Capitolo 12
Applicare i pattern

In questa seconda parte relativa ai pattern si è cercato volutamente di evitare gli elenchi. Ne esistono già parecchi e non è quindi necessario redigerne altri. Lo scopo dell'intera seconda parte del libro è invece quello di spiegare cosa si intende con design pattern e vedere alcune applicazioni pratiche.

Il passaggio più critico è sviluppare la struttura di classi di un'applicazione, individuare quando è necessario l'inserimento di un pattern nell'architettura e, soprattutto, quale pattern va eventualmente inserito. Questo processo va studiato e continuamente esercitato perché richiede esperienza. Eserciteremo questi passaggi con un esempio semplicissimo, che inizialmente non richiederà più di una classe, ma che poi crescerà di pari passo con le nuove esigenze.

Specifiche iniziali

Viene richiesto di scrivere un programma in grado di leggere il contenuto di un file di testo e visualizzarlo in standard output, riga per riga, in ordine inverso.

Vediamo una prima soluzione intuitiva che risponda alle specifiche date.

```
public class Reader
{
   public static void main(String args[]){
      String fileName = args[0];
      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();
         }
         for(int i = storage.size()-1; i>=0; i--){
            System.out.println(storage.get(i));
         }
      }
   }
```

Il programma implementa le specifiche con il minimo sforzo. Per poter visualizzare il contenuto del file in ordine inverso, ogni riga viene registrata in una lista che poi verrà scorsa e visualizzata in standard output dall'ultimo elemento al primo. Il nome del file viene specificato come parametro di `main()`, quindi letto dalla linea di comando. Per leggere un file di testo linea per linea, Java mette a disposizione la classe `BufferedReader` con il suo metodo `readLine()`.

Nuove richieste

Cerchiamo ora di immaginare nuove richieste per complicare leggermente i requisiti del problema e vedere come rendere flessibile il codice con l'aggiunta di modelli di design.

Ordine e stream

Prevedere che il programma sia in grado di stampare il contenuto del file riga per riga, anche in ordine corretto, non necessariamente inverso, e non per forza solo su standard output, ma anche su altri tipi di stream (file).

Trasformazione del contenuto

Immaginiamoci inoltre di prevedere la possibilità di stampare l'output richiesto completamente in maiuscolo, trasformando quindi il contenuto di partenza.

Informazioni supplementari sul contenuto

Ultima richiesta supplementare: vogliamo dare la possibilità di ottenere informazioni in più sul contenuto del testo visualizzato, come ad esempio quante volte compare la lettera "a" nel file di partenza. Con una certa dose di enfasi, identifichiamo questa funzionalità con "statistica".

Ruolo dei pattern

Già nel 1997, quando organizzai per la prima volta un corso sui design pattern, proposi una versione simile di questo problema, discussa per i suoi aspetti didattici con due miei ex colleghi assistenti all'Università di Basilea, Dieter Holz e Urs Hengartner, in seguito nuovamente miei colleghi presso Canoo.

Ero solito proporre agli studenti il problema iniziale come esercizio e poi chiedevo di presentare le varie soluzioni senza discuterle. Tutte risultavano essere molto simili alla versione appena mostrata. In seguito chiedevo a ognuno di inserire a modo suo le aggiunte per soddisfare le nuove richieste. I programmi, pur nella semplicità del problema, crescevano a dismisura, pieni di condizioni e senza nessuna flessibilità. A quel punto diventava chiara l'esigenza di un design che permettesse una gestione migliore delle richieste.

In un libro, un problema del genere, più che come esercizio teorico, si presta maggiormente come *tutorial*, da seguire passo per passo.

Realizzazione diretta

Consideriamo unicamente la prima richiesta, cioè aggiungere la visualizzazione del testo iniziale su file, nello stesso ordine di partenza. Ecco la realizzazione più immediata, partendo dalla soluzione precedente.

```
public class Reader
{
  public static void main (String args[]){
    String fileName= args[0];
    String outFile= args[0] + ".copy";
    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{
      PrintWriter fileOut = new PrintWriter(new FileWriter(outFile));
      String str = null;
      for(int i = storage.size()-1; i>=0; i--){
        System.out.println(storage.get(i));
      }
```

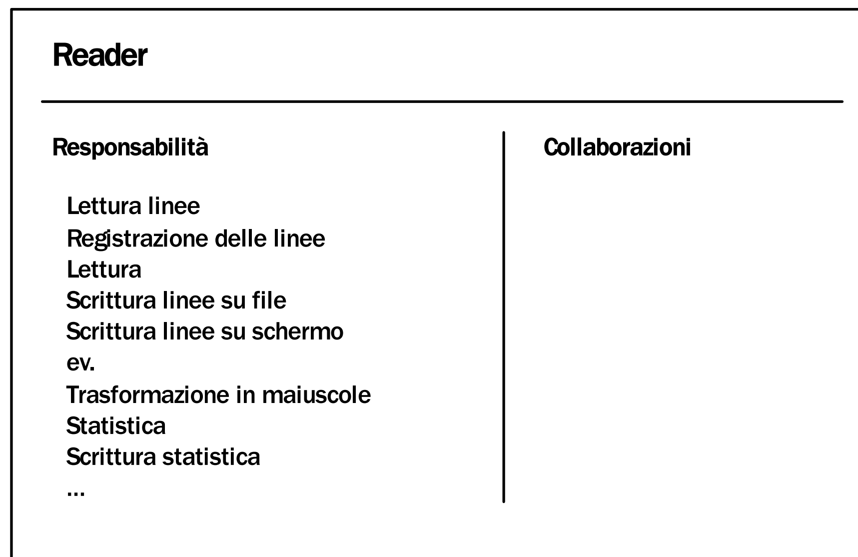

Figura 12.1 – Scheda CRC iniziale.

```
            for(int i = 0; i < storage.size(); i++){
                fileOut.println(storage.get(i));
            }
        }catch (IOException e){
            e.printStackTrace();
        }
    }
}
```

La funzionalità è stata implementata, su questo non c'è niente da dire. Si tratta però di una soluzione che, pur considerando la prima richiesta, non prevede nessun grado di flessibilità. Cosa dovremmo adattare, se invece di due stream di output volessimo prevederne tre, quattro o altri ancora? Come generalizzare questo aspetto? E siamo solo alla prima richiesta. Ce ne sono ancora due da affrontare…

Quella realizzata fin qui è una chiara soluzione *ad hoc*, specifica per un certo caso, ma non aperta a ulteriori cambiamenti. Se le richieste formulate sopra diventassero reali, come vogliamo, dovremmo realizzare il nostro codice in modo più modulare e più object oriented.

