Design Pattern

università di ferrara

DA SEICENTO ANNI GUARDIAMO AVANTI.

Riferimenti

- Gamma, Helm, Johnson & Vlissides (1994). Design Patterns (the Gang of Four book). Addison-Wesley. ISBN 0-201-63361-2
- Steven J. Metsker. Design pattern in Java. Manuale pratico. Addison-Wesley
- Sandro Pedrazzini. Tecniche di progettazione agile con Java. Design pattern, refactoring, test. Tecniche Nuove.
- M. Fowler. UML Distilled Guida rapida al linguaggio di modellazione standard - terza edizione. Pearson Education Italia
- J.Cooper The Design Pattern Java Companion http://www.patterndepot.com/put/8/JavaPatterns.htm
- Wikibooks https://en.wikibooks.org/wiki/Computer_Science_Design_Patterns

Iterator pattern - Step 0

```java
public class MyListArray {

    public int [] lista;

    public MyListArray(int n){

        lista = new int[n];

        for (int i=0;i<n ; i++){
            lista[i]= i*10;
        }

    }

}
```

```java
public class Main {

    public static void main(String[] args) {

        MyListArray la = new MyListArray(10);

        for (int i=0; i< la.lista.length; i++){
            System.out.println("El. "+ i + ":\t"+ la.lista[i]);
        }
    }

}
```

Iterator pattern - Step 1

```
public class MyListArray {

    public int [] lista;

    public MyListArray(int n){
        lista = new int[n];
        for (int i=0;i<n ; i++){
            lista[i]= i*10;}
    }

    private int indice=-1;

    public void rewind(){
        indice=-1;
    }

    public int nextElement(){
        indice++;
        return lista[indice];
    }

    public boolean hasNextElement(){
        return indice<lista.length-1;
    }

}
```

```
public class Main {

    public static void main(String[] args) {

        MyListArray la = new MyListArray(10);

        while (la.hasNextElement()){
            System.out.println("El.: "+ la.nextElement());

        }
    }
}
```

Iterator pattern - Step 2

```java
public class MyListIterator {

    private int [] lista;

    public MyListIterator(MyListArray la){
        lista =la.lista;
    }

    private int indice=-1;

    public void rewind(){
        indice=-1;
    }

    public int nextElement(){
        indice++;
        return lista[indice];
    }

    public boolean hasNextElement(){
        return indice<lista.length-1;
    }

}
```

```java
public class MyListArray {

    public int [] lista;

    public MyListArray(int n){

        lista = new int[n];

        for (int i=0;i<n ; i++){
            lista[i]= i*10;
        }

    }

}
```

```java
public class Main {

    public static void main(String[] args) {

        MyListArray la = new MyListArray(10);

        MyListIterator li = new MyListIterator(la);
        while (li.hasNextElement()){
            System.out.println("El.: "+ li.nextElement());
        }
    }
}
```

Iterator pattern - Step 3

```java
public class MyListIterator implements MyIterator {

    private int [] lista;

    public MyListIterator(MyListArray la){
        lista =la.lista;
    }

    private int indice=-1;

    public void rewind(){
        indice=-1;
    }

    public int nextElement(){
        indice++;
        return lista[indice];
    }

    public boolean hasNextElement(){
        return indice<lista.length-1;
    }

}
```

```java
public interface MyIterator {

    void rewind();

    int nextElement();

    boolean hasNextElement();

}
```

