Capitolo 4
Fattorizzare per il test

Sembra una piccolezza, ma se osserviamo meglio il contenuto del metodo resoconto() ci accorgiamo che ci sono due calcoli sovrapposti: l'ammontare totale e l'accumulo di punti.

In entrambi i casi l'estrazione in un metodo separato da resoconto() permette di isolare due nuove funzionalità semplici e facilmente verificabili con metodi di test: calcolaImportoTotale() e calcolaPuntiTotale(). Inoltre non dimentichiamoci che lo scopo finale è quello di estrarre da resoconto() il massimo della logica, per permettere di implementare metodi resocontoXML() o resocontoHTML() che devono duplicare il meno possibile elementi di resoconto().

A questo possiamo anche aggiungere la volontà di creare nel programma due nuove funzionalità, a disposizione di altri oggetti.

Estrazione dei metodi

Ecco come si presentano i due nuovi metodi, rimasti nella classe Cliente, con la nuova versione di resoconto():

```
public class Cliente
{
  ...
  double calcolaImportoTotale(){
    double result = 0;
    Iterator iterNoleggi = getNoleggi().iterator();
```

```
        while(iterNoleggi.hasNext()){
            Noleggio ogniNoleggio = (Noleggio)iterNoleggi.next();
            result += ogniNoleggio.calcolaImporto();
        }
        return result;
    }

    int calcolaPuntiTotale(){
        int result = 0;
        Iterator iterNoleggi = getNoleggi().iterator();
        while(iterNoleggi.hasNext()){
            Noleggio ogniNoleggio = (Noleggio)iterNoleggi.next();
            result += ogniNoleggio.calcolaPunti();
        }
        return result;
    }

    public String resoconto(){
        String testo = "Resoconto di noleggio per " + getNome() + "\n";
        Iterator iterNoleggi = getNoleggi().iterator();
        while(iterNoleggi.hasNext()){
            Noleggio ogniNoleggio = (Noleggio)iterNoleggi.next();
            testo += "\t" + ogniNoleggio.getSci().getModello() + "\t" + ogniNoleggio.calcolaImporto() + "\n";
        }
        testo += "Ammontare dovuto:\t Fr. " + calcolaImportoTotale() + "\n";
        testo += "Ha guadagnato " + calcolaPuntiTotale()+ " punti di bonus\n";
        return testo;
    }
}
```

I due metodi calcolaPuntiTotale() e calcolaImportoTotale() contengono ognuno 6 righe di istruzione (quindi 12 in totale) per eliminarne complessivamente 4 all'interno di resoconto(). Questo significa che il programma è cresciuto di 8 righe. Questo svantaggio è comunque da considerare minimo, rispetto al vantaggio di estrarre i due metodi, semplificando resoconto(), mettendo a disposizione due metodi utili in più ai potenziali utenti di Cliente e, soprattutto, rendendo la funzionalità totale di Cliente meglio verificabile con test.

Va inoltre detto che questo svantaggio dipende unicamente dal linguaggio di programmazione. Non è colpa nostra se in Java sono necessarie tutte queste linee di codice per un ciclo…

Siamo ora in grado di creare un nuovo metodo resoconto() che restituisca la stessa informazione utilizzando un formato diverso. Non si tratta ancora della soluzione migliore, perché, come vedremo più avanti, ci sono ancora alcuni aspetti di questo metodo che potrebbero essere estratti e richiamati nella sua versione per un nuovo formato, ad esempio resocontoHtml(), senza dover essere duplicati.

Lo scopo di questi primi passaggi era però quello di dettare il ritmo delle modifiche, mostrando come procedere con i cicli di refactoring e quali aspetti considerare.

Ecco allora la versione attuale del metodo resocontoHtml():

```
public String resocontoHtml(){
   String testo = "<H1>Resoconto di noleggio per " + getNome()+"</H1>\n";
   Iterator iterNoleggi = getNoleggi().iterator();
   while(iterNoleggi.hasNext()){
      Noleggio ogniNoleggio = (Noleggio)iterNoleggi.next();
      testo += "\t" + ogniNoleggio.getSci().getModello() + "\t" + ogniNoleggio.calcolaImporto() + "<BR>\n";
   }
   testo += "<P>Ammontare dovuto:\t Fr. " + calcolaImportoTotale() + "</P>\n";
   testo += "<P>Ha guadagnato <EM>"+ calcolaPuntiTotale()+"</EM>punti di bonus</P>\n";
   return testo;
}
```

Ed ecco, in figura 4.1, l'architettura attuale del programma.