Abbiamo già visto quali sono i vantaggi della programmazione a oggetti. Nel nostro caso specifico, dopo aver riutilizzato classi di libreria (BufferedReader, FileReader, PrintWriter e ArrayList), vogliamo sfruttare questo paradigma per aumentare il grado di flessibilità e di adattamento del nostro programma e migliorare il livello di riutilizzo delle singole parti, il che vuol anche dire aumentare la produttività e rendere il programma più robusto.

Nuovo design

Vediamo per prima cosa di definire un design che ci permetta di delegare maggiormente le responsabilità. Se dovessimo illustrare la situazione attuale con le CRC Cards, otterremmo un'unica scheda, quella mostrata in figura 12.1. La scheda ci indica che le responsabilità della classe `Reader`, l'unica classe per ora implementata, sono troppe (grado di coesione basso). Le uniche collaborazioni avvengono con classi di libreria. Collaborazioni che però, per chiarezza, non inseriamo nella scheda.

Cerchiamo allora di ridefinire una suddivisione di compiti che tenga conto dell'elevato numero di responsabilità. Lo facciamo unicamente per le responsabilità identificate fino alla pri-

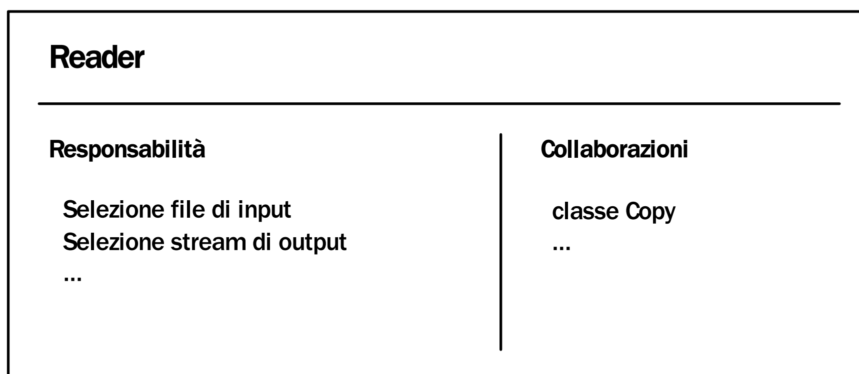

Figura 12.2 – Scheda CRC con nuove responsabilità.

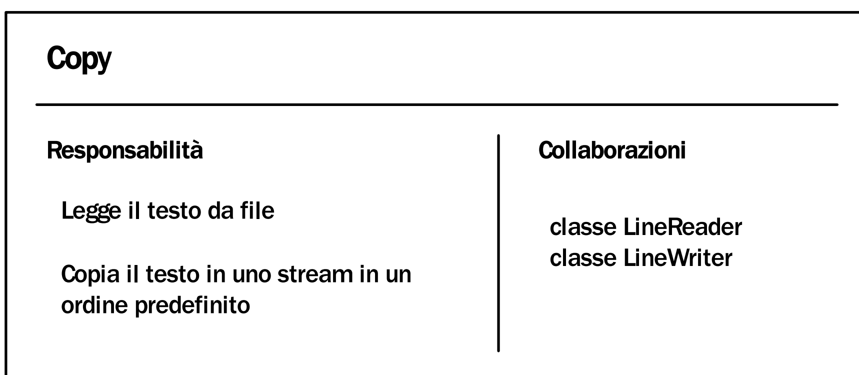

Figura 12.3 – Responsabilità della nuova classe Copy.

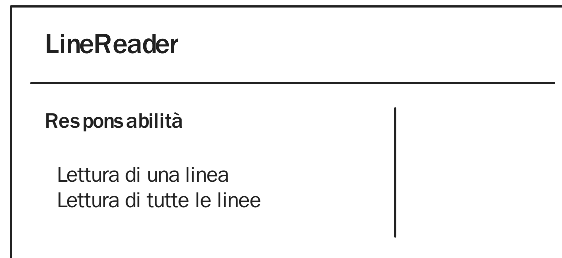

Figura 12.4 – Scheda CRC di LineReader.

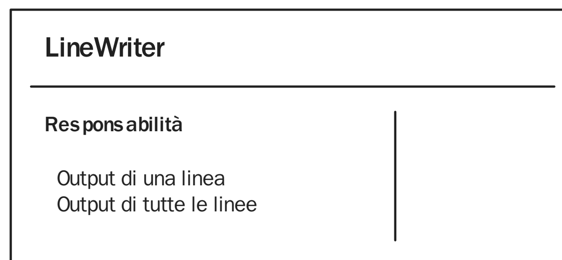

Figura 12.5 – Scheda CRC di LineWriter.

ma richiesta supplementare, quella del secondo canale di output e del diverso ordine di visualizzazione. Ecco, in figura 12.2, le nuove responsabilità assegnate a Reader.

Lo schema presuppone che venga specificata una nuova classe Copy, responsabile della gestione di input e output. Il compito della classe principale Reader si riduce per il momento alla scelta del canale di input e dei canali di output. Vediamo allora le responsabilità della classe Copy e le eventuali collaborazioni, mostrate in figura 12.3.

La classe Copy collabora con i due tipi LineReader e LineWriter, a cui vengono delegati i compiti di lettura di tutto il testo, inserimento riga per riga in una lista intermedia e scrittura riga per riga in uno stream, come mostrato dalle due schede specifiche (figura 12.4 e figura 12.5).

L'elemento responsabile della lettura dal file iniziale è LineReader. Non vengono specificate collaborazioni, perché per semplicità tralasciamo di citare le classi di libreria. La classe è in grado di leggere un intero file di testo, linea per linea.

L'elemento LineWriter ha invece la responsabilità di inviare il contenuto del file in output, riga per riga. Vediamo come si presentano le due classi Reader e Copy dopo questa delega di competenze. Riprendiamo la versione iniziale del problema.

```
public class Reader
{
   public static void main (String args[]){
      String filename=args[0];
      Copy copy = new Copy(new LineReader(fileName));
      copy.toOutput(new LineWriter(System.out));
   }
}

public class Copy
{
   protected List storage;

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

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

Vediamo ora una possibile implementazione di LineReader. Si tratta di riprendere il codice di lettura specificato precedentemente in main() e isolarlo nella classe LineReader, creata per questo compito specifico, cioè per la lettura dell'input riga per riga.

```
public class LineReader
{
   BufferedReader reader;

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

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

```
            storage.addElement(line);
            line=reader.readLine();
          }
        }catch (IOException e){
          e.printStackTrace();
        }
      }
    }
```

Per il momento non vediamo l'implementazione di LineWriter, anche perché vedremo che in realtà sarà più utile definire questa classe come astratta e istanziare nella classe principale del programma delle sottoclassi di LineWriter. Ci basta sapere, al momento, che dopo che LineReader ha letto il contenuto del file di partenza e lo ha inserito in una lista (storage, all'interno di Copy) una sottoclasse di LineWriter lo visualizzerà nello stream di output passato come parametro a printAllLines().

