# Capitolo 10
# Utilizzo di un pattern: Iterator

In questo capitolo mostriamo con un esempio l'utilità del pattern Iterator. Tale pattern è stato utilizzato sistematicamente nel collection framework di Java, fornito a partire dalla versione 1.2.

## Situazione iniziale

Partiamo da alcuni estratti di codice e vediamo come migliorarlo. Supponiamo di dover organizzare in un elenco un certo tipo di informazione e di dover iterare sul contenuto della lista stessa per poter mostrare a video il valore di ogni singolo elemento. Se la lista in questione è organizzata sottoforma di array, la parte di codice riferita all'iterazione si presenterà come segue:

```
for(int i=0; i < lista.length; i++){
   System.out.println(lista[i]);
}
```

Supponiamo invece di organizzare gli stessi elementi in una lista dinamica semplice, come quelle che si impara a implementare nei primi corsi di programmazione, in cui una cella è

rappresentata da un contenitore (`Node`) con puntatore alla cella successiva. L'iterazione sulla lista sarà qualcosa di simile:

```
Node node = list.first();
while(node.next() != null){
  System.out.println(node.value());
  node = node.next();
}
```

Entrambi i metodi di iterazione, con array e con lista semplice, hanno alcuni svantaggi che vanno discussi.

In primo luogo c'è un'esposizione completa dell'implementazione della lista. Questa esposizione causa dipendenze in caso di modifiche nell'implementazione della lista stessa. Oltre a ciò, un cambiamento del tipo di iterazione provocherebbe modifiche in tutte le parti di codice in cui l'iterazione viene utilizzata.

# Prime modifiche

Proviamo ad apportare alcune modifiche che ci permettano almeno di rendere indipendente l'oggetto client dall'implementazione della lista. Si tratta prima di tutto di racchiudere la lista all'interno di una classe, definendo un'interfaccia che rappresenti l'iterazione sulla lista, senza esporne l'implementazione. Ecco una possibile definizione:

```
void rewind();
Object nextElement();
boolean hasMoreElements();
```

Il primo metodo serve a resettare la posizione dell'iteratore, il secondo a restituire l'elemento successivo e spostarsi di una posizione in avanti, il terzo serve invece a stabilire se ci troviamo alla fine della lista. Un possibile utilizzo è in questo estratto di codice:

```
list.rewind();
while(list.hasMoreElements()){
  System.out.println(list.nextElement());
}
```

## Discussione

Le prime modifiche apportate hanno raggiunto lo scopo iniziale: separare il codice di utilizzo da quello di implementazione della lista. Abbiamo inoltre creato un'astrazione che permette, oltre a modifiche nell'implementazione, di apportare cambiamenti anche nella semantica dell'iterazione, come ad esempio definire un iteratore che scansioni la lista in senso inverso.

