Capitolo 3
Prima verifica e fattorizzazione

Non basta usare un linguaggio che preveda il concetto di classe per scrivere codice object-oriented. Questo è dimostrato anche nell'esempio appena visto. Il procedimento usato, quello cioè di identificare le classi e le relazioni esistenti tra di loro e poi iniziare la codifica, è corretto. Nel nostro esempio si potrebbero però utilizzare meglio alcune particolarità della programmazione ad oggetti per facilitare modifiche nel codice. Cosa succede, ad esempio, se si decide di introdurre una nuova classificazione degli sci? Vedremo questo più avanti.

Per ora concentriamoci sul problema più evidente, cioè il metodo `resoconto()`. Oltre alla sua eccessiva lunghezza, che richiama un'urgente fattorizzazione, è chiaro l'esagerato carico di responsabilità dato a questo metodo o a questa classe. Alcune parti di funzionalità andrebbero più facilmente delegate ad altre classi.

Come esempio, per capire la gravità della situazione, proviamo a chiederci cosa dovremmo adattare se volessimo dare la possibilità di generare un resoconto in formato HTML, oltre a quello in testo semplice.

Inoltre, proviamo a immaginare di voler scrivere un metodo di test per `resoconto()`. La cosa sarebbe possibile, ma dovremmo testare una somma di funzionalità, senza poter verificare a una a una le singole parti. Anche questo è un buon indizio (nella terminologia agile si parla di *smell*, "odore") che ci indica che il codice ha bisogno di una revisione.

E come la mettiamo con la leggibilità? È vero che in questo criterio si nasconde una buona dose di soggettività, ma da un metodo con una funzionalità piuttosto banale ci si dovrebbe aspettare un grado di leggibilità sicuramente maggiore.

Da queste osservazioni e domande risulta evidente la necessità di delegare parti di funzionalità contenute in `resoconto()`, semplificando nel contempo il metodo.

C'è però ancora un aspetto di peso da considerare nella decisione di modificare il codice: il programma funziona bene, perché modificarlo? In modo più diretto: perché andare a complicarsi la vita, rischiando di compromettere codice funzionante, per fare modifiche che non aggiungono nessuna nuova funzionalità al programma?

Ebbene qui risiede l'importanza dei test e dei metodi di refactoring. Se uno sviluppatore ha il sospetto che una parte di codice dev'essere modificata, deve poterlo fare in ogni momento, senza paura di aggiungere nuovi errori o ritrovarsi in stati inconsistenti. I test a disposizione devono essere lo specchio della correttezza del programma, devono cioè servire in ogni momento da verifica che il programma funzioni correttamente. I metodi di refactoring sono invece lo strumento per ridurre i rischi di inconsistenza nelle modifiche del codice. Parecchi sono ormai a disposizione dei più sofisticati ambienti di sviluppo, che li eseguono in modo automatico. Gli altri devono far parte, per usare il gergo sportivo, della "tecnica individuale" di ogni programmatore.

Si parlava prima di indizi ("odori") per stabilire se il codice ha bisogno di modifiche. La paura del programmatore di modificare il suo codice ha un gran brutto odore, perché significa che il programma è considerato poco gestibile e quindi è molto vicino alla sua fine.

Modificabilità di un programma

Come docente, il modo migliore per crearsi nemici tra gli studenti dei primi anni è quello di non considerare come corretto del codice "semplicemente" funzionante. Da studenti non si è abituati a prevedere di dover gestire il codice. Quando ci si trova di fronte a un problema, si va diritti alla soluzione, senza preoccuparsi di come si scrive il programma.

La realtà del software è però diversa: un programma corretto, ma scritto male avrà vita molto più corta di un programma non corretto, ma scritto bene. Il primo verrà abbandonato alla prima richiesta di modifica, il secondo potrà evolvere, dapprima verso una versione corretta, in seguito assecondando le nuove richieste.

Delega di funzionalità

Torniamo al metodo `resoconto()`, al centro della nostra attenzione. Si parlava di fattorizzazione, per renderlo almeno più corto e leggibile. Un passaggio sicuramente utile è quello di implementare in un metodo separato il calcolo dell'ammontare per ogni singolo noleggio. Riduce la dimensione del metodo e separa una parte logica di per sè già isolata anche a livello di linee di codice. Si tratta quindi di un passaggio poco problematico.