Parametri al costruttore o parametri al metodo?
Una puntualizzazione che non concerne molto il discorso sulle deleghe, ma che va fatta perché ci si trova spesso a confrontarsi con scelte di questo tipo, riguarda i parametri da passare al costruttore di Copy e al suo metodo toOutput(). A chi passare i parametri? Ci sono due varianti estreme: passare tutti i parametri al costruttore, una sola volta e poi chiamare il metodo senza parametri, anche più volte, oppure implementare una versione più vicina al paradigma funzionale, passando a toOutput(), ogni volta, il canale di input e il canale di output richiesti. Nel primo caso dovremmo specificare in Copy variabili di istanza per LineReader, LineWriter e List, nel secondo caso, nessuna variabile di istanza, anzi, il metodo potrebbe essere dichiarato static.
Qual è la soluzione migliore? In generale, la soluzione funzionale (parametri al metodo) è più chiara, perché i valori vanno passati al momento del loro bisogno. Passare tutti i parametri preventivamente al costruttore può essere utile nei casi in cui il metodo toOutput() venisse chiamato più volte all'interno del programma client, sempre con gli stessi oggetti LineReader e LineWriter, e questi non fossero sempre facilmente disponibili.
Ho scelto per il nostro esempio una via di mezzo. L'oggetto di input è sempre lo stesso; inoltre prevede la lettura dei dati da file, che è meglio eseguire una sola volta, perciò viene passato al costruttore che si occupa di riempire la lista. Al metodo toOutput() passiamo invece il canale di output, in modo che sia possibile chiamare toOutput() più volte, con stream di output diversi.

Trasformazioni

Vediamo ora di introdurre alcune modifiche nel codice che ci permettano di realizzare le richieste formulate in precedenza. Iniziamo dalla richiesta di poter scegliere di trasformare il contenuto del file, convertendo tutti i caratteri nei loro corrispondenti maiuscoli. Proviamo a rendere la richiesta ancora più aperta. Se abbiamo più canali di output, deve essere possibile scegliere di trasformare il contenuto per alcuni canali e non per altri. Inoltre vogliamo poter sostituire una conversione con un'altra in qualsiasi momento.

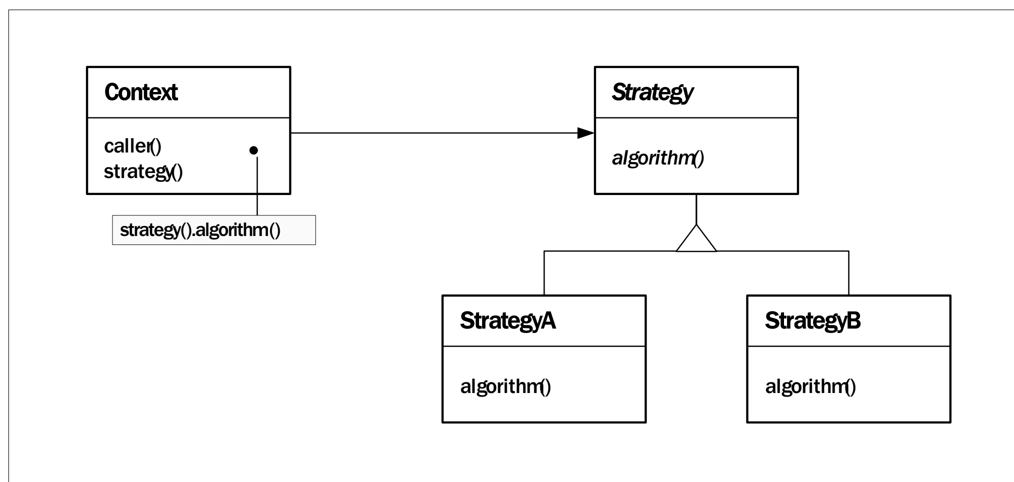

Figura 12.6 – Schema di classi per Strategy.

La scelta diventa quindi quella di racchiudere in un oggetto l'algoritmo di conversione, in modo che questo possa essere passato come parametro. Si tratta di una combinazione delle idee presenti in Strategy e in Command. Strategy serve a implementare l'idea di includere algoritmi interscambiabili in oggetti diversi, con identica interfaccia, come vedremo dalla descrizione, mentre Command incarna l'idea di rappresentare un'azione sotto forma di oggetto, in modo da poterla passare a piacimento all'interno del programma.

Strategy

Scopo del pattern *Strategy* è quello di definire una famiglia di algoritmi interscambiabili. Il client utilizza uno di questi algoritmi, attraverso un'interfaccia comune (in Java realizzata con una classe astratta o con un elemento *interface*).

Utilizzo

Quando diversi algoritmi si possono rivelare appropriati in tempi diversi; oppure quando si desidera realizzare una configurazione dinamica. In figura 12.6 è visibile il diagramma di classi del pattern, già visto nell'esempio iniziale della prima parte.

Conseguenze

Queste sono le conseguenze dell'utilizzo di Strategy:

- Sono possibili diverse implementazioni dello stesso algoritmo, che possono essere interscambiate dinamicamente (il principio "program to an interface, not to an implementation").

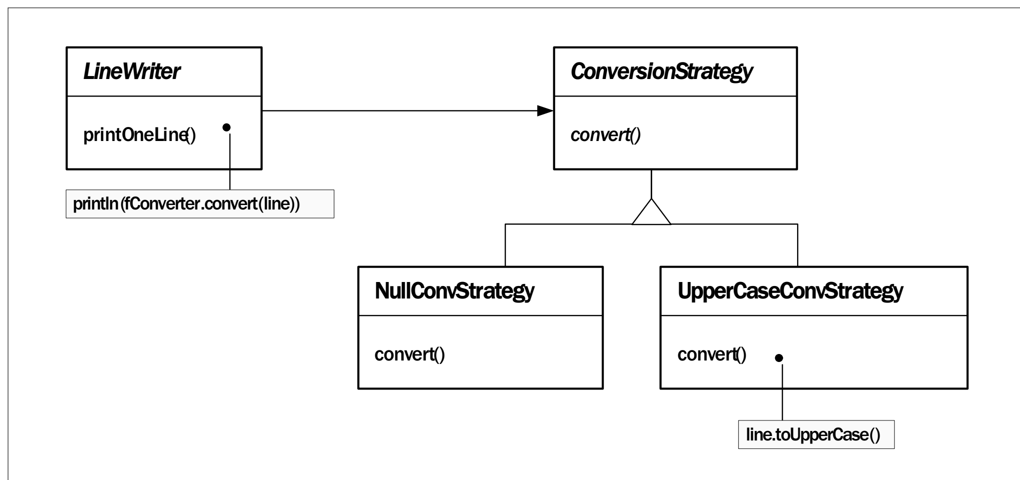

Figura 12.7 – Strategy nel contesto del problema.