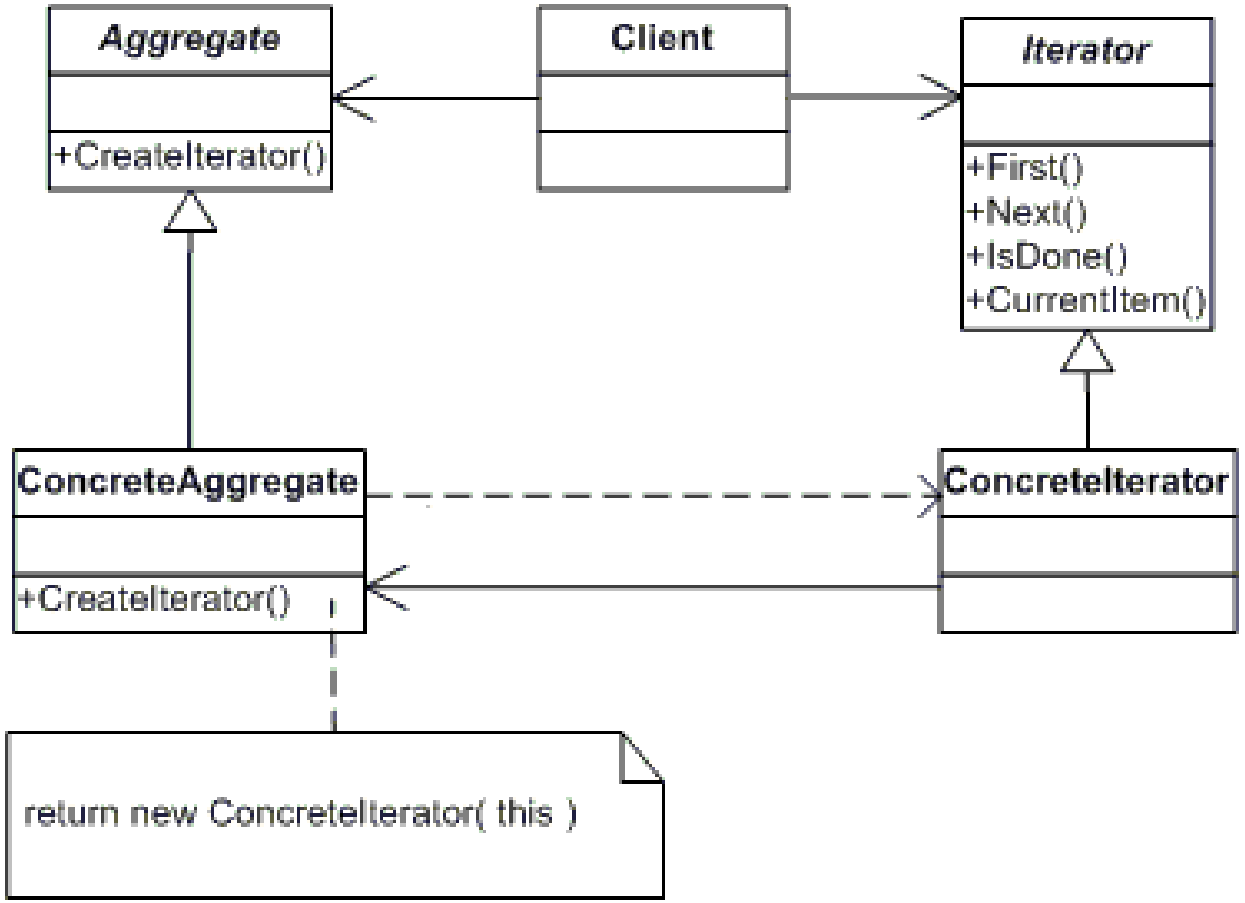
```java
public class Main {

    public static void main(String[] args) {

        MyListArray la = new MyListArray(10);

        MyIterator li = new MyListIterator(la);
        while (li.hasNextElement()){
            System.out.println("El.: "+ li.nextElement());
        }
    }
}
```

Iterator pattern - Step 4

```
public class MyListArray {

    public int [] lista;

    public MyListArray(int n){
        lista = new int[n];
        for (int i=0;i<n ; i++){
            lista[i]= i*10;}
    }

    public MyIterator getIter(){
        return new MyListIterator(this);
    }

}
```

```
public class Main {

    public static void main(String[] args) {

        MyListArray la = new MyListArray(10);

        MyIterator li = la.getIter();
        while (li.hasNextElement()){
            System.out.println("El.: "+ li.nextElement());
        }
    }
}
```

Iterator pattern - UML



Altri Pattern

Un semplice lettore di file di testo:

```
import java.io.*;
import java.util.*;

public class Lettore {

    public static void main(String[] args) {
        String fileName = "testo.txt";
        List storage = new ArrayList();
        try {
            BufferedReader reader =  new BufferedReader(new FileReader(fileName));
            String line = reader.readLine();
            while (line != null){
                storage.add(line);
                line =  reader.readLine();
            }
        } catch(IOException e) { e.printStackTrace();}

        Iterator i = storage.iterator();
        while (i.hasNext()) System.out.println(i.next());
    }
}
```

Una piccola modifica...