Ecco il nuovo metodo `resoconto()`:

```
public String resoconto(){
```

```
    String testo = "Resoconto di noleggio per " + getNome() + "\n";
    double totale = 0;
    int puntiAccumulati = 0;
    Iterator iterNoleggi = fNoleggi.iterator();
    while(iterNoleggi.hasNext()){
      Noleggio ogniNoleggio = (Noleggio)iterNoleggi.next();
      double costo = costoDi(ogniNoleggio);
      puntiAccumulati++;
      if ((ogniNoleggio.getSci().getTipo() == Sci.CARVING) &&
        (ogniNoleggio.getGiorni() > 1)){    //extra bonus per carving
          puntiAccumulati++;
      }
      testo += "\t" + ogniNoleggio.getSci().getModello() + "\t" + costo + "\n";
      totale += costo;     //accumula sul totale l'ammontare di ogni singolo noleggio
    }
    testo += "Ammontare dovuto:\t Fr. " + totale + "\n";
    testo += "Ha guadagnato " + puntiAccumulati + " punti di bonus\n";
    return testo;
}
```

Ecco lo switch usato per calcolare il costo, all'interno del nuovo metodo implementato costoDi():

```
private double costoDi(Noleggio ogniNoleggio) {
    double costo = 0;
     switch(ogniNoleggio.getSci().getTipo()){   //somma ammontare per ogni paio
      case Sci.NORMALE:     // 25 + 15 ogni giorno dopo il secondo giorno
        costo += 25;
        if (ogniNoleggio.getGiorni() > 2)
          costo += (ogniNoleggio.getGiorni() - 2) * 15;
        break;
      case Sci.CARVING: // 25 ogni giorno
        costo += ogniNoleggio.getGiorni() * 25;
        break;
      case Sci.BAMBINI:   // 15 * lunghezza in metri * num. giorni
        double base = 15 * (double)ogniNoleggio.getSci().getLunghezza()/100;
        costo += base;
        if (ogniNoleggio.getGiorni() > 3)
          costo += (ogniNoleggio.getGiorni() - 3) * base;
        break;
    }
    return costo;
}
```

Il metodo costoDi() è stato estratto direttamente dal metodo resoconto() e ha ereditato da questo anche i nomi delle variabili. Volendo guardar bene, alcuni nomi andrebbero adesso adattati al nuovo contesto.

Il parametro ogniNoleggio aveva ragione di chiamarsi in questo modo all'interno di un ciclo di iterazione, ma qui può semplicemente chiamarsi noleggio.

Discorso simile potrebbe valere per la variabile costo, che in questo nuovo contesto potrebbe chiamarsi semplicemente risultato, visto che è il risultato della chiamata a costoDi() (che poi il risultato sia in effetti un costo non fa altro che dimostrare la presenza di un certo grado di soggettività).

```
private double costoDi(Noleggio noleggio) {
  double risultato = 0;
  switch(noleggio.getSci().getTipo()){   //somma ammontare per ogni paio
    case Sci.NORMALE:     // 25 + 15 ogni giorno dopo il secondo giorno
      risultato += 25;
      if (noleggio.getGiorni() > 2)
        risultato += (noleggio.getGiorni() - 2) * 15;
      break;
    case Sci.CARVING: // 25 ogni giorno
      risultato += noleggio.getGiorni() * 25;
      break;
    case Sci.BAMBINI:   // 15 * lunghezza in metri * num. giorni
      double base = 15 * (double) noleggio.getSci().getLunghezza()/100;
      risultato += base;
     if (noleggio.getGiorni() > 3)
      risultato += (noleggio.getGiorni() - 3) * base;
    break;
  }
  return risultato;
}
```

In questo caso si è trattato di un passaggio di refactoring all'interno della stessa classe, perché il metodo estratto fa ancora parte della classe Cliente.

Guardando però più attentamente il metodo costoDi() ci si accorge che pur appartenendo a Cliente utilizza unicamente informazioni di Noleggio e Sci. Il metodo andrebbe quindi più correttamente inserito in una di queste due classi. Visto che utilizza il numero di giorni dalla classe Noleggio e da questa può in ogni caso accedere all'oggetto Sci associato, la cosa più semplice a questo punto è quella di delegare la funzionalità del metodo alla classe Noleggio (figura 3.1).