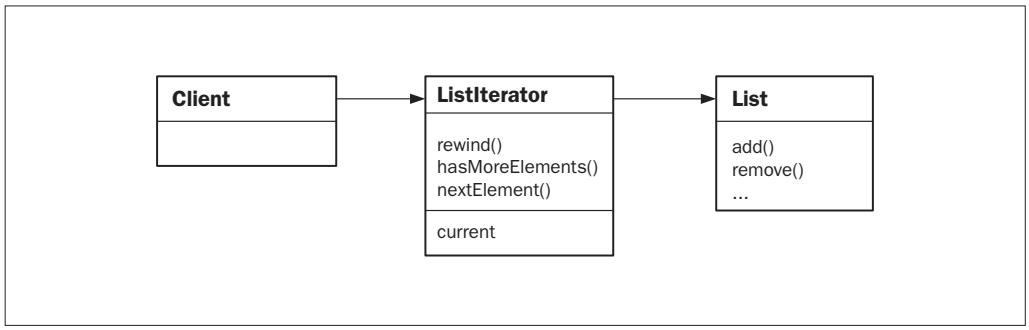


*Fig10.1 – Struttura dell'iteratore.*

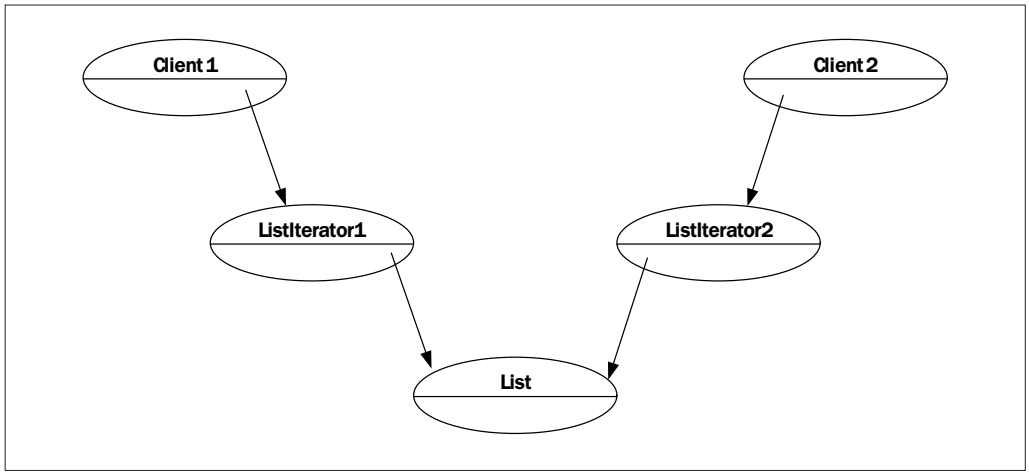


*Figura 10.2 – Schema a runtime del nuovo iteratore.*

Tutto questo ha però anche introdotto uno svantaggio piuttosto evidente: la gestione dell'iteratore all'interno della lista ne limita l'utilizzo, perché non permette più iterazioni parallele della stessa lista. Non è possibile cioè avere due oggetti che scansionano la stessa lista, perché le iterazioni di uno influenzerebbero le iterazioni del secondo.

# Secondo passaggio

Lo scopo principale di questo secondo passaggio consiste nel separare la gestione dell'elemento corrente di iterazione dalla lista dalla quale viene letto. In altre parole, si tratta di rende-

re possibili più iterazioni contemporaneamente su un'unica lista. Naturalmente vogliamo mantenere i vantaggi ottenuti in precedenza, soprattutto l'indipendenza tra lista e client.

La realizzazione prevede una classe intermedia tra client e lista, in grado di gestire l'elemento corrente e implementare l'interfaccia definita in precedenza. Si potrà notare (figura 10.1) come lo schema sia molto simile all'esempio visto nella parte I del libro, in cui la classe Noleggio si inseriva come un cuscinetto tra due elementi. Ecco un possibile utilizzo:

```
ListIterator iter = new ListIterator(list);
iter.rewind();
while(iter.hasMoreElements()){
  System.out.println(iter.nextElement());
}
```

Abbiamo in questo modo mantenuto la separazione e allo stesso tempo implementato la possibilità di iterare con più oggetti sulla stessa lista, come mostrato in figura 10.2

# Terzo passaggio: utilizzo polimorfico

A questo punto possiamo andare avanti con la generalizzazione del concetto di iteratore provando a implementare la possibilità di scambiarsi iteratori diversi all'interno dello stesso codice. Si tratta di sfruttare il polimorfismo, creando una gerarchia di iteratori, in cui un'interfaccia comune venga definita da una classe astratta o da un elemento interface iteratore, come mostrato in figura 10.3. Nel caso specifico si tratta di introdurre un elemento astratto Iterator nello schema precedente. Ecco un possibile utilizzo:

```
Iterator iter = new ListIterator(list);
iter.rewind();
while(iter.hasMoreElements()){
  System.out.println(iter.nextElement());
}
```

La differenza rispetto al codice precedente è minima, ma importante. La variabile iter non viene dichiarata della stessa classe dell'iteratore concreto utilizzato, bensì di un tipo astratto, in comune con altri iteratori. In questo modo solo la creazione dell'oggetto iteratore andrebbe modificata, in caso di sostituzione dell'iteratore stesso. L'utilizzo del pattern Iterator è tutto qui. I vantaggi principali che abbiamo ottenuto sono:

- Possibilità di modifiche nell'elemento aggregato, senza ripercussioni nel codice client.
- Semplificazione dell'interfaccia tra elemento client e lista. In effetti l'iteratore fa da mediatore tra client e lista, funge cioè da interfaccia verso il client e si occupa di accedere per esso agli elementi della lista.

