# Capitolo 5
# Problemi di classificazione

Proviamo ora a introdurre nuove richieste nel nostro programma. La più semplice e ovvia da immaginare è senza dubbio l'esigenza di modificare la classificazione degli sci, sia aggiungendo nuovi elementi, sia togliendone di esistenti.

Dobbiamo dapprima cercare all'interno del codice dove la classificazione viene specificata e dove viene utilizzata.

## Ulteriore delega di responsabilità

Ci accorgiamo così che la definizione avviene all'interno della classe Sci, con l'utilizzo di una variabile di istanza e una serie di costanti, mentre l'utilizzo vero e proprio avviene in calcolaPunti() e calcolaImporto(), metodi della classe Noleggio.

```
public class Noleggio
{
  ...
  double calcolaImporto(){
    ...
```

```
  }

  double calcolaPunti(){
    ...
  }
}
```

In precedenza, estraendo i metodi da Cliente, ci eravamo chiesti dove fosse più logico implementarli: in Noleggio o in Sci? Non cambiava molto, visto che le informazioni richieste erano due: la durata del noleggio (gestita in Noleggio) e il tipo di sci (gestito in Sci); ma abbiamo scelto di spostarli in Noleggio, perché questa era la classe più vicina a Cliente e quindi lo spostamento comportava meno modifiche.

Ora si aggiunge però un criterio che fa pendere la bilancia delle preferenze a favore di Sci, perché spostando i due metodi in quella classe sappiamo che isoliamo all'interno di Sci sia la definizione che l'utilizzo degli elementi di classificazione. Ogni modifica comporterà interventi unicamente in Sci.

Vediamo allora come procedere con le modifiche, iniziando dal metodo calcolaImporto(). Una prima cosa evidente consiste nel fatto che la durata del noleggio, non essendo presente in Sci, dev'essere passata come parametro.

```
public class Sci
{
  ...
  public double calcolaImporto(int giorni){
    ...
  }
}
```

Il nuovo metodo nella classe Sci diventerà adesso responsabile del calcolo, e verrà chiamato o da calcolaImporto() in Noleggio, o direttamente da resoconto() in Cliente. Meglio la prima soluzione perché ci evita di effettuare modifiche in Cliente e distribuisce in modo più logico le responsabilità.

Ecco quindi come si presenta in Noleggio la nuova versione semplificata che non fa altro che delegare la chiamata a Sci:

```
public class Noleggio
{
  ...
  double calcolaImporto(){
    return getSci().calcolaImporto(getGiorni());
  }
}
```

Ed ecco il metodo calcolaImporto() nella sua nuova veste all'interno di Sci, con i giorni di noleggio ricevuti come parametro:

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

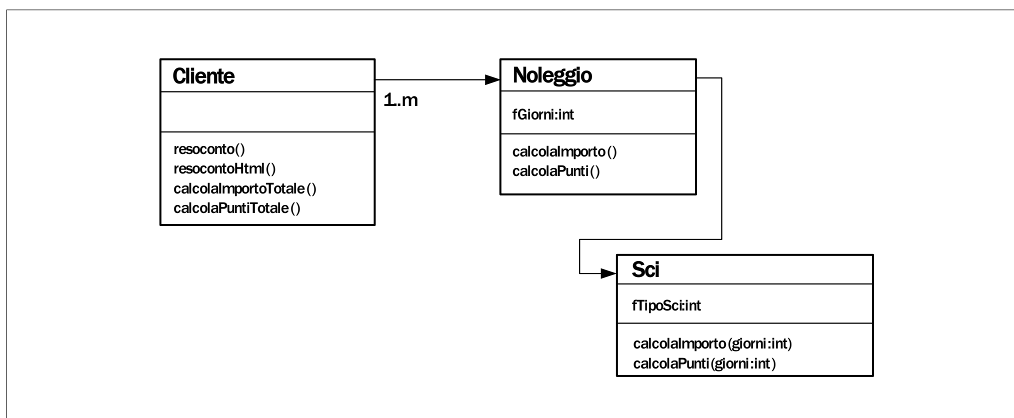


*Figura 5.1 – Spostamento di metodi.*

```
        return risultato;
    }
}
```

Stessa cosa succede per calcolaPunti(), con la chiamata in Noleggio e la nuova implementazione in Sci:

```
public class Noleggio
{
    ...
    public double calcolaPunti(){
        return getSci ().calcolaPunti(getGiorni());
    }
}


public class Sci
{
    ...
    public double calcolaPunti(int giorni){
        if ((getTipo() == Sci.CARVING) && (giorni > 1))
            return 2;
        else
            return 1;
    }
}
```

Con questi nuovi passaggi abbiamo di nuovo spostato e creato metodi, senza modifiche nel design e senza modifiche di funzionalità.

