aggiornamento sullo stato.
undoCounter che è necessario per garantire il funzionamento di undo e redo in modo corretto.

```
public class Originator extends Subject
{
```

```
private Image image;
private Settings settings;
private boolean baPreview;
private int undoCounter;


public Originator(Image img)
{
    this.image = img;
    this.baPreview = false;
    this.settings = new Settings();
}


public void setSettings(int parameter, int value)
{
    this.settings.setParameter(parameter, value);
    notifyObservers(this);
}

public Settings getSettings()
{
    return settings;
}

public Memento setMemento(int p, int v)
{
    int edit[] = new int[2];
    edit[0] = p;
    edit[1] = v;
    return new Memento(edit);
}

//unreId: [true to undo | false to redo]
public void getMemento(Memento memento, boolean unreId)
{
    int parameter = memento.getState()[0];
    int value = memento.getState()[1];
    if(unreId)
        value *= -1;

    this.setSettings(parameter, value);
}

public Image getImage()
{
    return image;
}

public void setBaPreview(boolean baPreview)
{
    this.baPreview = baPreview;
}

public boolean getBaPreview()
{
    return baPreview;
```

```
    }

    public void setUndoCounter(int undoCounter)
    {
        this.undoCounter = undoCounter;
    }

    public int getUndoCounter()
    {
        return undoCounter;
    }
}
```

Memento: in questa classe viene salvato lo stato dell'Originator dopo ogni modifica, non viene salvato tutto l'insieme di parametri attuali, ma solo l'ultimo modificato.

```
public class Memento
{
    private int[] state;

    public Memento(int[] edit)
    {
        this.state = edit;
    }

    public int[] getState()
    {
        return state;
    }
}
```

CareTaker: questa classe ha scopo di prendersi cura di una lista di oggetti di tipo Memento, infatti ogni volta che viene istanziato dall'Originator un nuovo Memento questo viene aggiunto alla lista con il metodo *add(Memento state)* e restituito per il ripristino dello stato con il metodo *get(int index)*.

```
import java.util.ArrayList;

public class CareTaker
{
    private ArrayList<Memento> mementoList = new ArrayList<Memento>();

    public void add(Memento state)
    {
        mementoList.add(state);
    }

    public Memento get(int index)
    {
        return mementoList.get(index);
    }

    public int getSize()
    {
```

```
        return mementoList.size();
    }
}
```

Subject: la classe astratta Subject si occupa di fornire all'Originator i metodi necessari per l'osservabilità, per non violare il principio di singola responsabilità questa funzionalità va disaccoppiata dall'Originator che ha già la responsabilità di mantenere il proprio stato.

```
import java.util.ArrayList;

public abstract class Subject
{
    private ArrayList<Observer> observers = new ArrayList<>();

    public void attach(Observer o)
    {
        observers.add(o);
    }

    public void detach(Observer o)
    {
        observers.remove(o);
    }

    protected void notifyObservers(Originator or)
    {
        for (Observer o : observers)
            o.update(or);
    }
}
```

Observer: un'interfaccia semplice ma efficace che permette di avere diverse classi che osservano lo stato di interesse, grazie al metodo *update()*, che viene invocato dal Subject, garantisce la distribuzione del riferimento all'Originator che contiene parametri appena modificati.

```
public interface Observer
{
    void update(Originator or);
}
```

SmartPreview: visualizza tramite una semplice stampa l'immagine con le relative modifiche, se la variabile booleana *baPreview* è stata settata dall'utente a *true*, allora istanzia un oggetto di tipo BeforeAfterPreview ed invoca su di esso il metodo *print(Originator or)*.

```
public class SmartPreview implements Observer, Preview
{
    private BeforeAfterPreview baP;

    public void update(Originator or)
```

```
    {
        print(or);
    }

    public void print(Originator or)
    {
        if (!or.getBaPreview()){
            System.out.println("\nSTANDARD PREVIEW: ");
            or.getImage().print();
            or.getSettings().print();
        }else{
            baP = new BeforeAfterPreview();
            baP.print(or);
        }
    }
}
```