E per l'output su altro file?

```
import java.io.*;
import java.util.*;

public class LettoreSuFile {

    public static void main(String[] args) {
        String fileName = "testo.txt";
        List storage = new ArrayList();
        try {
            BufferedReader reader =  new BufferedReader(new FileReader(fileName));
            String line = reader.readLine();
            while (line != null){
                storage.add(line);
                line =  reader.readLine();
            }
        } catch(IOException e) { e.printStackTrace();}


        try {
            String fileOut= "CopyOf_" +  fileName;
            PrintWriter fileOutWriter = new PrintWriter(new FileWriter(fileOut));
            Iterator i = storage.iterator();
            while (i.hasNext()) {
                String line = (String) i.next();
                System.out.println(line);
                fileOutWriter.println(line);
            }
            fileOutWriter.close();
        } catch(IOException e) { e.printStackTrace();}
  }
}
```

Altre richieste

- Output su stream qualsiasi
- Output tutto maiuscolo
- Output anche in ordine inverso
- Output con statistiche e altro...
- Output multiplo in contemporanea

Facciamo un po' d'ordine

- **`LettoreMain`**
  - Seleziona input e ouput
- **`Copy`**
  - Gestisce una **`List`** contenente tutte le righe del file
  - La riempie grazie a **`LineReader`**
  - La scrive grazie a **`LineWriter`**
- **`LineReader`**
  - Legge le linee del file
- **`LineWriter`**
  - Scrive tutte le righe

Main e Copy

```java
public class LettoreMain {
    public static void main(String[] args) {
        String fileName= "testo.txt";
        Copy copy = new Copy(new LineReader(fileName));
        copy.toOutput(new LineWriter(System.out));
    }

}
```

```java
import java.util.*;

public class Copy {
    protected List storage;

    public Copy(LineReader in){
        storage = new ArrayList();
        in.readAllLines(storage);
    }

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

}
```

LineReader e LineWriter

```java
import java.io.*;
import java.util.*;

public class LineReader {

    BufferedReader reader;

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

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

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

```java
import java.io.*;
import java.util.*;

class LineWriter {

    private PrintStream out;

    public LineWriter(PrintStream out) {
        this.out = out;
    }

    public void printAllLines(List storage){
        Iterator i = storage.iterator();
        while (i.hasNext()) {
            String line = i.next().toString();
            out.println(line);
        }
    }

}
```

Prima richiesta

- Avere l'uscita su un altro stream
- Possiamo modificare lo Stream di uscita velocemente, cambiando il suo riferimento nella chiamata del costruttore di **`LineWriter`**.
- Esempio:

```
new LineWriter (new PrintStream("output.txt"))
```

Seconda richiesta

- Vogliamo la possibilità di avere l'output formattato in una maniera particolare, ad esempio tutto maiuscolo
- Definiamo un'interfaccia che mostri il metodo

  ```
  String convert(String)
  ```

- Poi due classi che la implementano in modi diversi
- Possiamo utilizzare il pattern Strategy
- **Strategy pattern**: L'obiettivo di questa architettura è definire una famiglia di algoritmi, incapsulare ciascuno di essi e renderli intercambiabili a run-time.

Strategy pattern

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

