1 Introduzione

L'elaborato per l'esame di Ingegneria del Software realizzo prevede la combinazione di 4 Design patterns, i quali sono Strategy pattern, Decorator pattern, Template Pattern e Observer Pattern.

Lo scopo dell'elaborato è quello di leggere un file di testo composto da più righe di testo e processare tale testo in modo da risolvere le seguenti richieste:

1. La possibilità di avere l'output formattato in una maniera particolare, ad esempio tutto maiuscolo.
2. La possibilità di avere l'output in ordine inverso delle righe rispetto al file di ingresso.
3. La possibilità di aggiungere una funzionalità che ci permetta di avere in output anche alcune statistiche riguardo al file processato.
4. La possibilità di stampare l'output su più destinazioni diverse ogni volta che si cambia il file di input.

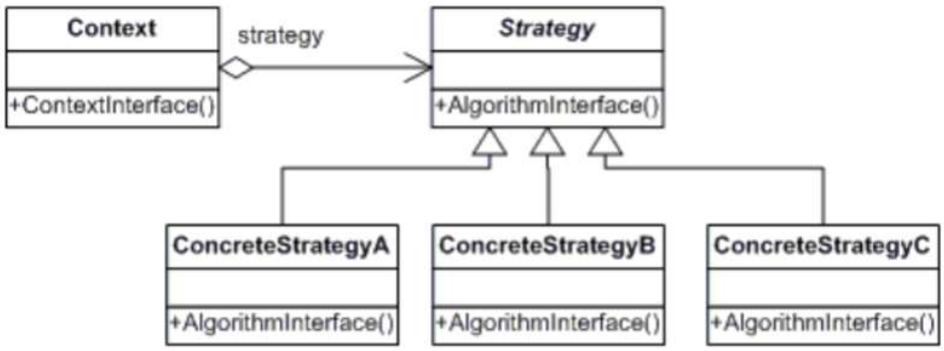
2 Design Patterns

Di seguito viene riportato la descrizione dei design patterns utilizzati per risolvere il problema.

2.1 Strategy Pattern

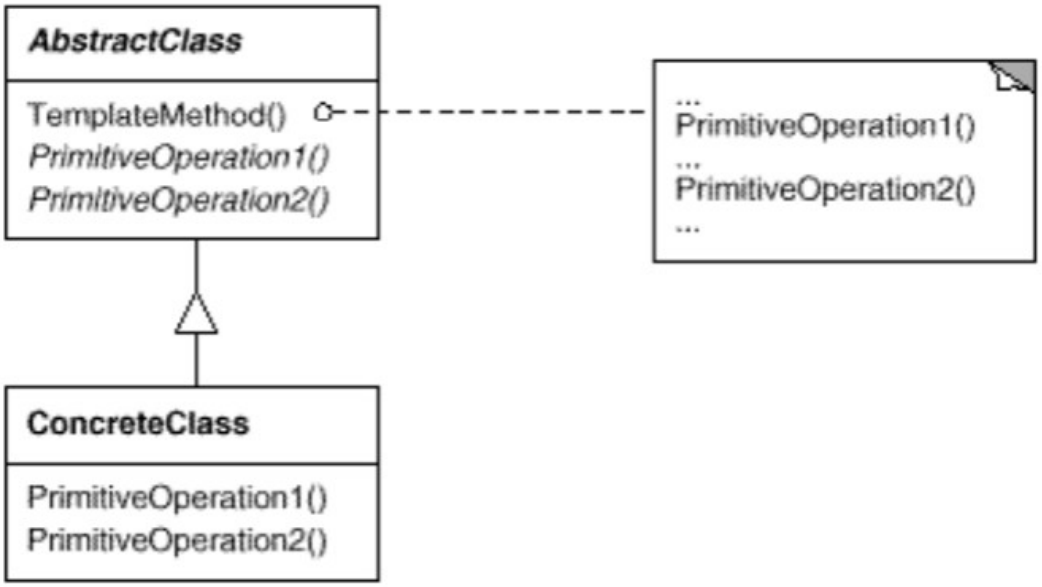
L'obiettivo di questo pattern è definire una famiglia di algoritmi, incapsulare ciascuno di essi e renderli intercambiabili a run-time.

Questo permette che l'algoritmo vari indipendentemente dal Client che lo usa.



2.2 Template Pattern

Pattern comportamentale che viene utilizzato per definire la struttura di un algoritmo delegando alcuni passi di dettaglio alle sottoclassi.