Preview: l'interfaccia del Proxy che permette la scelta tra le due classi di visualizzazione della preview fornendo a entrambe il metodo *print(Originator or)*.
Il metodo dell'interfaccia non viene specificato public poiché le interfacce hanno i metodi public di default.

```
public interface Preview
{
    void print(Originator or);
}
```

BeforeAfterPreview: viene istanziata da SmartPreview solo se l'utente decide di voler visualizzare il confronto tra prima e dopo l'editing della foto, poiché la creazione di questo oggetto si considera onerosa rispetto alla semplice previev. Nella realtà si avrebbe una barra verticale o orizzontale che l'utente può spostare per vedere progressivamente i cambiamenti della foto dovuti all'editing.

```
public class BeforeAfterPreview implements Preview
{
    public void print(Originator or)
    {
        System.out.println("\n\nBEFORE PREVIEW: ");
        or.getImage().print();
        Settings s = new Settings(); //default settings
        s.print();

        System.out.println("\nAFTER PREVIEW: ");
        or.getImage().print();
        or.getSettings().print(); //new settings
    }
}
```

Image: necessaria per la creazione di oggetti di tipo immagine, per semplicità comprende solo tre attributi, nome, dimensione e locazione.

```
public class Image
```

```
{
    private String name;
    private float size;
    private String location;

    public Image(String name, float size)
    {
        this.name = name;
        this.size = size;
        this.location = "/home/giacomo/Pictures";
    }

    public String getName()
    {
        return this.name;
    }

    public void setName(String n)
    {
        this.name = n;
    }

    public float getSize()
    {
        return this.size;
    }

    public void setSize(float s)
    {
        this.size = s;
    }

    public String getLocation()
    {
        return this.location;
    }

    public void print() {
        System.out.println("\nFileName: " + name);
        System.out.println("Location: " + location);
        System.out.println("Size: " + size + " MB");
    }
}
```

Settings: è una classe semplice contenente una array di interi, ad ogni posizione corrisponde un parametro di editing il cui valore è un intero compreso tra -100 e 100.
Quando viene caricata una nuova foto tutti i valori dei parametri sono settati a 0, pronti per essere modificati.
Ho preso in considerazione solo alcuni parametri da usare come esempio, in un editor reale ce ne sarebbero molti di più.
Leggendo il metodo *print()* è possibile capire ad ogni posizione dell'array quale parametro corrisponde.

```
public class Settings
{
```

```
    private int parameters[];

    public Settings()
    {
        parameters = new int[9];
        for(int i=0; i<9; i++)
            parameters[i] = 0;
    }

    public void print()
    {
        System.out.println("\nSETTINGS: ");
        System.out.println("Exposure: " + parameters[0]);
        System.out.println("Contrast: " + parameters[1]);
        System.out.println("Hilights: " + parameters[2]);
        System.out.println("Shadows: " + parameters[3]);
        System.out.println("Whites: " + parameters[4]);
        System.out.println("Blacks: " + parameters[5]);
        System.out.println("Clarity: " + parameters[6]);
        System.out.println("Vibrance: " + parameters[7]);
        System.out.println("Saturation: " + parameters[8]);
    }

    public int getParameter(int index)
    {
        return this.parameters[index];
    }

    public void setParameter(int index, int value)
    {
        int v = this.parameters[index] + value;
        if((v>=-100) && (v<=100))
            this.parameters[index] = v;
        else
            System.out.println("Value out of range!");
    }
}
```

ExportStrategy: è l'interfaccia del design pattern Strategy che ho utilizzato per decidere a run-time quale algoritmo di esportazione utilizzare.

```
public interface ExportStrategy
{
    public void export(Originator or);
}
```

TIFF: questa classe si occupa di esportare le immagini senza compressione con un algoritmo che garantisce la qualità del file.

```
public class TIFF implements ExportStrategy
{
    public void export(Originator or)
    {
        String newName = or.getImage().getName() + ".tif";
```

```
        or.getImage().setName(newName);
        or.getImage().print();
        System.out.println("\nTIFF IMAGE EXPORTED");
    }
}
```