# Eliminare lo *switch*

La cosa più fastidiosa della soluzione usata per gestire la classificazione consiste nella duplicazione dei controlli. Abbiamo nel nostro caso due soli metodi che utilizzano le informazioni di classificazione (calcolaPunti() e calcolaImporto()), ma entrambi andrebbero modificati nel caso in cui venissero inserite nuove regole di classificazione. Se avessimo ulteriori metodi con switch o if che utilizzassero il tipo di Sci come criterio di scelta, tutti andrebbero modificati.

La programmazione a oggetti, con l'ereditarietà e il polimorfismo, ci offre meccanismi molto più eleganti per gestire situazioni di questo tipo. Queste due caratteristiche ci permettono di isolare i vari calcoli di prezzi e punti, semplificando il codice dei metodi, delegando al sistema la scelta della giusta chiamata. Si tratta infatti di definire una gerachia di classi, in cui ogni sottoclasse rappresenti un singolo tipo, con i suoi due metodi calcolaImporto() e calcolaPunti().

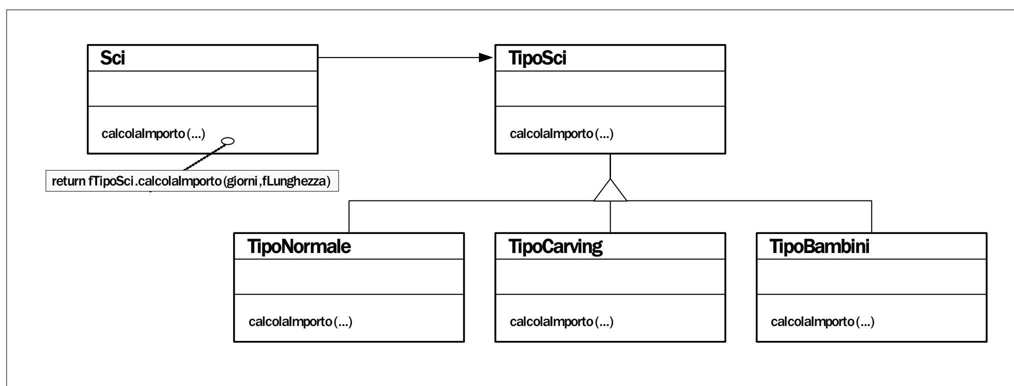


*Figura 5.2 – Nuova situazione con gerarchia.*

La gerarchia di classi rimpiazza l'utilizzo della variabile di istanza di tipo int in Sci e gestisce la logica dei calcoli di punti e importo. La classe Sci mantiene il riferimento a un oggetto della gerarchia TipoSci. In runtime l'oggetto realmente usato sarà un'istanza di una delle sottoclassi di TipoSci. In un utilizzo del genere della programmazione a oggetti si parla di modello chiamato Strategy pattern (in un caso del genere si potrebbe anche parlare di State pattern, visto che il punto di vista iniziale è l'oggetto Sci, ma la discussione sulle differenze tra Strategy e State non ha senso in questo capitolo introduttivo).

```
public class Sci
{
  ...
  private TipoSci fTipoSci;

  double calcolaImporto(int giorni){
    return fTipoSci.calcolaImporto(giorni, fLunghezza);
  }

  double calcolaPunti(int giorni){
    return fTipoSci.calcolaPunti(giorni);
  }
}
```

Ogni oggetto Sci avrà associato il suo oggetto TipoSci (TipoBambini, TipoNormale o TipoCarving), responsabile dei singoli calcoli calcolaImporto() e calcolaPunti(). È possibile che da qualche parte nel programma si debba ancora ricorrere a uno switch, al momento di creare il giusto oggetto TipoSci da associare a Sci. Ci sono modi per evitarlo, ma anche non potendo farlo, la cosa importante è garantire che questo si trovi in un punto unico e isolato del programma, responsabile

della creazione degli oggetti. Tutte le altre funzionalità dipendenti dalla classificazione devono invece trovarsi nelle sottoclassi di TipoSci.

Vediamo come viene suddiviso il codice nel caso di calcolaImporto():

```
class TipoBambini extends TipoSci
{
  double calcolaImporto(int giorni, int lunghezza){
    double base = 15 * (double)lunghezza/100;
    double risultato = base;
    if (giorni > 3)
      risultato += (giorni - 3) * base;
    return risultato;
  }
}

class TipoNormale extends TipoSci
{
  double calcolaImporto(int giorni, int lunghezza){
    double risultato = 25;
    if (giorni > 2)
      risultato += (giorni - 2) * 15;
    return risultato;
  }
}

class TipoCarving extends TipoSci
{
  ...
  double calcolaImporto(int giorni, int lunghezza){
    return giorni * 25;
  }
}
```

Come si vede, distribuendo il codice dei vari calcoli nelle relative sottoclassi, anche la leggibilità delle modalità di calcolo dei singoli prezzi è migliorata.