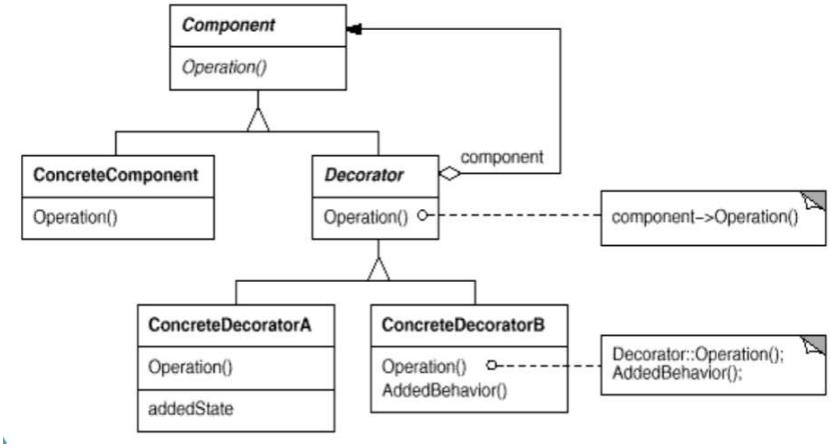
Il template pattern è molto simile con lo strategy pattern la differenza tra i due è quando si sceglie l'algoritmo concreto.

Con il template pattern questo avviene al momento della compilazione, attraverso le sottoclassi. Ogni sotto classe fornisce un diverso algoritmo concreto implementando i metodi astratti del template.

Al contrario, lo schema dello strategy pattern permette di scegliere un algoritmo a runtime, cioè gli algoritmi concreti sono implementai da classi o funzioni separate che vengono passate alla strategy come parametro al suo costruttore o ad un metodo setter. L'algoritmo che viene scelto può variare dinamicamente in base allo stato o agli input del programma.

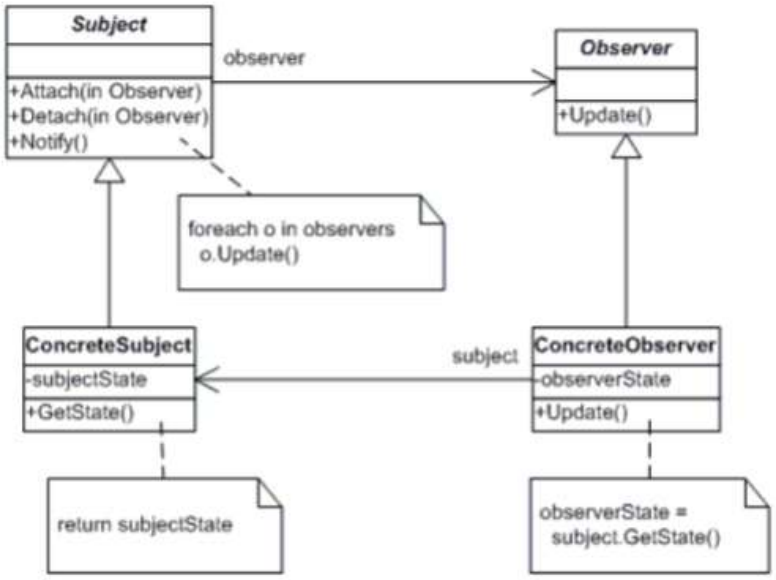
2.3 Decorator pattern

Fornisce un'alternativa flessibile all'ereditarietà per estendere la funzionalità degli oggetti. Tale pattern consente di arricchire dinamicamente, a run-time, un oggetto con nuove funzionalità.