Problemi di performance

Almeno due parole vanno spese sui problemi di prestazione, veri o presunti, legati alla fattorizzazione del codice, soprattutto dopo le ultime modifiche.

Abbiamo estratto due funzionalità contenute in un while e le abbiamo isolate, ognuna con il suo ciclo, lasciando quello da cui sono state derivate. Abbiamo in altre parole triplicato il numero dei cicli (anche se è diminuito il contenuto di elaborazione da eseguire per ogni ciclo). Inoltre abbiamo raddoppiato le chiamate a calcolaImporto(), che ora viene chiamato sia nel ciclo all'interno di resoconto() per mostrare i totali parziali, sia in calcolaImportoTotale().

Questi passaggi di refactoring devono essere portati a termine senza considerare eventuali problemi di velocità. Lo scopo del refactoring è quello di migliorare il codice. Se poi ci sono problemi, li si va a cercare in seguito, in modo mirato.

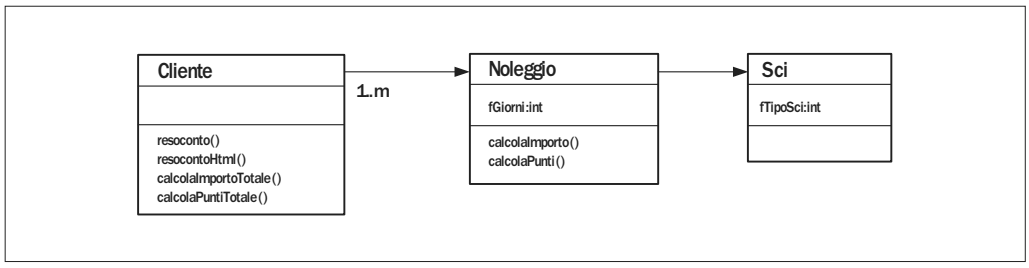

Figura 4.1 – Struttura attuale.

Come facciamo a dire se abbiamo una reale perdita di performance?

C'è un unico modo per saperlo: misurare i tempi con un *profiler*. A meno che ci sia una chiara differenza di algoritmo a livello asintotico (ecco a cosa serve studiare gli algoritmi di sort…), non c'è veramente altro modo per decidere se eseguire ottimizzazioni nel codice.

Studio degli algoritmi di sort

Durante gli anni di insegnamento, mi è capitato più volte di dover tenere corsi di programmazione ai primi semestri, dove si introduce il primo linguaggio e dove si insegnano i fondamenti di informatica. In quel tipo di corsi, tra docenti, ci si chiede spesso quali argomenti trattare e quali invece non considerare, soprattutto pensando a ciò che risulterà effettivamente necessario ai futuri laureati nel mondo del lavoro.

Ebbene, mi è capitato di sentire che gli algoritmi di sort dovrebbero essere bellamente eliminati dal contenuto di un corso di fondamenti di programmazione, con la motivazione che nessuno avrà mai la necessità di implementare un algoritmo simile. A parte un'eventuale obiezione a quest'ultimo punto, gli algoritmi di sort vanno assolutamente insegnati non tanto per la loro funzionalità intrinseca, quanto per la loro utilità nel capire le strutture di algoritmi e nel poter studiare e confrontare i costi asintotici degli stessi.

L'argomento dei costi asintotici non è dei più semplici, per chi inizia a programmare. A maggior ragione va affrontato con algoritmi facili, corti, conosciuti e confrontabili tra di loro.