JPEG: questa classe si occupa di esportare le immagini comprimendole con un algoritmo che garantisce un file di dimensione notevolmente ridotta rispetto all'originale.

```
public class JPEG implements ExportStrategy
{
    public void export(Originator or) //simula l'algoritmo di compressione JPEG
    {
        String newName = or.getImage().getName() + ".jpg";
        or.getImage().setName(newName);
        float newSize = or.getImage().getSize()/10;
        or.getImage().setSize(newSize);
        or.getImage().print();
        System.out.println("\nJPEG IMAGE EXPORTED");
    }
}
```

5 Test

Main: per semplicità i test di funzionamento del sistema sono stati scritti nel Main. Prima di tutto viene istanziata la classe ImageEditor che si occuperà di istanziare a sua volta per ogni immagine caricata nel programma un Originator, un CareTaker e uno SmartPreview.
Dopo aver caricato le immagini in un nuovo catalogo si passa alla fase di editing, in particolare a *"DolomitesLandscape"* vengono settati: il Contrasto a +10, i Bianchi a +8, i Neri a -8 e la Chiarezza a +25, prima di quest'ultima modifica viene richiesta la preview before-after. Nell'immagine *"EagleCloseUp"* vengono aperte le Ombre settandole a +10. Dopo aver settato alcuni parametri si verifica il corretto funzionamento di undo() e redo() su entrambe le immagini.
Infine le due immagini vengono esportate con le due diverse tecniche di esportazione generando un file .tif e un .jpg.

```
public class Main {

    public static void main(String[] args) {

        ImageEditor imgEditor = new ImageEditor();

        Image img1 = new Image("DolomitesLandscape", 21);
        Image img2 = new Image("EagleCloseUp", 17);

        //importing a new catalog
        imgEditor.loadImage(img1);
        imgEditor.loadImage(img2);

        imgEditor.edit(0, 1, 10); //Contrast +10
        imgEditor.edit(0, 4, 8); //Whites +8
```

```
        imgEditor.edit(0, 5, -8); //Blacks -8
        imgEditor.setBaPreview(0, true);
        imgEditor.edit(0, 6, 25); //Clarity +25
        imgEditor.setBaPreview(0, false);
        imgEditor.edit(1, 3, 15); //Shadows +10
        System.out.println("\nUNDO:");
        imgEditor.undo(0);
        System.out.println("\nREDO:");
        imgEditor.redo(0);
        System.out.println("\nUNDO:");
        imgEditor.undo(1);
        System.out.println("\nREDO:");
        imgEditor.redo(1);

        //exporting the catalog
        imgEditor.exportImage(0, true);
        imgEditor.exportImage(1, false);
    }
}
```

Output: l'output di questo Main di test è piuttosto lungo, poiché per ogni modifica fatta alle immagini vengono stampati tutti i settings di quell'immagine, perciò mi sono limitato a riportare un esempio di undo-redo e l'esportazione delle due immagini.
Com'è possibile facilmente convincersi dai risultati dei test l'intento è stato raggiunto.

```
[...]

UNDO:

STANDARD PREVIEW:

FileName: DolomitesLandscape
Location: /home/giacomo/Pictures
Size: 21.0 MB

SETTINGS:
Exposure: 0
Contrast: 10
Hilights: 0
Shadows: 0
Whites: 8
Blacks: -8
Clarity: 0
Vibrance: 0
Saturation: 0

REDO:

STANDARD PREVIEW:

FileName: DolomitesLandscape
Location: /home/giacomo/Pictures
Size: 21.0 MB

SETTINGS:
```

```
Exposure: 0
Contrast: 10
Hilights: 0
Shadows: 0
Whites: 8
Blacks: -8
Clarity: 25
Vibrance: 0
Saturation: 0

[...]

FileName: DolomitesLandscape.tif
Location: /home/giacomo/Pictures
Size: 21.0 MB

TIFF IMAGE EXPORTED

FileName: EagleCloseUp.jpg
Location: /home/giacomo/Pictures
Size: 1.7 MB

JPEG IMAGE EXPORTED

Process finished with exit code 0
```