Con questo spostamento si può eliminare il parametro, perché questo diventa l'oggetto *invoker*. Inoltre è utile riadattare il nome del metodo al nuovo contesto.

```
public class Noleggio
{
```

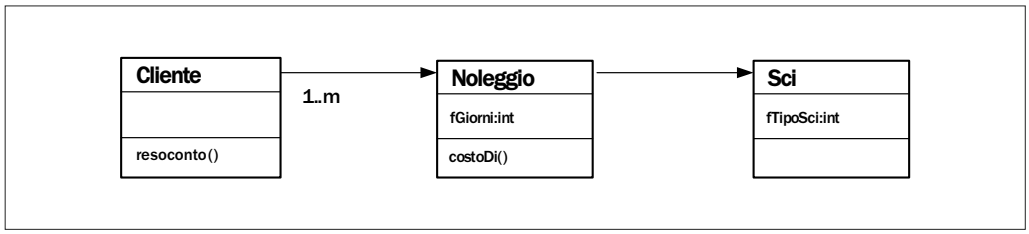

Figura 3.1 – Spostamento in Noleggio.

```
    ...
    public double calcolaImporto() {
      double risultato = 0;
      switch(getSci().getTipo()){   //somma ammontare per ogni paio
        case Sci.NORMALE:     // 25 + 15 ogni giorno dopo il secondo giorno
          risultato += 25;
          if (getGiorni() > 2)
            risultato += (getGiorni() - 2) * 15;
          break;
        case Sci.CARVING: // 25 ogni giorno
          risultato += getGiorni() * 25;
          break;
        case Sci.BAMBINI:   // 15 * lunghezza in metri * num. giorni
          double base = 15 * (double)getSci().getLunghezza()/100;
          risultato += base;
          if (getGiorni() > 3)
            risultato += (getGiorni() - 3) * base;
          break;
      }
      return risultato;
    }
}
```

Analogamente, andrà adattata la chiamata all'interno di resoconto(). Se prima la chiamata risultava essere:

```
costo = costoDi(ogniNoleggio);
```

ora diventerà questa:

```
costo = ogniNoleggio.calcolaImporto();
```

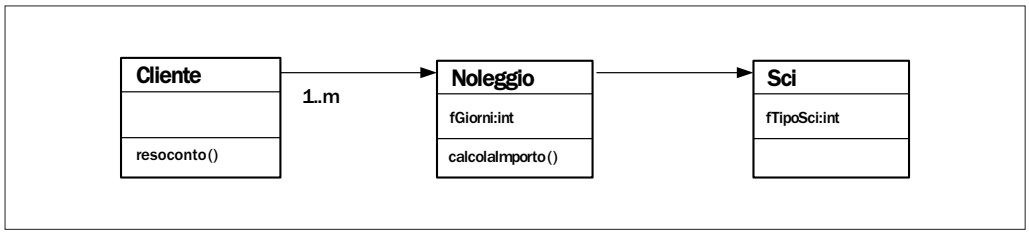

Figura 3.2 – Nuova situazione con nuovo metodo.

E in figura 3.2 viene riportata la nuova situazione nello schema di classi.

Un effetto benefico aggiuntivo di questi passaggi consiste nel fatto che ora il nostro programma ha un metodo in più, esposto dalla classe Noleggio, che può essere richiamato e sfruttato all'interno di altre classi. È sempre possibile sapere l'importo da pagare per ogni singolo noleggio.

Test

Prima di continuare con ulteriori modifiche di resoconto() vale la pena fare un'annotazione relativa al test. Non intendo più mostrare codice di test all'interno di questo esempio, perché abbiamo già visto come fare e in questa parte vogliamo concentrarci sulle modifiche del programma.

L'estrazione prima e lo spostamento poi del metodo calcolaImporto() mostra però come, oltre a fattorizzare il codice creando un metodo in più che espone una nuova funzionalità, abbiamo reso lo stesso meglio verificabile con test. Infatti per il metodo calcolaImporto() isolato dalla situazione caotica di resoconto() è ora molto semplice scrivere un corrispondente metodo di test che valuti la funzionalità e la correttezza dei risultati.

Ulteriore delega di funzionalità

Tornando ad osservare e analizzare il metodo resoconto(), ci accorgiamo che esiste una parte analoga a quella che abbiamo isolato in precedenza: il calcolo dei punti di bonus accumulati con il noleggio degli sci.

Si tratta di punti che vengono distribuiti sulla base dei giorni di noleggio e del tipo di sci. Potranno poi permettere di ricevere sconti su fatture future.