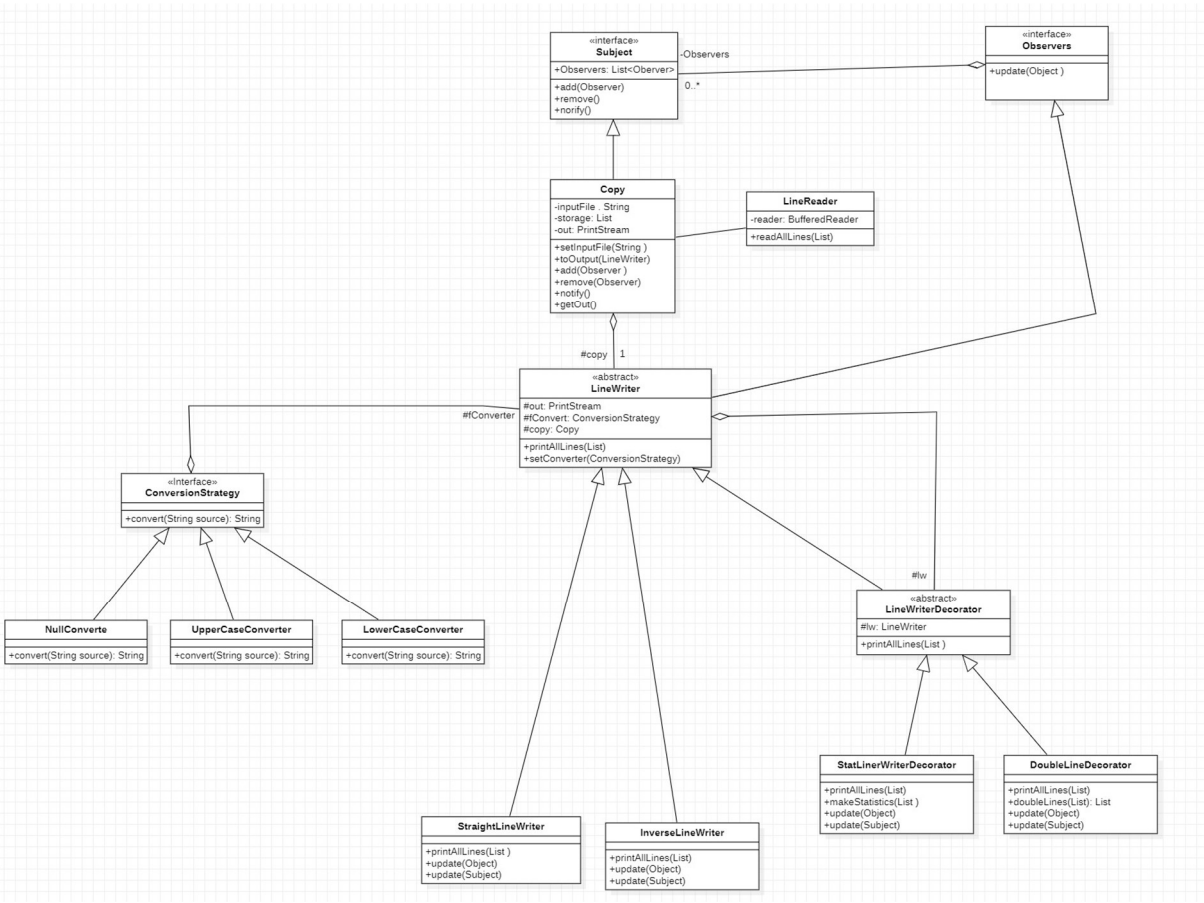


2.4 Observer

Il pattern Observer permette di definire una dipendenza uno a molti fra oggetti in modo tale che se un oggetto (Subject) cambia il suo stato, tutti gli oggetti dipendenti da questo (Observer) siano notificati e aggiornati automaticamente. Per fare ciò il subject tiene una lista con tutti gli observer



3 Implementazione



Partecipanti:

- **LineReader:** classe che permette di leggere le righe di un file salvandole in una lista.

```
public class LineReader {

    private BufferedReader reader;

    public LineReader(String filename){
        try{
            reader = new BufferedReader(new FileReader(filename));
        } catch (IOException e) {e.printStackTrace(); }


    }

    public void readAllLine(List storage){
        try{
            String line = reader.readLine();
            while(line != null){
                storage.add(line);
                line = reader.readLine();
            }
```

```
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

- **Subject**: interfaccia che definisce i metodi per aggiungere, rimuovere e notificare gli observer.

```
public abstract class Subject {
    public abstract void addObserver(Observer o);

    public abstract void removeObserver(Observer o);

    public abstract void notifyAllObservers();

}
```

- **Observer:** interfaccia che definisce il metodo update, la quale deve essere implementata dalle classi concrete.

```
public interface Observer {
    public void update(Object o); //push
    public void update(Subject s); //pull
}
```

- **Copy:** rappresenta il *ConcreteSubject* nel pattern Observer. Oltre a implementare i metodi per poter aggiungere, rimuovere e aggiornare gli observer, espone un metodo per poter conoscere il file con cui sta lavorando. Ogni volta che si cambia il file viene lanciata una nofitica agli observer in ascolto i quali procedono a ristampare l'ouput.