6 About Design Patterns

6.1 Memento

Il modello Memento è un behavioural pattern che viene utilizzato per riportare un oggetto ad uno stato precedente senza violare l'incapsulamento.
Definizione fornita dal GoF: *"Captures and externalizes an object's internal state so that it can be restored later, all without violating encapsulation."*
La funzionalità in questione è **Undo** un comando molto utile in diversi ambiti applicativi tra cui la fase di editing di file multimediali (immagini, video, musica e progettazione 3D) o anche semplicemente file testuali.
Per questo scopo viene spesso utilizzato anche il pattern Command che effettua undo e redo di azioni (metodi).
In questo caso specifico ho adottato Memento poiché lo scopo è memorizzare e ripristinare oggetti di piccole dimensioni.
Questo design pattern è composto da tre classi (vedi Figura 3):

1. **Originator** è l'oggetto per il quale lo stato deve essere salvato. Crea il Memento e lo utilizza in futuro per ripristinare lo stato precedente.

2. **Memento** è l'oggetto che manterrà lo stato di origine, di fatto è solo un pojo, ovvero un oggetto contenente soltanto delle proprietà ed i corrispondenti metodi accessors(get e set) per accedervi rispettando l'incapsulamento.

3. **CareTaker** è l'oggetto che tiene traccia di più oggetti di tipo Memento ogni qual volta lo stato di Originator viene salvato.

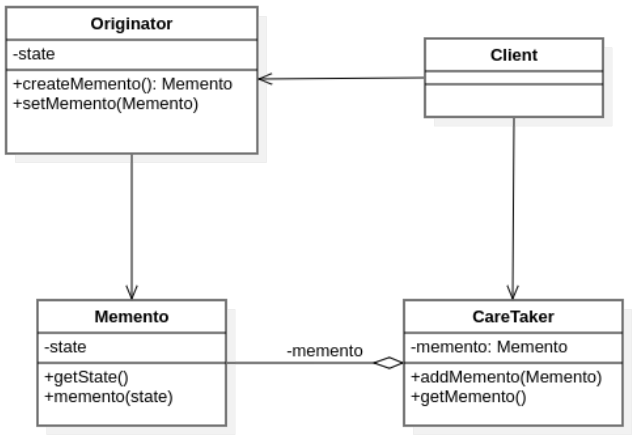

Figura 3: Memento Design Pattern.

6.2 Observer

Un design pattern behavioral utile in molteplici applicazioni, il suo scopo è quello di fornire l'osservabilità rispettando i principi della programmazione ad oggetti, è composto da (vedi Figura 4):

1. **Subject** è una classe astratta si occupa di fornire al ConcreteSubject la capacità di essere osservabile senza violare il principio di singola responsabilità.
2. **ConcreteSubject** è la classe concreta che detiene lo stato che può venire settato dal Client, il che comporta una notifica a tutti gli observers.
3. **Observer** può essere una classe astratta o un'interfaccia, garantisce la riusabilità del codice, perché è estremamente immediato aggiungere nuovi ConcreteObserver.
4. **ConcreteObserver** la classe finale di questo design pattern, si occupa di inseguire lo stato del ConcreteSubject e in base al valore di tale stato solitamente ne consegue qualcosa.

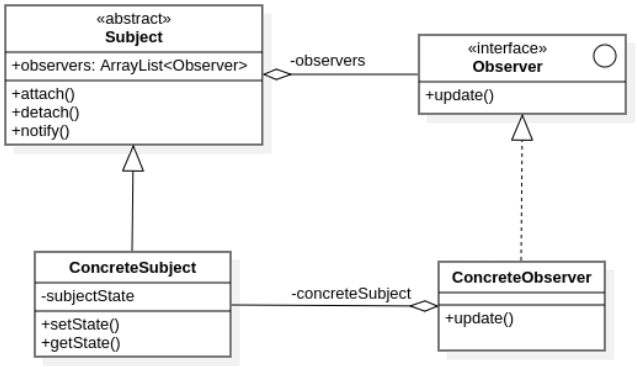

Figura 4: Observer Design Pattern.