```
public class NullConverter implements ConversionStrategy {

    public String convert(String source) {
        return source;
    }

}
```

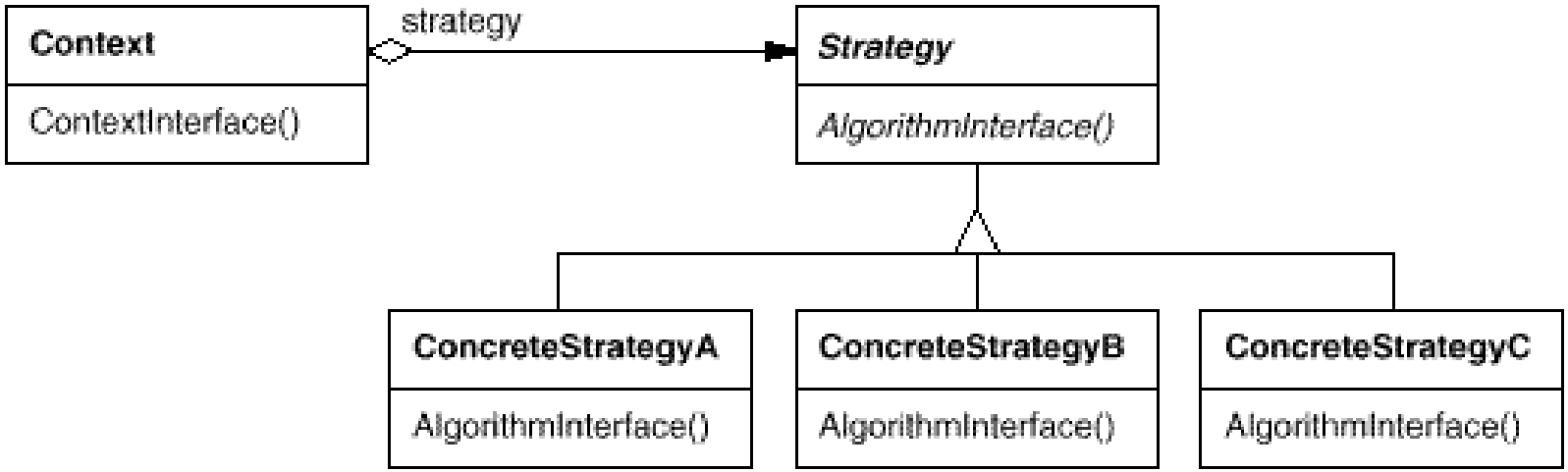
```
public class UpperCaseConverter implements ConversionStrategy {

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

}
```

Strategy pattern

```
import java.io.*;
import java.util.*;

class LineWriter {

    private ConversionStrategy fConverter;
    private PrintStream out;

    public LineWriter(PrintStream out, ConversionStrategy fConverter) {
        this.out = out;
        this.fConverter = fConverter;
    }

    public void printAllLines(List storage){
        Iterator i = storage.iterator();
        while (i.hasNext()) {
            String line = i.next().toString();
            out.println(fConverter.convert(line));
        }
    }

}
```

```
public class LettoreMain {
    public static void main(String[] args) {
        String fileName= "testo.txt";
        Copy copy = new Copy(new LineReader(fileName));
        copy.toOutput(new LineWriter(System.out,new UpperCaseConverter()));
    }