```
public class Copy extends Subject {

    private List storage;
    private PrintStream out;
    private String inputFile;

    private List<Observer> observers;

    public Copy(String inputFile, PrintStream out) {
        storage = new ArrayList();
        this.out = out;
        this.inputFile = inputFile;

        //legge le righe di un file
        LineReader in = new LineReader(inputFile);
        in.readAllLine(storage);

        //instanzio la lista deli Observer
        observers = new ArrayList();
    }

    public void toOutput(LineWriter out) {
        out.printAllLines(storage);
```

```
    }

    public PrintStream getOut() {
        return this.out;
    }

    public void setInputFile(String filename) {
        this.inputFile = filename;
        //update a tutti gli observers
        storage = new ArrayList();
        LineReader in = new LineReader(this.inputFile);
        in.readAllLine(storage);
        this.notifyAllObservers();


    }


    @Override
    public void addObserver(Observer o) {
        this.observers.add(o);
    }

    @Override
    public void removeObserver(Observer o) {
        this.observers.remove(o);

    }

    @Override
    public void notifyAllObservers() {

        for (Observer o : this.observers) {
            o.update(this.storage); //push
            //o.update(this); //pull
        }

    }

    public List getStorage() {
        return this.storage;
    }
}
```

-

- **ConversionStrategy:** rappresenta lo *Strategy* nel pattern Strategy. Dichiara un'interfaccia comune a tutte le classi concrete.

```
public interface ConversionStrategy {
    String convert(String source);
}
```

- **UppercaseConverter:** rappresenta il *ConcreteStrategy* nel pattern Strategy ed implementa l'algoritmo caratterizzante che permette di trasformate la stringa in input tutta in maiuscolo.

```
public class UpperCaseConverter implements ConversionStrategy {

    @Override
    public String convert(String source) {
        return source.toUpperCase();
    }
}
```

- **LineWriterDecorator**: rappresenta il *Decorator* nel decorator pattern. Al suo interno c'è il riferimento a l'oggetto di tipo *LineWriter* già esistente e fornisce l'interfaccia della funzionalità aggiuntiva.

```
public abstract class LineWriterDecorator extends  LineWriter{

    protected LineWriter lineWriter;

    public LineWriterDecorator(LineWriter lw, Copy c) {
        this.lineWriter  =lw ;
        this.out = lw.out;
        this.fConverter = lw.fConverter;
        this.copy = c;
    }


    @Override
    public abstract void printAllLines(List storage);

}
```

- **StatLineWriterDecorator:** rappresenta un *ConcreteDecorator* nel decorator pattern. Fornisce un'implementazione del metodo definito nell'interfaccia. Inoltre, rappresenta anche un *ConcreteObserver* del pattern observer, definisce quindi il metodo update.

```
public class StatLineWriterDecorator extends LineWriterDecorator{

    public StatLineWriterDecorator( LineWriter lw, Copy c) {
        super(lw, c);
    }

    @Override
    public void printAllLines(List storage) {
        lineWriter.printAllLines(storage);

        makeStatistics(storage);
    }

    public void makeStatistics(List storage){
        int tot = 0;
        Iterator i = storage.iterator();
        while(i.hasNext()){
            tot+= i.next().toString().length();
        }
        out.println("Totale caratteri: " + tot);
    }
```

```
    @Override
    public void update(Object o) {
            printAllLines((List) o );
    }
    @Override
    public void update(Subject s) {
        Copy c = (Copy) s;
        printAllLines(c.getStorage());
    }
}
```

- **StraighLineWriter:** rappresenta una classe concreta nel template pattern. Fornisce quindi, un'implementazione del metodo definito nella classe padre. Inoltre, rappresenta anche un *ConcreteObserver* del pattern observer.

```
public class StraighLineWriter extends LineWriter {

    public StraighLineWriter(ConversionStrategy fConverter, Copy
c) {
        this.out = c.getOut();
        this.fConverter = fConverter;
        this.copy = c;
    }

    @Override
    public void printAllLines(List storage) {
        Iterator i = storage.iterator();

        while (i.hasNext()) {
            String line = i.next().toString();
            out.println(fConverter.convert(line));
            //out.append(fConverter.convert(line));
        }
    }

    @Override
    public void update(Object o) {
        printAllLines((List) o);
    }

    @Override
    public void update(Subject s) {
        Copy c = (Copy) s;
        printAllLines(c.getStorage());
    }
}
```

- **LineWriter:** rappresenta il *Component* del decorator pattern, e contemporaneamente rappresenta sia il Context dello strategy pattern , il Template del template pattern e implemente anche interfaccia Observer.

```
abstract class LineWriter implements Observer {

    protected PrintStream out;
    protected ConversionStrategy fConverter;
    protected  Copy copy;

    public abstract  void printAllLines(List storage);

    public void setfConverter(ConversionStrategy stratey){
        this.fConverter = stratey;
    }

}
```