- La classe che utilizza l'algoritmo non vede i dettagli dell'implementazione.
- Le classi concrete di Strategy racchiudono unicamente i dati che servono all'implementazione dell'algoritmo.
- L'utilizzo di Strategy evita di inserire nel codice controlli di condizione.
- Aumenta il numero di classi nel programma.

Da notare che un'alternativa all'utilizzo di Strategy potrebbe essere l'ereditarietà di implementazione, che renderebbe però fissa la relazione tra Context e la conversione scelta. Si creerebbero cioè varie implementazioni di Context, in cui l'unica differenza consisterebbe nell'algoritmo di trasformazione. Ed ecco, in figura 12.7, la versione di Strategy adattata al nostro caso specifico. A gestire la gerarchia di Strategy è LineWriter, che lo farà chiamando il metodo convert(), il quale risponderà con una conversione in maiuscolo (UpperCaseConvStrategy), oppure con un nulla di fatto (NullConvStrategy). Dallo schema dovrebbe risultare chiaro che l'aggiunta di ulteriori conversioni avrebbe costi minimi.

Vediamo ora il codice necessario per la realizzazione dello schema.

```
public class LineWriter
{
  private ConversionStrategy fConverter;

  public LineWriter(ConversionStrategy conv){
```

```
      fConverter = conv;
   }

   public void printOneLine(String line){
      println(fConverter.convert(line));
   }
   ...
}

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

public class NullConverter implements ConversionStrategy
{
   public String convert(String source){
      return source;
   }
}

public class UpperCaseConvStrategy implements ConversionStrategy
{
   public String convert(String source){
      return source.toUpperCase();
   }
}
```

Tralasciamo per ora alcuni dettagli di LineWriter, anche perché si tratterà in realtà di una classe astratta, il cui codice verrà attivato da sottoclassi specifiche. Dobbiamo però almeno specificare dove l'oggetto concreto di Strategy viene scelto e viene passato a LineWriter da adottare come algoritmo di conversione. La scelta viene fatta nella classe principale.

```
public class Reader
{
   public static void main (String args[]){
      String fileName=args[0];
      Copy copy = new Copy(new LineReader(fileName));
      copy.toOutput(new LineWriter(System.out, new NullConvStrategy()));
   }
}
```

Ed ecco (figura 12.8) la modifica alle CRC Cards per LineWriter e il suo uso di un convertitore.

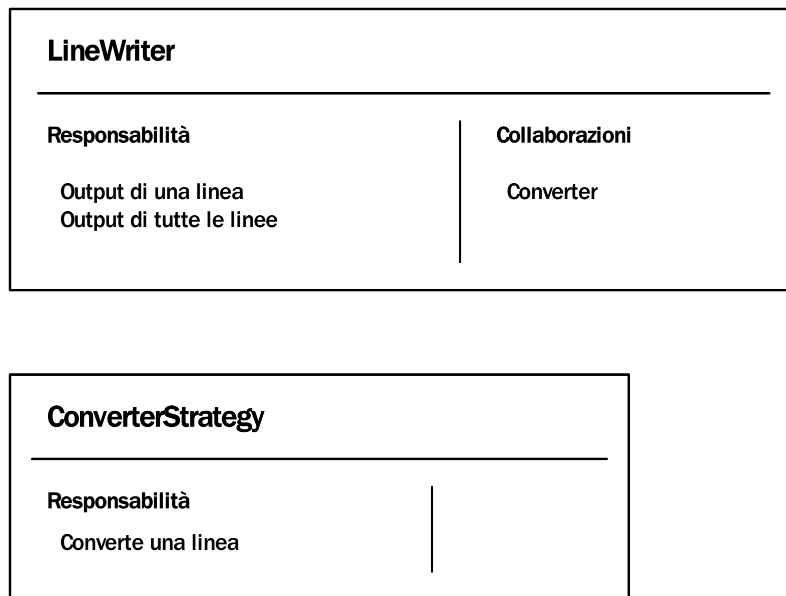

Figura 12.8 – Nuove CRC per LineWriter.

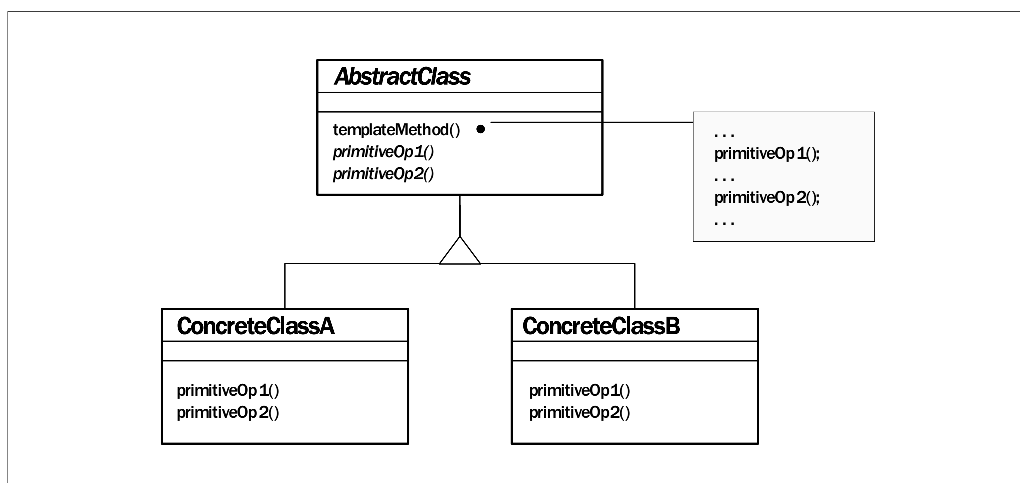

Figura 12.9 – Schema di classi per Template.

Sequenza di output

La prima richiesta supplementare che abbiamo formulato era questa: "Prevedere che il programma sia in grado di stampare il contenuto del file riga per riga, anche in ordine corretto, non necessariamente inverso, e non per forza solo su standard output, ma anche su altri tipi di stream (file)".

Mettiamo da parte per il momento l'idea di avere più stream di output e concentriamoci sulla scelta dell'ordine in cui le righe vengono visualizzate. Vogliamo poter scegliere in ogni momento l'ordine di visualizzazione (ci possono essere altri tipi di sequenza, non solo dalla prima linea all'ultima e viceversa) per ogni stream che entra in considerazione. Ci troviamo in una situazione, in cui abbiamo un algoritmo fondamentalmente identico (stampa delle righe), con un unico elemento diverso (direzione di iterazione). Il pattern Template ci permette la definizione astratta dell'algoritmo, lasciando alcuni punti da definire in sottoclassi (*hot spot*).

Template

Scopo del pattern Template è la definizione dello scheletro di un algoritmo, utilizzando operazioni da definire in sottoclassi. Questo permette alle sottoclassi di implementare i metodi utilizzati nell'algoritmo, senza doverlo riscrivere e duplicare.