Un procedimento analogo l'abbiamo con calcolaPunti(), con l'aggiunta che in questo caso usiamo l'ereditarietà sia per il polimorfismo, sia per condividere una parte di implementazione, cioè quella relativa al calcolo dei punti per i tipi Bambini e Normale, che, essendo uguali, possono essere riassunte nella classe base TipoSci:

```
abstract class TipoSci
{
  abstract double calcolaImporto(int giorni, int lunghezza);
```

```
  double calcolaPunti(int giorni){
    return 1;
  }
}

class TipoCarving extends TipoSci
{
  ...
  double calcolaPunti(int giorni){
    if (giorni > 1)
      return 2;
    else
      return 1;
  }
}
```

Anche questi ultimi passaggi utilizzati per eliminare duplicazioni di elementi condizionali (switch e if) non hanno aggiunto niente a livello di funzionalità. Hanno però migliorato il codice preparandolo a eventuali modifiche nelle modalità di classificazione degli sci. Aggiunte di nuo-

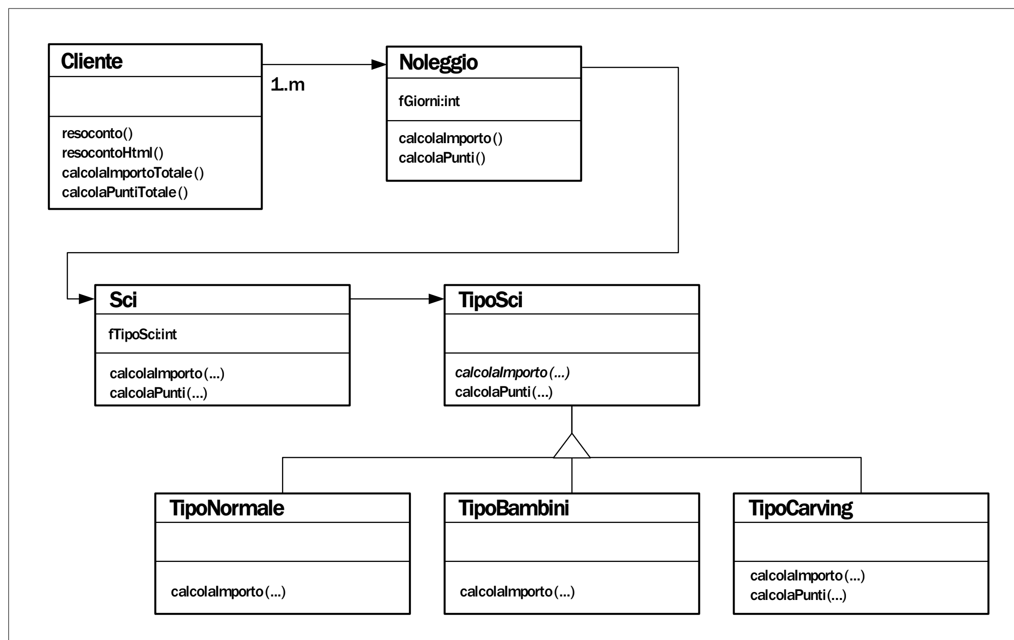


*Figura 5.3 – Schema completo delle classi.*

vi tipi di sci hanno come conseguenza l'aggiunta di nuove sottoclassi e di nuovi metodi di calcolo, ma non la modifica di quelli esistenti.

Inoltre anche singole modifiche nel calcolo dei prezzi o dei punti sono gestite in modo del tutto locale, visto che ogni singolo calcolo si trova in una sua classe specifica.

Come inizializzare ora la variabile fTipoSci all'interno di Sci?

Il modo più semplice in questo caso è sfruttare l'informazione di tipo int per creare un oggetto della classe specifica. È vero che introduciamo uno switch, ma siamo in grado di assicurare che questo sia l'unico, perché è quello che centralizza la creazione degli oggetti.

```
public class Sci
{
  ...
  private TipoSci fTipoSci;

  public Sci(String modello, int lunghezza, int tipoSci){
    fModello = modello;
    fLunghezza = lunghezza;
    setTipo(tipoSci);
  }

  public void setTipo(int tf){
     switch(tf){
        case Sci.NORMALE:
          fTipoSci = new TipoNormale();
          break;
       case Sci.CARVING:
          fTipoSci = new TipoCarving();
          break;
       case Sci.BAMBINI:
          fTipoSci = new TipoBambini();
          break;
    }
  }
}
```

Esistono altre maniere più raffinate, tra cui rientra l'utilizzo di un pattern chiamato Factory Method, che vedremo più avanti.

# In questo capitolo…

Con le modifiche proposte in questo capitolo abbiamo affrontato uno dei problemi principali della programmazione: la duplicazione della logica. Ogni scelta condizionale che si ripete

più di una volta va eliminata perché rappresenta una potenziale fonte di errore in caso di modifiche. Il meccanismo di ereditarietà, nel suo sfruttamento del polimorfismo, è in questo caso un ottimo mezzo che permette di delegare lo switch al sistema, piuttosto che al codice del programma.

Nel prossimo capitolo vedremo come usare l'ereditarietà in un modo diverso, con l'utilizzo di un nuovo pattern.