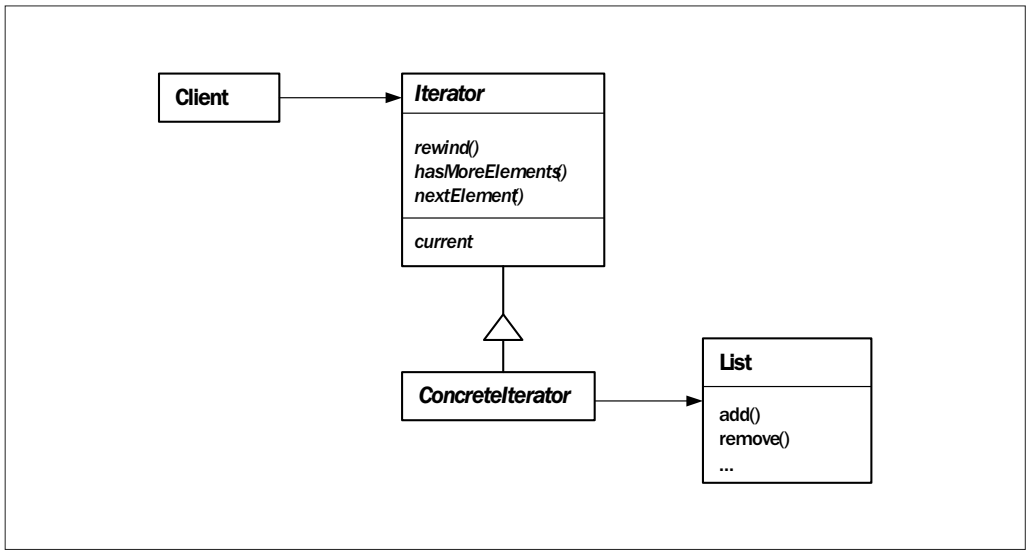


*Figura10.3 – Iteratore con interface.*

- Possibilità di avere più iterazioni simultanee sulla stessa lista.

# Quarto passaggio: metodo factory

A questo punto diventa spontaneo un quarto passaggio, che ci permette di associare un iteratore specifico a ogni tipo di lista, permettendo al codice di raggiungere un ulteriore grado di genericità. Per ottenere questo, dobbiamo rendere polimorfico non solo l'iteratore, ma anche l'elemento aggregato, cioè la lista (figura 10.4).

Se ogni tipo di lista ha un suo iteratore specifico, la creazione dell'oggetto iteratore può essere delegata alla lista stessa. Questo è il principio di un altro pattern: il pattern Factory Method. Il suo scopo consiste nell'isolare la creazione di un oggetto all'interno di un metodo. Questo permette di specificare la creazione di oggetti diversi, con chiamate uguali. L'oggetto factory chiamante definisce quale oggetto "prodotto" verrà creato.

L'utilizzo di questa idea in un altro pattern (Abstract factory) porta alla definizione di un'interfaccia per la creazione di un oggetto, lasciando poi alle sottoclassi l'implementazione della creazione. Nel nostro esempio questo significa che la lista sa quale iteratore deve creare per permettere al client un'iterazione "standard" su sé stessa. Scelta la lista, non è quindi più necessario scegliere l'iteratore, perché questo viene creato dal metodo factory della lista:

```
Iterator iter = list.createIterator();
iter.rewind();
```

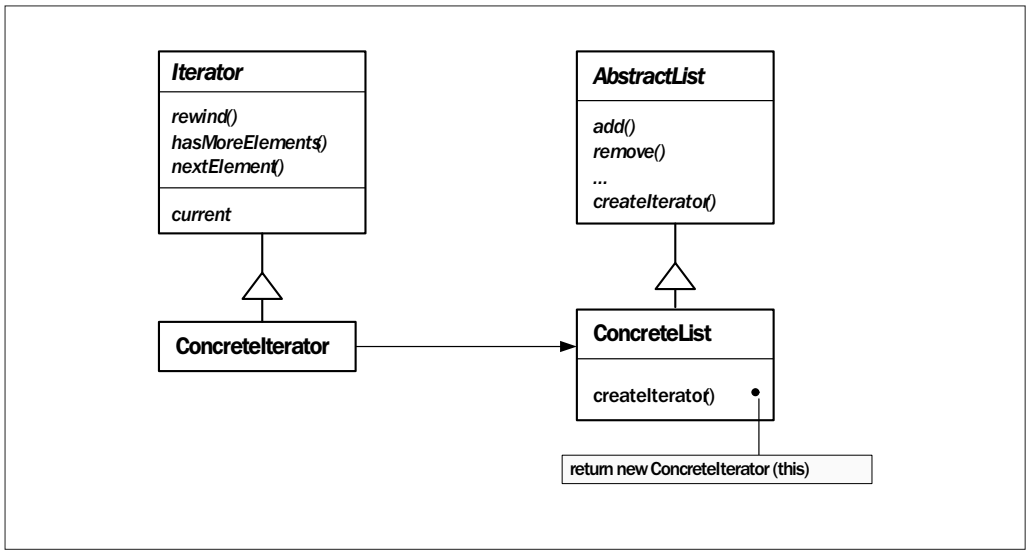


*Figura 10.4 – Lista con metodo factory.*

```
while(iter.hasMoreElements()){
  System.out.println(iter.nextElement());
}
```

Questo meccanismo è stato ripreso da Java nel Collection framework. Ogni lista risponde a un messaggio iterator(), restituendo un oggetto di tipo Iterator, che permette un'iterazione standard sulla lista stessa.

# In questo capitolo…

In questo capitolo siamo partiti da una situazione molto semplice e tipica per capire la necessità e l'utilità dell'introduzione di un'architettura più generica e più flessibile. L'architettura viene realizzata con il pattern Iterator.

Abbiamo poi esteso l'architettura di partenza, mostrando quali possano essere altri elementi di genericità.