Utilizzo

Quando si desidera l'implementazione unica delle parti invarianti di un algoritmo, lasciando le parti varianti alle sottoclassi; quando è necessario riassumere un comportamento comune (risultato di "refactoring to generalize"); per limitare l'estensibilità di un algoritmo a parti ben definite dello stesso. In figura 12.9 è riportato il diagramma di classi.

Conseguenze

Queste sono le conseguenze dell'utilizzo del pattern Template in un caso come il nostro:

- Riutilizzo del codice (nella buona tradizione dell'ereditarietà di implementazione).
- Definizione di una parte invariante (classe astratta) e di una parte variante (classi concrete).
- Controllo del flusso globale con adattabilità delle singole componenti (principio del framework).
- Possibilità di definire operazioni cosiddette *hook*, che realizzano un comportamento di default che può poi essere sovrascritto da sottoclassi.

Nel nostro caso l'algoritmo in comune (stampa delle righe), viene specificato in LineWriter (classe astratta), mentre le sottoclassi, per ora LineStraightWriter e LineReverseWriter, aggiungeranno la scelta dell'iteratore corrispondente.

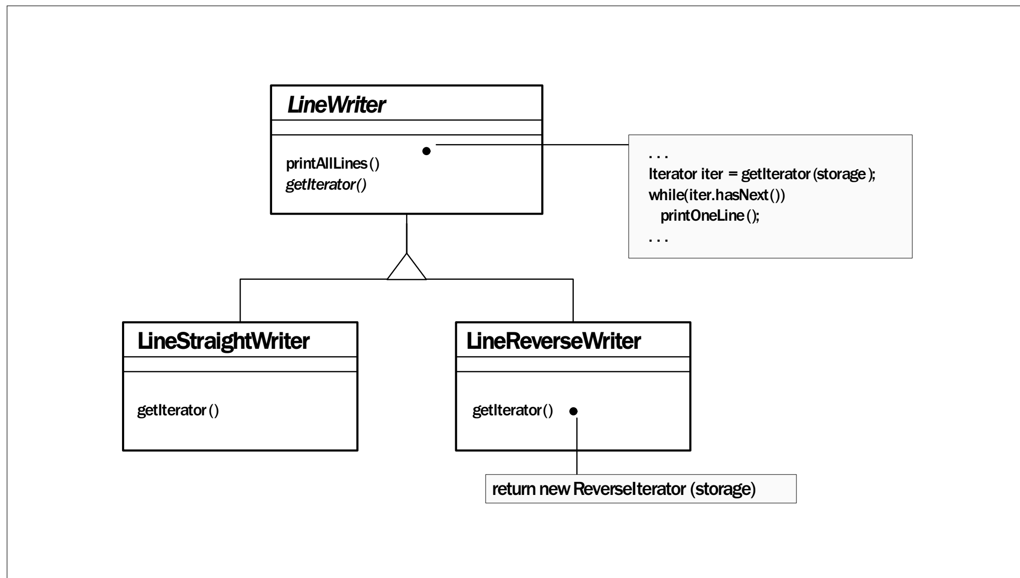

Figura 12.10 – Template nel contesto del problema.

La classe LineWriter diventa un'interfaccia.

```
public interface LineWriter
{
   Iterator getIterator(List storage);
   void printAllLines(List storage);
}
```

Il metodo getIterator() di LineStraightWriter sarà molto semplice perché potrà sfruttare l'iteratore di default di storage, che appartiene a List.

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

Per la classe LineReverseWriter bisognerà invece mettere a disposizione un iteratore specifico, in grado di leggere le linee immagazzinate in storage in ordine inverso. La classe dovrà comunque implementare l'interfaccia Iterator, messa a disposizione dall'ambiente Java. Da notare che, essendo il metodo usato in Template uno che serve fondamentalmente a creare un oggetto, questo utilizzo di Template si fonde con l'idea del pattern Factory Method.

Ma ecco come viene implementata la scelta in Reader, la classe principale. La soluzione presentata sceglie di mostrare in output le linee in ordine inverso, senza conversione del contenuto.

```
public class Reader
{
  public static void main (String args[]){
    String fileName=args[0];
    Copy copy = new Copy(new LineReader(fileName));
    copy.toOutput(new LineReverseWriter(System.out,new NullConvStrategy()));
  }
}
```

A questo punto, se volessimo visualizzare il contenuto nell'ordine corretto su un secondo stream di output, le modifiche al codice andrebbero eseguite unicamente in Reader. Mostriamo l'esempio, in cui il secondo stream di output è un file, e per questo tipo di output viene anche scelto di eseguire la conversione in maiuscolo.

```
public class Reader
{
  public static void main (String args[]){
    String fileName=args[0];
    String outFile=args[0]+ ".copy";
    PrintWriter fileOut = new PrintWriter(new FileWriter(outFile));
    Copy copy = new Copy(new LineReader(fileName));
    copy.toOutput(new LineReverseWriter(System.out,new NullConvStrategy()));
    copy.toOutput(new LineStraightWriter(System.out,new UpperCaseConvStrategy()));
  }
}
```

Informazioni supplementari

Arriviamo ora alla richiesta di informazioni supplementari, quelle che in precedenza abbiamo chiamato "statistiche". Vogliamo poter calcolare e poi visualizzare quante lettere "a" ci sono nel testo del file. La richiesta è semplice, ma conosciamo i nostri committenti… Chi chiede qualcosa del genere, un attimo dopo desidera sapere anche quante "b" ci sono, quante "c", quante "a" e "d", quante "x", "y" e "z" (letteralmente…) e tutte le combinazioni possibili. Non solo, ma la statistica fa parte dell'output; quindi a un output si desidera combinare l'informazione sulle "a", a un altro le informazioni sulle "b" e "c" insieme, e così via.

E quando si è saturi con le informazioni riguardanti le singole lettere? Ci sarà qualcos'altro da aggiungere all'output. Se non arrivasse la prima richiesta non sarebbe necessario prevedere niente di tutto ciò. Ma visto che la richiesta di statistica sulle "a" è arrivata e che quindi in ogni caso il codice deve essere fattorizzato, tanto vale prepararlo a un certo grado di flessibilità.

Si tratta quindi di trovare un meccanismo sufficientemente dinamico per poter prevedere aggiunte di questo tipo, senza dover utilizzare l'ereditarietà, che assocerebbe in modo statico un tipo di informazione a un certo output. L'ereditarietà, inoltre, per la sua poca dinamicità, ci obbligherebbe a creare una classe per ogni combinazione diversa, obbligandoci a prevedere tutte le possibili combinazioni, aumentando nel contempo a dismisura il numero di classi del programma.