Evidenziamo la parte da estrarre da resoconto():

```
public String resoconto(){
   String testo = "Resoconto di noleggio per " + getNome() + "\n";
   double totale = 0;
   int puntiAccumulati = 0;
```

```
    Iterator iterNoleggi = fNoleggi.iterator();
    while(iterNoleggi.hasNext()){
      Noleggio ogniNoleggio = (Noleggio)iterNoleggi.next();
      double costo = ogniNoleggio.calcolaImporto();
      puntiAccumulati++;
      if ((ogniNoleggio.getSci().getTipo() == Sci.CARVING) && (ogniNoleggio.getGiorni() > 1)){
        puntiAccumulati++;   //extra bonus per carving
      }
      testo += "\t" + ogniNoleggio.getSci().getModello() + "\t" + costo + "\n";
      totale += costo;    //accumula sul totale l'ammontare di ogni noleggio
    }
    testo += "Ammontare dovuto:\t Fr. " + totale + "\n";
    testo += "Ha guadagnato " + puntiAccumulati + " punti di bonus\n";
    return testo;
  }
```

Visto che anche questi dipendono dal numero di giorni di noleggio e dal tipo di sci, significa che contengono una logica di calcolo che appartiene più propriamente alla classe Noleggio (o eventualmente alla Sci) che alla classe Cliente.

```
public class Noleggio
{
  ...

  double calcolaPunti(){
    if ((getSci().getTipo() == Sci.CARVING) &&
      (getGiorni() > 1))
    return 2;
  else
    return 1;
  }
}
```

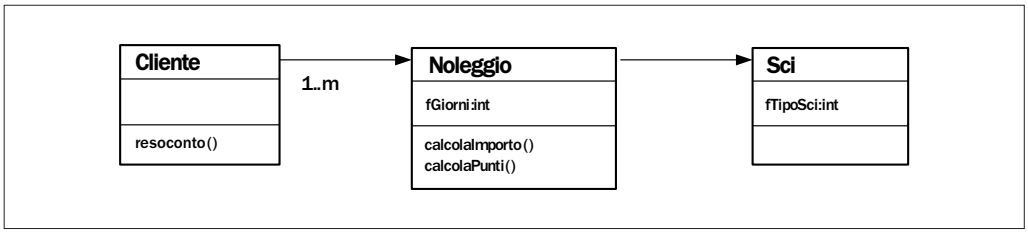

Figura 3.3 – Situazione con nuovo metodo.

Vediamo in figura 3.3 la nuova situazione dello schema di classi, con l'aggiunta del nuovo metodo. Il discorso fatto prima per calcolaImporto() a proposito della nuova funzionalità aggiunta a Noleggio e sul fatto che il metodo si presta meglio ai test, vale anche per calcolaPunti().

Ed ecco come si presenta il metodo resoconto(). Da notare che, siccome i punti vanno accumulati, si usa l'operatore += invece del semplice = usato per calcolaImporto().

```
public String resoconto(){
   String testo = "Resoconto di noleggio per " + getNome() + "\n";
   double totale = 0;
   int puntiAccumulati = 0;
   Iterator iterNoleggi = fNoleggi.iterator();
   while(iterNoleggi.hasNext()){
      Noleggio ogniNoleggio = (Noleggio)iterNoleggi.next();
      double costo = ogniNoleggio.calcolaImporto();
      puntiAccumulati+= ogniNoleggio.calcolaPunti();
      testo += "\t" + ogniNoleggio.getSci().getModello() + "\t" + costo + "\n";
      totale += costo;     //accumula sul totale l'ammontare di ogni noleggio
   }
   testo += "Ammontare dovuto:\t Fr. " + totale + "\n";
   testo += "Ha guadagnato " + puntiAccumulati + " punti di bonus\n";
   return testo;
}
```

In questo capitolo…

Partendo da un design iniziale che rappresentasse una prima soluzione al problema, abbiamo eseguito alcuni passaggi di refactoring allo scopo di suddividere meglio il codice nelle tre classi di partenza, dando ad ognuna la sua giusta responsabilità.

In particolare abbiamo estratto dal metodo "principale" resoconto() due nuovi metodi che abbiamo in seguito spostato dalla classe Cliente alla classe Noleggio, perché quella sembrava la loro posizione migliore.

Vedremo nel prossimo capitolo che la fattorizzazione non è ancora terminata.