}
```

Strategy pattern - UML



Terza richiesta

- Vogliamo la possibilità di avere l'output in ordine inverso delle righe rispetto al file di ingresso
- Possiamo creare un metodo **`getIterator`** astratto in **`LineWriter`**
- Poi due classi estendono **`LineWriter`** implementando tale metodo in modi diversi
- Possiamo utilizzare il pattern Template
- **Template pattern**: permette di definire la struttura di un algoritmo lasciando alle sottoclassi il compito di implementarne alcuni passi come preferiscono.

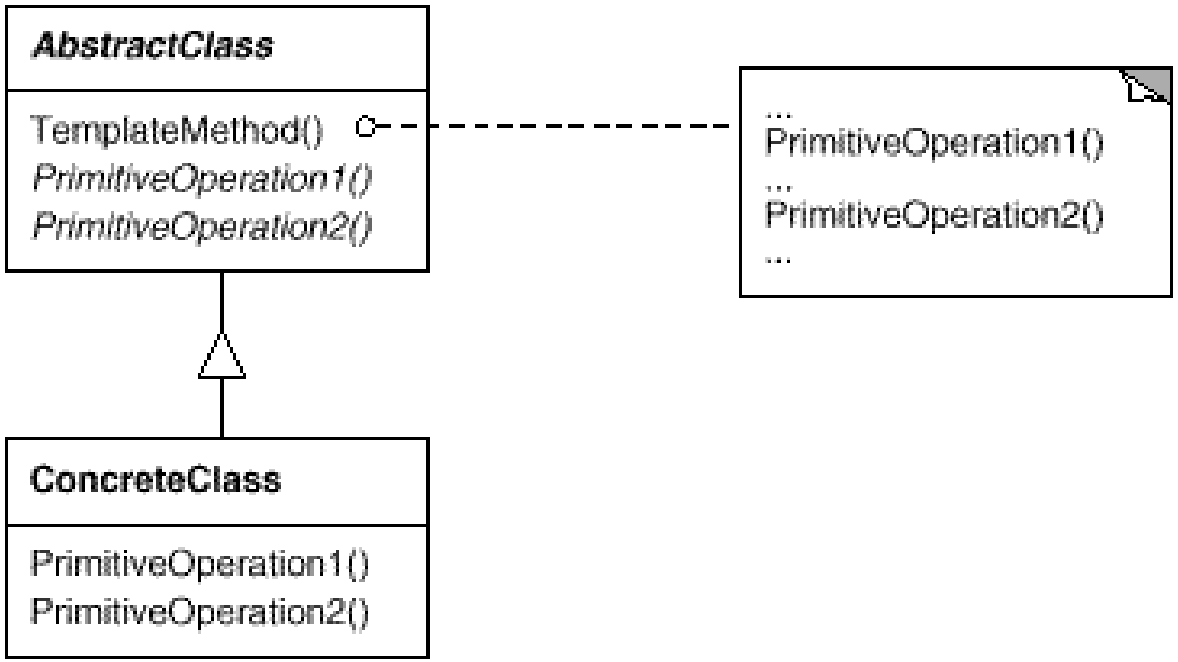
Template pattern

```java
public class LettoreMain {
    public static void main(String[] args) {
        String fileName= "testo.txt";
        Copy copy = new Copy(new LineReader(fileName));
        copy.toOutput(new StraightLineWriter(System.out,new NullConverter()));
    }

}
```

```java
import java.io.*;
import java.util.*;

abstract class LineWriter {

    private ConversionStrategy fConverter;
    private PrintStream out;

    public LineWriter(PrintStream out, ConversionStrategy fConverter) {...}

    public void printAllLines(List storage){...}

    abstract Iterator getIterator(List storage);

}
```

```java
import java.io.*;
import java.util.*;

public class StraightLineWriter extends LineWriter {

    public StraightLineWriter(PrintStream out, ConversionStrategy fConverter) {
        super (out, fConverter);
    }

    @Override
    Iterator getIterator(List storage) {
        return storage.iterator();
    }

}
```

Template pattern - UML



Quarta richiesta

- Vogliamo la possibilità di avere in output anche alcune statistiche riguardo al file processato
- In generale, possiamo voler aggiungere funzionalità alle classi dopo la loro creazione
- Possiamo farlo con il pattern Decorator.
- **Decorator pattern**: consente di aggiungere nuove funzionalità ad oggetti già esistenti.

Decorator pattern

```java
import java.util.*;

interface LineWriter {

    public void printAllLines(List storage);

    public Iterator getIterator(List storage);

}
```

```java
import java.io.*;
import java.util.*;

public class StraightLineWriter implements  LineWriter {
    PrintStream out;
    ConversionStrategy fConverter;