Cerchiamo di formalizzare meglio le richieste. Vogliamo avere la possibilità di aggiungere all'output una o più statistiche, in modo dinamico, potendo scegliere combinazioni diverse per output diversi. Il fatto di avere o meno la statistica deve essere trasparente, in altre parole la scelta deve essere fatta in un solo luogo del programma (meglio se centralmente, quindi in `Reader`, dove già vengono prese tutte le decisioni) senza conseguenze nel resto del codice.

Visto che vogliamo evitare l'ereditarietà, per avere maggiore dinamicità ed evitare un'inutile frammentazione di sottoclassi statiche, il pattern che ci viene in aiuto per realizzare questo concetto generale di estensione è Decorator.

Decorator

Il pattern Decorator serve per l'estensione di singoli oggetti (non classi) con nuove funzionalità.

Utilizzo

Quando si desidera aggiungere o togliere funzionalità in modo dinamico; quando l'utilizzo dell'ereditarietà porterebbe a un numero troppo elevato di classi; quando è utile poter combinare più oggetti Decorator in modo ricorsivo. In figura 12.11 è riportato il diagramma di classi originale per questo pattern.

La struttura è interessante, perché mostra in modo combinato le potenzialità di composizione e di polimorfismo. Se osserviamo la classe astratta `Decorator`, possiamo notare come questa erediti da `Component` e allo stesso tempo abbia un riferimento a un oggetto di tipo `Component`. Questi legami hanno due scopi ben distinti.

Nel primo caso (ereditarietà) si tratta di creare un legame polimorfico tra `Component` e `Decorator`, compresi, evidentemente, tutti i discendenti di quest'ultimo. Questo legame ha quale obiettivo la trasparenza. In altre parole, a una variabile di tipo `Component` possiamo associare senza problemi un elemento di tipo `DecoratorA`. Oppure, ancora, a un metodo che si aspetti un oggetto di tipo `Component` possiamo passare un oggetto di tipo `DecoratorA`.

Conseguenze

Queste sono le conseguenze dell'utilizzo del pattern Decorator in un caso come il nostro:

- Le responsabilità vengono passate dinamicamente da un oggetto a un altro, cioè dal decoratore all'oggetto decorato ("favor class composition over class inheritance").
- Sono possibili combinazioni ricorsive. Questo significa che non abbiamo limiti nel combinare decoratori uno dentro l'altro. In questo modo evitiamo anche l'esplosione del numero di classi che diventerebbero necessarie utilizzando il meccanismo di ereditarietà.

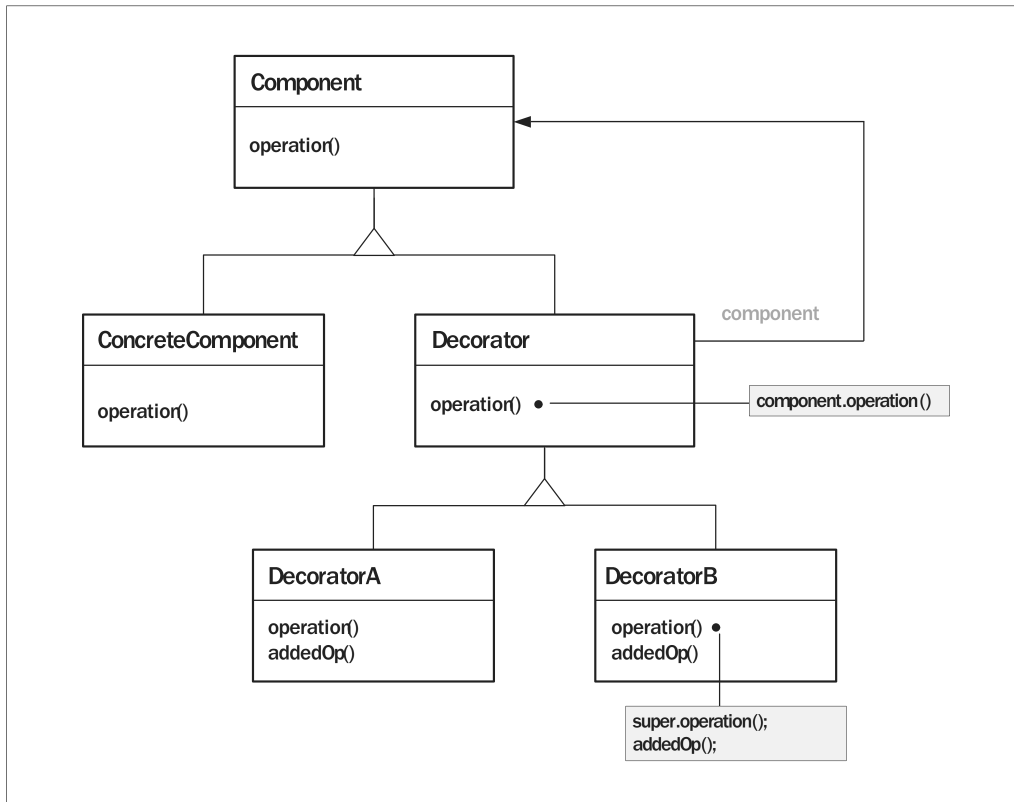

Figura 12.11 – Schema di classi per Decorator.

- Non abbiamo inoltre bisogno di prevedere una superclasse con molte funzionalità (principio chiamato "pay as you go", un "pedaggio progressivo", nel senso che si possono facilmente aggiungere funzionalità al momento del bisogno).

Ecco, in figura 12.12, la realizzazione del design per il nostro caso.

La classe LineWriterDecorator è una classe astratta che implementa la relazione esistente tra decoratore e LineWriter. Ecco la sua realizzazione.

```
abstract class LineWriterDecorator implements LineWriter
{
  protected LineWriter fLWriter;

  LineWriterDecorator(LineWriter lw){
    fLWriter = lw;
```