6.3 Proxy

Il design pattern strutturale Proxy permette di creare un'interfaccia ed istanziare oggetti complessi solo se realmente necessario.
Il Proxy è composto dalla seguente struttura (vedi Figura 5):

1. **SubjectInterface** è l'interfaccia al di sopra del pattern.
2. **Proxy** è la classe cardine di questo design pattern, si occupa di decidere se allocare e chiamare l'operazione richiesta sulla classe reale.
3. **RealSubject** implementa effettivamente l'operazione richiesta dal Proxy, dopo esser stato istanziato da quest'ultimo.

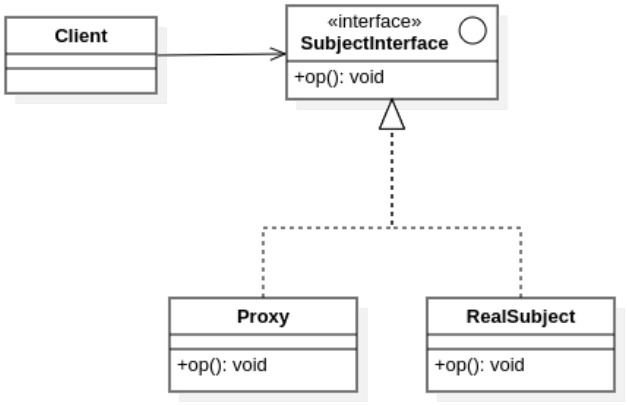

Figura 5: Proxy Design Pattern.

6.4 Strategy

Il design pattern behavioral Strategy permette di creare un'interfaccia che definisce un metodo, il quale verrà implementato in due o più modi diversi.
In fase di esecuzione è possibile decidere quale classe concreta istanziare, sulla quale verrà invocato un metodo.
Lo Strategy è composto dalla seguente struttura (vedi Figura 6):

1. **Context** è la classe in cui si decide quale strategy concreto istanziare.
2. **Strategy** è l'interfaccia al di sopra del pattern che espone il metodo da implementare negli strategy concreti.
3. **ConcreteStrategy** implementa il metodo esposto dall'interfaccia e viene istanziato nel Context.

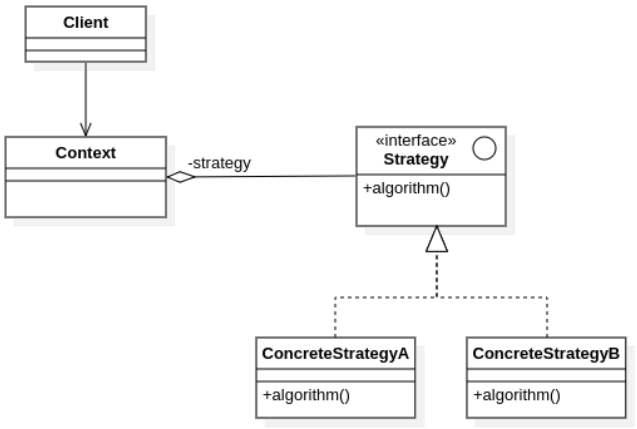

Figura 6: Strategy Design Pattern.

7 Conclusion

In conclusione è possibile affermare che il sistema realizzato soddisfa i requisiti richiesti facendo uso di Memento, Observer, Proxy e Strategy proposti da *Design Patterns: Elements of Reusable Object-Oriented Software (1994) by The "Gang of Four"*.

8 Sources

- Software Engineering Slides by Enrico Vicario
- stackoverflow.com
- dzone.com
- geeksforgeeks.org