    public StraightLineWriter(PrintStream out, ConversionStrategy fConverter) {...}

    public void printAllLines(List storage){...}

    public Iterator getIterator(List storage) {...}

}
```

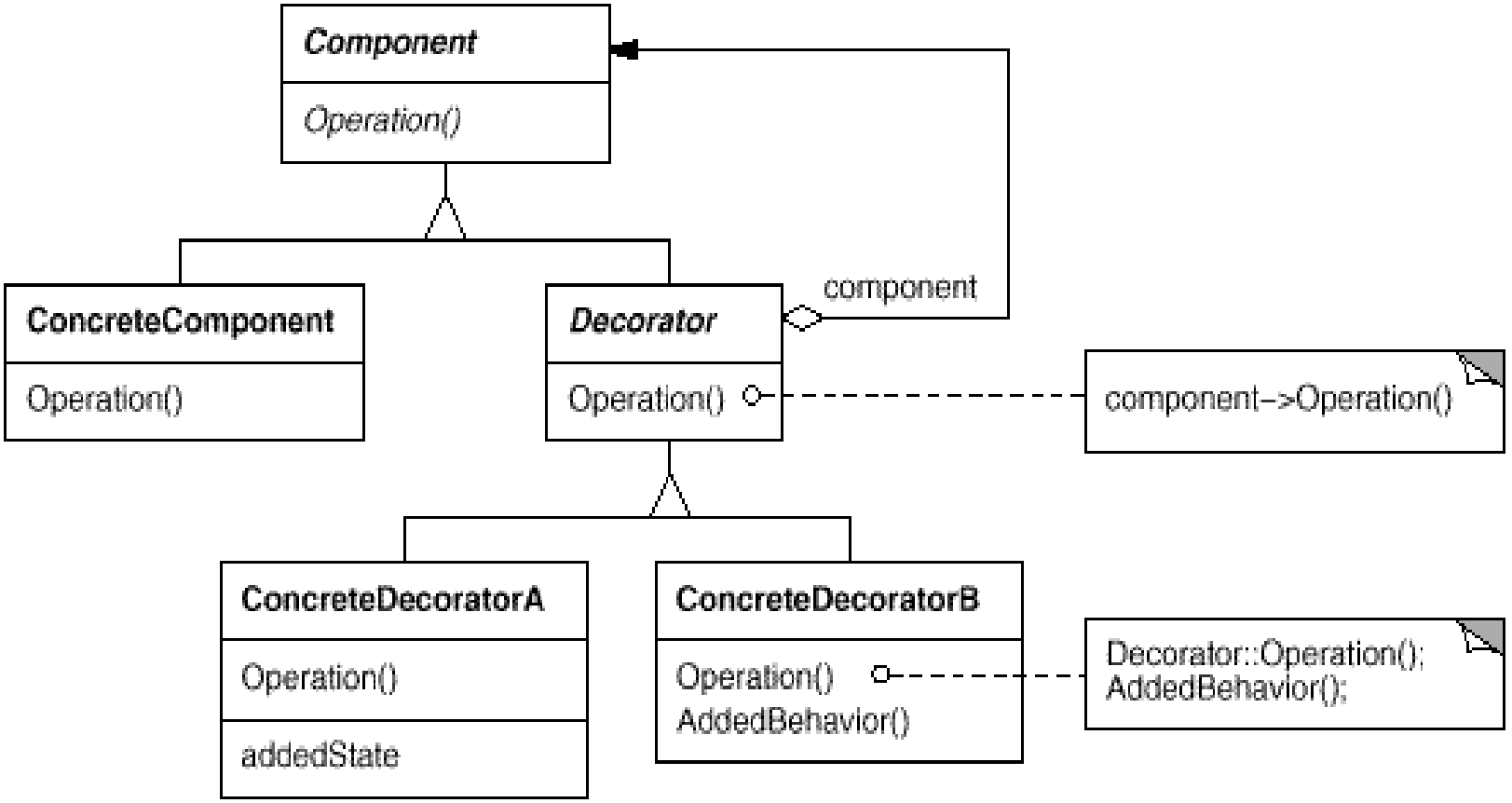
```java
import java.util.*;


public class LineWriterDecorator implements  LineWriter {

    protected LineWriter lw;

    public LineWriterDecorator(LineWriter lw) {
        this.lw = lw;
    }

    public void printAllLines(List storage){
        lw.printAllLines(storage);
    }

    public Iterator getIterator(List storage) {
        return lw.getIterator(storage);
    }

}
```

Decorator pattern

```java
import java.util.*;

public class StatLineWriterDecorator extends LineWriterDecorator{

    public StatLineWriterDecorator(LineWriter lw) {
        super(lw);
    }

    @Override
    public void printAllLines(List storage){
        super.printAllLines(storage);
        makeStatistics(storage);
    }