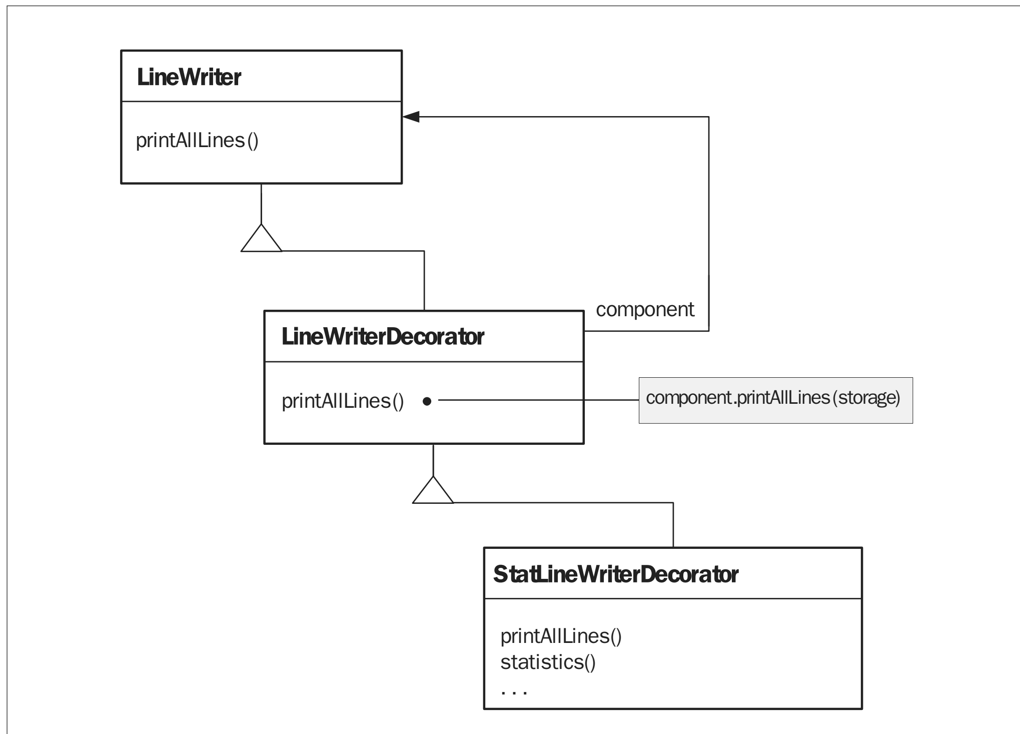

Figura 12.12 – Decorator nel contesto del problema.

```
    }

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

    public void printOneLine(String line){
      fLWriter.printOneLine(line);
    }

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

La classe StatLineDecorator introduce invece in modo concreto la modifica richiesta all'output. In particolare aggiunge la lettura del testo con relativa conta delle occorrenze di una lettera (specificata come parametro nel costruttore) e l'aggiunta della chiamata "decorativa" makeStatistics() all'interno di printAllLines().

Il metodo *protected* makeStatistics() ha il compito di contare le occorrente della lettera scelta e visualizzare in output l'informazione desiderata.

```
public class StatLineWriterDecorator extends LineWriterDecorator
{
   char fToFind;

   public StatLineWriterDecorator(LineWriter lw, char ch){
      super(lw);
      fToFind = ch;
   }

   protected void makeStatistics(List storage){
      …
   }

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

Vediamo allora come viene implementata la scelta in Reader, la classe principale. La soluzione presentata sceglie di mostrare l'output con due decoratori: uno che conta le occorrenze della lettera "a" e uno che conta le occorrenze della lettera "b".

Di nuovo abbiamo ridotto le modifiche a semplici aggiunte di classi, oltre, naturalmente, alle scelte effettuate nella classe principale.

```
public class Reader
{
   public static void main (String args[]){
      String fileName=args[0];
      Copy cp = new Copy(new LineReader(fileName));
      LineWriter lw = new LineReverseWriter(System.out,new UpperCaseConvStrategy());
      cp.toOutput(new StatLineWriterDecorator(new StatLineWriterDecorator(lw,'a'), 'b'));
   }
}
```

Diversi canali di output

Se torniamo alle richieste di modifica dei requisiti, possiamo notare come sia rimasto ancora un punto in sospeso: la possibilità di inviare i risultati verso più canali di output. In verità la prima soluzione a questo problema era semplice. Abbiamo introdotto la possibilità di stampare i dati anche su un file, prevedendo di fatto due stream di output (standard output e file). Le modifiche hanno però avuto conseguenze sull'intero codice.

Vogliamo adesso raggiungere due obiettivi: fare in modo che le modifiche possano essere centralizzate in un unico luogo (classe principale) e generalizzare la funzionalità richiesta rendendo possibile l'aggiunta di un numero indefinito di canali di output.

Ripensando al problema, ci accorgiamo che dobbiamo gestire una relazione "da uno a molti" tra un oggetto responsabile di raccogliere il messaggio di stampa e i diversi canali di output eventualmente coinvolti. Esiste un pattern, molto utilizzato, che serve proprio a definire questo tipo di dipendenza: si tratta del pattern Observer.

Observer

Scopo del pattern Observer è quello di definire una relazione *1-m* tra oggetti, in modo che quando uno di essi modifica il suo stato, gli altri ne vengono informati.

Utilizzo

Quando la modifica di un oggetto rende necessarie le modifiche di altri, ma non si sa quanti sono gli altri; quando la dipendenza deve essere dinamica, cioè quando un oggetto deve notificare i suoi cambiamenti, senza dover necessariamente conoscere gli altri oggetti.

Vediamo in figura 12.13 come si presenta il diagramma delle classi.

C'è un elemento di tipo `Subject` che è in relazione con più elementi di tipo `Observer`. Questa relazione viene solitamente gestita con una lista all'interno di `Subject`. Si tratta di una relazione Model-View, in cui il "modello" tiene informate le sue "viste" appena viene modificato.

Anche gli elementi `Observer` possono eventualmente accedere a `Subject`, o con una relazione di composizione o con un riferimento ricevuto attraverso la chiamata a `update()`, metodo di interfaccia di `Observer`.

Conseguenze

Queste sono le conseguenze dell'utilizzo di Observer:

- `Observer` e `Subject` possono essere modificati in modo indipendente. La loro relazione è di tipo *has-a* ("favor object composition over class inheritance").

- Nuovi elementi di tipo `Observer` possono essere aggiunti dinamicamente, senza conseguenze per l'elemento `Subject`.

- Gli oggetti di tipo `Observer` possono essere parte di qualsiasi gerarchia di classe.

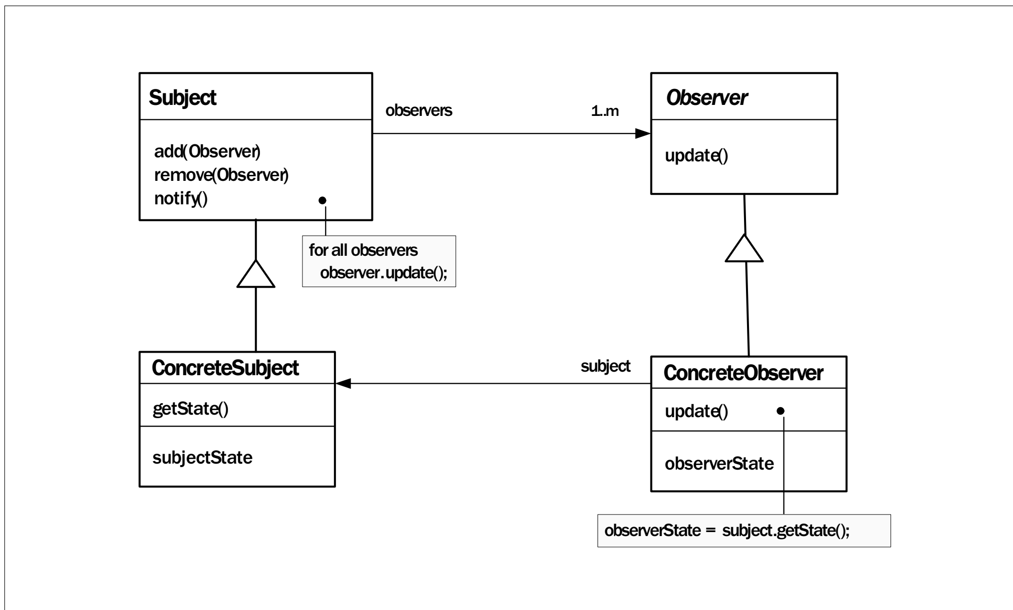

Figura 12.13 – Schema di classi per Observer.

