Capitolo 6
Condivisione della sequenza

Abbiamo visto in precedenza che l'implementazione di resocontoHtml() a partire da resoconto() non è ancora la soluzione che ci si aspetterebbe.

Con questa ulteriore modifica non arriveremo ancora alla soluzione ottimale, ma avremo almeno modo di sfruttare ulteriormente la programmazione a oggetti per identificare alcune ridondanze in resoconto() e resocontoHtml() che possono essere eliminate.

Si tratta inoltre dell'occasione per introdurre già in questo capitolo un nuovo pattern, dopo Strategy, anch'esso molto semplice e vicino al modo più intuitivo di sfruttamento dell'ereditarietà: si tratta del Template Method Pattern.

Trovare gli elementi ridondanti

In che cosa consiste, qui, la ridondanza? Osservando i due metodi ci accorgiamo che tutti gli elementi di calcolo sono stati delegati a metodi e oggetti separati, quindi condivisibili. Ci sono però ancora doppioni all'interno di resoconto() nella sequenza delle chiamate delle operazioni. Queste ripetizioni di comportamento possono essere eliminate sfruttando di nuovo il meccanismo di ereditarietà.

Partendo dalla codifica di resoconto(), vediamo quali sono le operazioni che vengono chiamate con la stessa sequenza anche in resocontoHtml().

```
public class Cliente
{
  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;
  }
}
```

Possiamo identificare almeno tre operazioni chiamate con la stessa sequenza:

- un'intestazione

```
String testo = "Resoconto di noleggio per " + getNome() + "\n";
```

- un dettaglio di noleggio

```
testo += "\t" + ogniNoleggio.getSci().getModello() + "\t" + ogniNoleggio.calcolaImporto() + "\n";
```

- una chiusura

```
testo += "Ammontare dovuto:\t Fr. " + calcolaImportoTotale() + "\n";
testo += "Ha guadagnato " + calcolaPuntiTotale() + "punti di bonus\n";
```

Questi elementi sono diversi nei due metodi resoconto() e resocontoHtml(), ma vengono utilizzati in sequenza uguale. Questo significa che possiamo utilizzare una classe che contiene un metodo principale che chiama in sequenza tre metodi (intestazione(), dettaglioNoleggio() e chiusura()) definiti astratti e implementati in due sottoclassi diverse, ResocontoTestuale e ResocontoHtml (figura 6.1), mentre la classe astratta che specifica il metodo principale (valore()) è Resoconto.

Vediamo prima la classe astratta, che non è altro che la trasformazione del metodo resoconto() usato in precedenza:

```
abstract class Resoconto
{
```

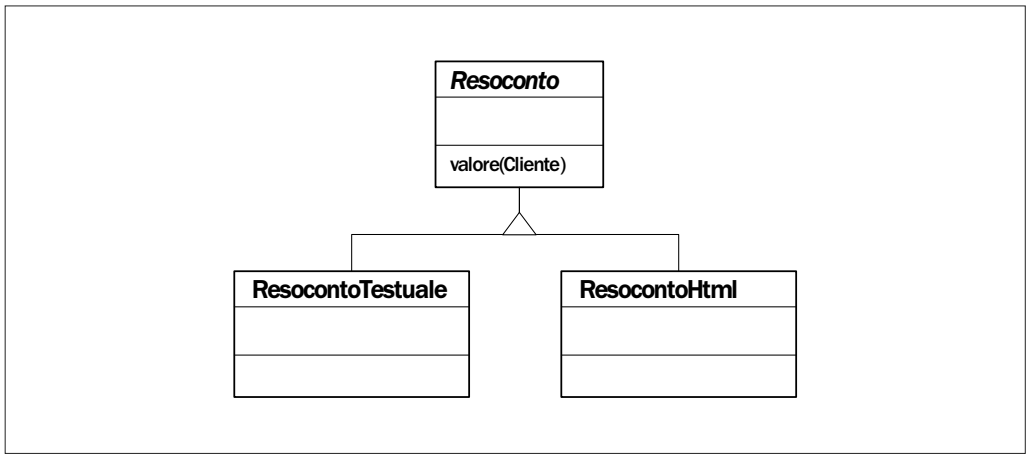

Figura 6.1 – Ereditarietà per sfruttare un modello comune.

```
    public String valore(Cliente cliente){
      String testo = intestazione(cliente);
      Iterator iterNoleggi = cliente.getNoleggi().iterator();
      while(iterNoleggi.hasNext()){
        Noleggio ogniNoleggio = (Noleggio)iterNoleggi.next();
        testo += dettaglioNoleggio(ogniNoleggio);
      }
      testo += chiusura(cliente);
      return testo;
    }

    abstract String intestazione(Cliente cliente);

    abstract String dettaglioNoleggio(Noleggio noleggio);

    abstract String chiusura(Cliente cliente);
  }
```

Il metodo valore() sarà quello chiamato dall'oggetto Cliente, dopo aver creato l'oggetto Resoconto concreto (ResocontoTestuale o ResocontoHtml).