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

}
```

```java
public class LettoreMain {
    public static void main(String[] args) {
        String fileName= "testo.txt";
        Copy copy = new Copy(new LineReader(fileName));
        LineWriter lw = new StraightLineWriter(System.out,new NullConverter());
        copy.toOutput(new StatLineWriterDecorator(lw));
    }

}
```

Decorator pattern - UML



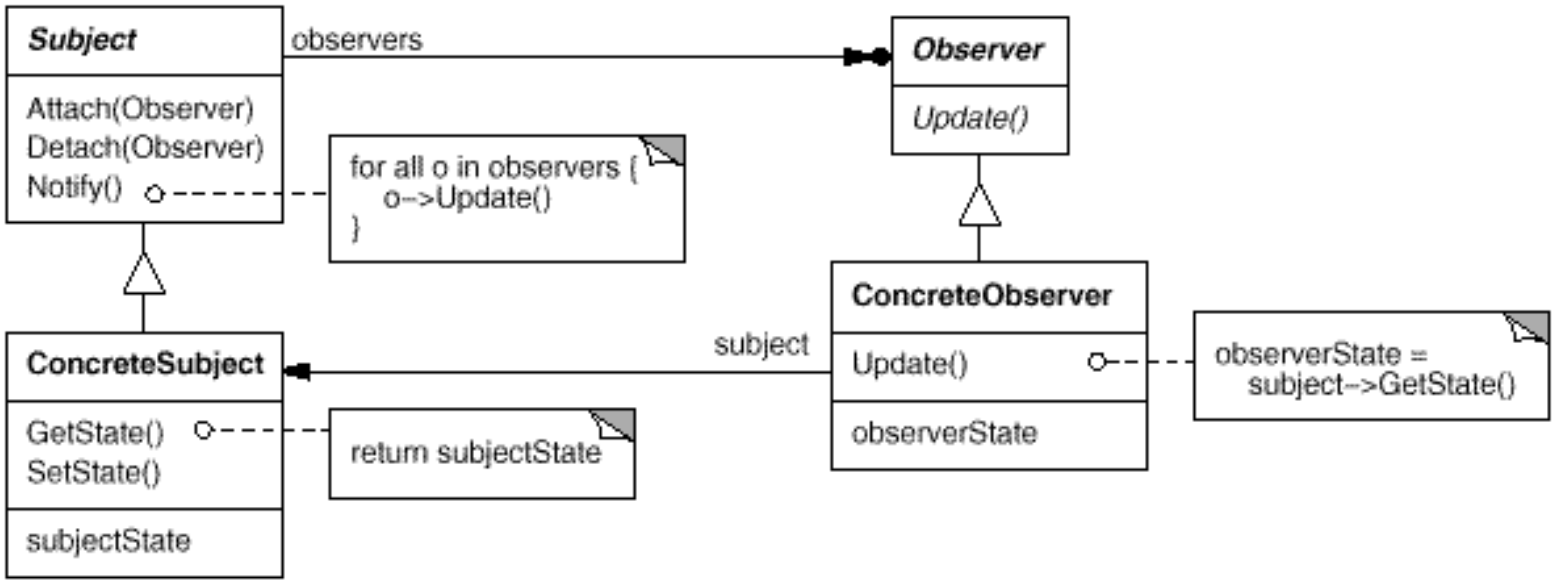
Ultima richiesta

- Stampare l'output contemporaneamente su più destinazioni
- Creare una classe **`Broadcaster`** che contiene una lista di **`LineWriter`**. Avrà i metodi:

  `void add(LineWriter)`

- Inoltre **`Broadcaster`** implements **`LineWriter`** e ridefinirà il metodo **`printAllLines`** inserendo un ciclo che lo inoltra sulla lista di **`LineWriter`**
- Possiamo farlo con il pattern Observer

Observer


Subject
observers
Attach(Observer)
Detach(Observer)
Notify()
for all o in observers {
o->Update()
}
Observer
Update()
ConcreteObserver
Update()
observerState
observerState =
subject->GetState()
subject
ConcreteSubject
GetState()
SetState()
subjectState
return subjectState