È inutile metter mano nel codice, complicandolo e rendendolo illeggibile con la pretesa di renderlo più efficiente, se non ci si basa su misurazioni certe. Soprattutto dopo che abbiamo fatto il possibile per renderlo leggibile…

Ho imparato, anche a mie spese, che in molti casi i problemi di performance risiedono in una parte molto limitata del codice, spesso anche solo in un paio di istruzioni. Non è quindi sensato mettersi a modificare il codice prima di aver identificato chiaramente il vero problema: da un punto di vista metodologico occorre sempre agire cercando di scoprire le effettive cause di lentezza del programma.

Regole di ottimizzazione

Nel 1974, Donald E. Knuth scriveva: "Non dobbiamo preoccuparci di efficienza in almeno il 97% dei casi. L'ottimizzazione prematura è la causa di tutti i guai".

L'anno successivo, M. A Jackson pubblicava le sue due regole sull'ottimizzazione:

- Regola 1: non ottimizzare.
- Regola 2 (solo per gli esperti): non ottimizzare ancora, almeno fino a quando non disponi di una soluzione non ottimizzata e perfettamente chiara [Bloch-2001].

L'altra cosa che deve essere detta è che la performance va misurata quando il problema è reale. Nel nostro caso, anche triplicare il numero di volte in cui il contenuto di un ciclo `while` viene eseguito, se questo poi non avrà mai più di una decina di elementi, non avrà certamente ripercussioni sulla velocità dell'applicazione. È vero che in generale in sistemi real time la gestione della performance è più importante e critica che in sistemi gestionali, ma in entrambi è fondamentale la leggibilità del codice, che permette una migliore manutenzione.

Se per un paio di millisecondi di miglioramento di velocità di elaborazione impiego poi due giorni invece di un'ora per effettuare una modifica, la scelta è presto fatta… In ogni caso, diffidate degli esperti che vi dicono come strutturare il codice in modo che sia più efficiente: misurate la performance in modo oggettivo con i programmi di utilità a disposizione; poi prendete le giuste misure, se necessario.

Variabili locali e variabili globali

Sempre nell'ambito dell'insegnamento del primo linguaggio di programmazione, in uno dei miei primi anni di docenza, durante un semestre, per aiutarmi a gestire il grosso numero di studenti durante le esercitazioni in laboratorio, mi venne affiancato un collega con funzione di assistente. Arrivavo da alcuni anni di esperienza di programmazione funzionale in Lisp, e sviluppo a oggetti in CLOS. I due paradigmi hanno in comune la parte più essenziale della programmazione: modularità e concetto puro di funzione, che a livello di C (il linguaggio che si insegnava) per iniziati si può tradurre banalmente in "utilizzare il più possibile funzioni e variabili locali, a scapito di quelle globali".

Quello delle variabili locali era però un concetto che non veniva assorbito. Lo spiegavo ripetutamente a lezione, ma poi mi accorgevo che gli esercizi venivano risolti utilizzando pochissimo le variabili locali; anzi, la tendenza era quella di non utilizzarle per niente, dichiarando tutto globale, dalle variabili dei cicli agli elementi usati solo all'interno di poche funzioni. Così decisi di organizzare una lezione di teoria unicamente sui problemi generati dall'utilizzo di variabili globali. Dopo aver mostrato alcuni casi di errori ed effetti collaterali difficili da scoprire, la mia sorpresa fu notevole quando uno studente, timidamente, alzò la mano e mi spiegò che tutti usavano solo variabili globali perché cosí aveva spiegato loro, a mia insaputa, lo zelante collega, argomentando che quello era l'unico modo per scrivere programmi veramente efficienti e che lui aveva esperienza a sufficienza per garantire che quello era il metodo giusto…

In questo capitolo…

In questo capitolo abbiamo di nuovo estratto metodi da `resoconto()`. Quest'ultimo è stato così ulteriormente semplificato, permettendoci nel contempo di aggiungere nuova funzionalità alla classe, modificata, che ora può offrire due nuovi "servizi" all'esterno: `calcolaImportoTotale()` e `calcolaPuntiTotale()`.

È stato inoltre fatto notare come operazioni di questo tipo possano teoricamente avere ripercussioni a livello di performance. Impariamo però a contestualizzare tale tipo di problema e, in ogni caso, prima di eseguire modifiche nel codice con la pretesa di migliorarne i valori, misuriamo se i presunti problemi esistono realmente.