I tre metodi astratti vengono invece implementati nelle due sottoclassi.

```
  public class ResocontoTestuale extends Resoconto
  {
    String intestazione(Cliente cliente){
```

```
        return "Resoconto di noleggio per " + cliente.getNome() + "\n";
    }

    String dettaglioNoleggio(Noleggio noleggio){
        return "\t" + noleggio.getSci().getModello() + "\t" + noleggio.calcolaImporto() + "\n";
    }

    String chiusura(Cliente cliente){
        return "Ammontare dovuto:\t Fr. " + cliente.calcolaImportoTotale() + "\n"+
                "Ha guadagnato " + cliente.calcolaPuntiTotale() + " punti di bonus\n";
    }
}


public class ResocontoHtml extends Resoconto
{
    String intestazione(Cliente cliente){
        return "<H1>Resoconto di noleggio per "+cliente.getNome()+"</H1>\n";
    }

    String dettaglioNoleggio(Noleggio noleggio){
        return "\t" + noleggio.getSci().getModello() + "\t" + noleggio.calcolaImporto() + "<BR>\n";
    }

    String chiusura(Cliente cliente){
        return "<P>Ammontare dovuto:\t Fr. " + cliente.calcolaImportoTotale() + "</P>\n" +
                "<P>Ha guadagnato <EM>" + cliente.calcolaPuntiTotale() + "</EM> punti di bonus</P>\n";
    }
}
```

Come si può notare questo meccanismo permette la condivisione di elementi che prima erano duplicati nei due metodi resoconto() e resocontoHtml().

Per ottenere questo abbiamo bisogno di una nuova gerarchia di classi che agisca da supporto a Cliente, in realtà ancora il "titolare" logico del resoconto di noleggio.

Ecco, infine, il main() di prova visto all'inizio adattato ai cambiamenti effettuati.

```
public class Main
{
    public static void main(String[] args){
        Noleggio noleggio1 = new Noleggio(new Sci("Stöckli 2B", 170, Sci.NORMALE),4);
        Noleggio noleggio2 = new Noleggio(new Sci("Völkl BX", 110, Sci.BAMBINI),5);
        Cliente cliente = new Cliente("Pinco Pallino");
        cliente.addNoleggio(noleggio1);
```

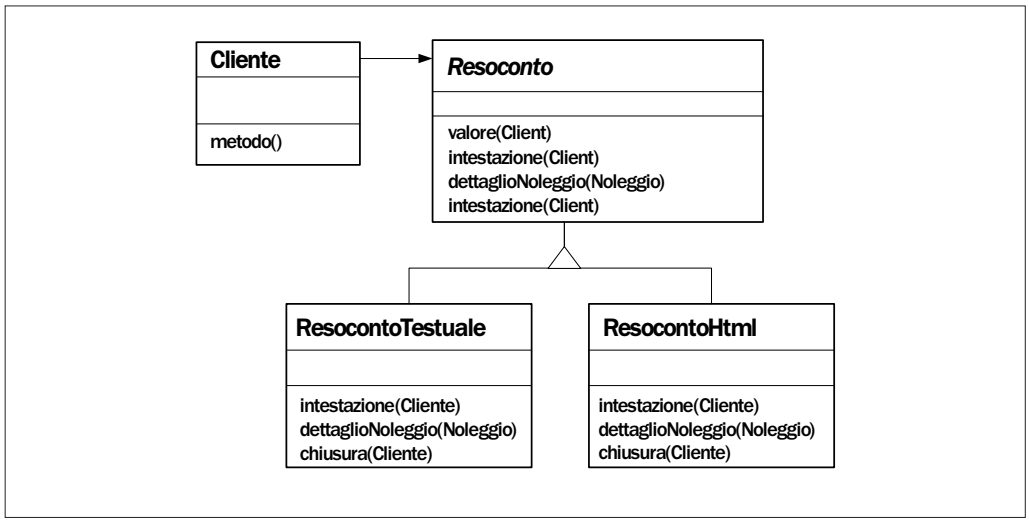

Figura 6.2 – Schema finale del pattern.

```
        cliente.addNoleggio(noleggio2);
        System.err.println(new ResocontoTestuale().valore(cliente));
        System.err.println(new ResocontoHtml().valore(cliente));
    }
}
```

Non siamo ancora alla soluzione ottimale perché con l'ausilio di nuove tecnologie di trasformazione, indipendenti dal linguaggio di programmazione utilizzato, potremmo semplicemente generare un output unico e neutro (ad esempio in XML) e poi trasformarlo con tecniche di trasformazione specifiche (ad esempio XSLT per XML). Torneremo sull'argomento in uno dei prossimi capitoli.

Il pattern Template Method ha comunque mostrato un utilizzo diverso dell'ereditarietà, cioè quello di costruirsi un modello (*template*, appunto) che determini lo scheletro comune di un algoritmo. Questo diventa poi adattabile attraverso chiamate a metodi astratti (anche detti *hook*), implementati in sottoclassi, che ne determinano il comportamento finale.

In questo capitolo…

In questo capitolo abbiamo visto un secondo modo di sfruttare la gerarchia, dopo quello visto in precedenza con il pattern Strategy.

In questa prima parte del libro, partendo da un design iniziale che rappresentasse una prima soluzione al problema, abbiamo eseguito alcuni passaggi di refactoring allo scopo di suddi-

videre meglio il codice nelle tre classi di partenza, dando a ognuna la sua giusta responsabilità. In seguito, valutando quali avrebbero potuto essere nuove richieste, abbiamo ulteriormente adattato il codice, utilizzando un paio di design pattern. Il risultato ottenuto è stato quello di un programma maggiormente object-oriented e pronto a integrare in modo omogeneo i cambiamenti previsti.