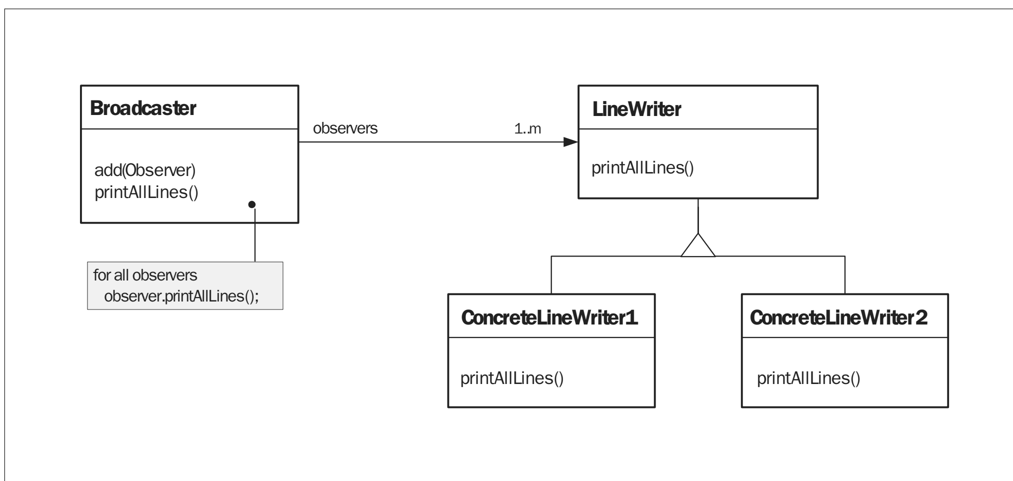

Figura 12.14 – Observer nel contesto del problema.

- Il modello supporta l'idea di broadcasting (limitato agli elementi che l'elemento Subject gestisce nella sua lista). Del resto questa idea è intrinseca nella relazione Model-View, modellata dal pattern Observer.

Nel nostro caso dobbiamo unicamente sfruttare la relazione da Subject a Observer, senza bisogno del riferimento all'indietro. L'architettura risulta quindi semplificata. In figura 12.14 è riportato lo schema di classi relativo al nostro caso. Abbiamo definito una classe Broadcaster che ha il compito di gestire la presenza di più canali di output. Quando all'oggetto Broadcaster arriva la richiesta printAllLines(), esegue un'iterazione sui canali di output da lui gestiti e a ognuno manda la richiesta printAllLines(). Di Observer sfruttiamo quindi essenzialmente il meccanismo implicito di moltiplicazione di un messaggio da uno a molti.

Polimorfismo

Al momento di implementare la soluzione c'è un altro aspetto interessante che può essere considerato. Finora tutte le richieste printAllLines() vengono mandate dall'oggetto Copy a un oggetto di tipo LineWriter. Questo oggetto viene passato a Copy dal programma principale attraverso il suo metodo toOutput(), come mostrato dall'implementazione di Copy qui riportata.

```
public class Copy
{
  protected List storage;

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

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

Se vogliamo introdurre l'oggetto Broadcaster come elemento intermedio tra Copy e i diversi canali di output rappresentati da LineWriter, senza per questo dover intervenire nel codice, la cosa più semplice e pulita è quella di sfruttare il polimorfismo. Come? Definendo Broadcaster come classe di tipo LineWriter, cioè classe che implementa l'interfaccia LineWriter. In questo modo non cambia niente per Copy se l'oggetto passatogli come parametro di toOutput() è un oggetto LineWriter che manda subito in output il contenuto di storage, oppure è un oggetto Broadcaster che effettua un ciclo sugli elementi LineWriter da esso gestiti.

Vediamo l'estratto dell'implementazione di Broadcaster che ci interessa.

```
public class Broadcaster implements LineWriter
{
  protected List observers = new ArrayList();

  public void addPrintStream(LineWriter stream) {
    observers.add(stream);
  }

  …

  public void printAllLines(List storage) {
    Iterator iter = storage.getIterator();
    while (iter.hasMoreElements()){
      LineWriter lw = (LineWriter)iter.nextElement();
      lw.printAllLines(storage);
    }
  }
}
```

E ora vediamo come si presenta il programma principale, considerando di specificare tre canali di output: uno che va in standard output e due su file.

```
public class Reader
{
  public static void main (String args[]){
    String filename = args[0];
    String outFile1= fileName + ".copy1", outFile2= fileName + ".copy2";
    Broadcaster bc = new BroadCaster();
    LineWriter lw0 = new LineReverseWriter(System.out,new UpperCaseConvStrategy());
    LineWriter lw1 = new LineReverseWriter(outFile1,new NullConvStrategy());
    LineWriter lw2 = new LineStraightWriter(outFile2,new NullConvStrategy());
    bc.addPrintStream(lw0);
    bc.addPrintStream(lw1);
    bc.addPrintStream(new StatLineWriterDecorator(new StatLineWriterDecorator(lw2,'a'), 'b'));
    Copy cp = new Copy(new LineReader(fileName));
    cp.toOutput(bc);
  }
}
```

Come si vede, al metodo toOutput() viene semplicemente passato l'oggetto Broadcaster, a cui vengono precedentemente aggiunti i canali di output in forma di LineWriter. La trasformazione da chiamata singola a printAllLines() a chiamate multiple avviene in modo del tutto trasparente in Broadcaster.

In questo capitolo…

Questo lungo capitolo è servito per mostrare come aggiungere man mano nuovi elementi di design all'interno di un programma. L'esempio iniziale era semplice, ma già le prime richieste di modifica hanno mostrato che c'era un forte rischio di effettuare modifiche *ad hoc*, senza integrarle in un concetto più generale. Anche un problema così banale può generare soluzioni complesse, se non si tiene il design sotto controllo.

Abbiamo così cercato di intervenire nel design, generalizzando nel contempo i requisiti del problema. Ne è scaturito un programma più flessibile e con una funzionalità più generica rispetto ai requisiti. Introducendo i pattern nell'architettura, abbiamo inoltre creato elementi documentati, con un funzionamento chiaro, senza necessità di ulteriori spiegazioni nel codice.