Parte I
Visione generale: verso un design object-oriented

"The task of the software development team is to create an illusion of simplicity. We build abstractions to help us create this illusion; these abstractions are an essential way we mitigate the intellectual complexity that lurks within our systems…"

(Il compito della squadra di sviluppo del software consiste nel creare un'illusione di semplicità. Costruiamo astrazioni che ci aiutino a creare questa illusione; tali astrazioni sono uno strumento essenziale per mitigare la complessità intellettuale che sta in agguato nei nostri sistemi…)

GRADY BOOCH

Capitolo 2
Problema iniziale

Parafrasando un famoso detto a proposito delle immagini, possiamo dire che in informatica non c'è niente di più esplicativo di un buon esempio, naturalmente comprendente codice.

Siccome l'intero testo, pur basandosi su elementi teorici, vuole avere un taglio pratico, tutti i capitoli di questa prima parte si baseranno su un esempio che vuole essere un modello di partenza per tutto ciò che verrà spiegato più avanti. Una sorta di aperitivo per stuzzicare l'appetito che verrà appagato (lo speriamo) con gli approfondimenti dei singoli temi trattati nelle parti a seguire.

Il titolo dell'intera parte è "Verso un design object-oriented" perché, indipendentemente dal fatto che si parli di design pattern, refactoring o test, il risultato di tutte le modifiche che verranno apportate al codice e al design porteranno a una codifica che sfrutti meglio i principi che stanno alla base della programmazione a oggetti. Imparare a usare i pattern significa infatti anche apprendere a sfruttare al meglio le potenzialità della programmazione a oggetti. Avremo ancora modo di parlare di questo aspetto quando approfondiremo il tema dei pattern.

Un esempio, soprattutto se presentato all'inizio del libro, deve essere semplice. Il rischio, nel nostro caso, è quello di voler spiegare l'utilità di elementi (design pattern, refactoring e test) con un problema troppo semplice per poterne giustificare l'utilizzo. È un rischio che mi prendo, sapendo quanto sia importante e utile, a fini didattici, catturare l'attenzione attraverso casi pratici.

L'idea è quella di impostare un programma con un design semplice che ne garantisca una prima funzionalità. In seguito, attraverso refactoring e inserimento di pattern si cercherà di far crescere il codice, rendendolo più flessibile e più pronto ai cambiamenti: in altre parole più object-oriented. L'approccio è quello dello sviluppo agile, che presuppone l'implementazione

veloce delle funzionalità di base, iterando in seguito su design e implementazione per aggiungere man mano nuove funzionalità, o semplicemente per migliorare il codice e renderlo più pronto a cambiamenti futuri.

Descrizione

Il problema presentato prende lo spunto da un esempio descritto da Fowler nel suo libro sui refactoring [Fowler-1999], con scopi e procedimenti diversi. Qui si andrà un po' più lontano, discutendo più soluzioni e introducendo i concetti di design pattern e test di unità.

Immaginiamo di trovarci in un villaggio alpino a vocazione turistica, durante il periodo invernale. La regione vive di turismo e offre una ricca serie di attività, di cui lo sci, con gli impianti di risalita, non rappresenta che una possibilità fra le tante.

Non ci si aspetta che ogni turista arrivi già perfettamente equipaggiato per ogni genere di attività, ma il villaggio, con il suo ufficio del turismo, i suoi negozi e suoi alberghi, si impegna a mettere a disposizione degli ospiti (in vendita, a noleggio o in prestito) ogni sorta di materiale per qualsiasi attività.

Consideriamo ora di trovarci in un negozio di sport di questo centro alpino. La vendita di materiale sportivo, come gli sci, è sicuramente un'attività importante, come in ogni negozio del genere. In più, però, qui gli sci si possono anche noleggiare, pagandoli sulla base dei giorni di noleggio, del tipo di sci, ecc.

Quello che vogliamo realizzare noi è un programma in grado di calcolare e stampare l'ammontare dovuto da un cliente che, arrivato al villaggio, noleggia sci in questo negozio per alcuni giorni durante il periodo delle sue vacanze.

Il programma gestisce per ogni cliente quali e quante paia di sci sono stati noleggiati e per quanti giorni (immaginiamo ad esempio un padre di famiglia che decida di noleggiare sci per tutti durante un'intera settimana). Attraverso queste informazioni e il tipo di sci usati, il programma è in grado di stampare la fattura.

Ci preoccupiamo qui unicamente della logica del programma e non della gestione su banca dati delle informazioni registrate nei singoli oggetti.

Primo design

Vediamo in figura 2.1 un primo schema di classi, derivato direttamente dai requisiti appena esposti, e cerchiamo subito di analizzarne i punti deboli.

La classe Sci, oltre a informazioni specifiche che tralasciamo e che raggruppiamo sotto la variabile di istanza fModello, ha la responsabilità della gestione del tipo e della lunghezza degli sci. Dal tipo di sci, ed eventualmente dalla lunghezza, dipenderà direttamente il costo giornaliero del noleggio.

```
public class Sci
{
```

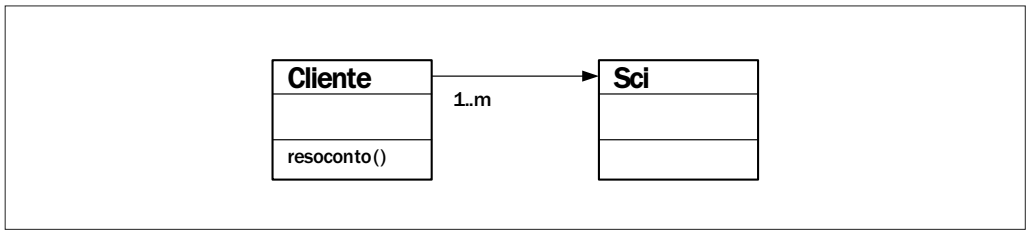

Figura 2.1 – Primo design.

```
    public static final int NORMALE = 0;
    public static final int CARVING = 1;
    public static final int BAMBINI = 2;

    private String fModello;
    private int fLunghezza;
    private int fTipoSci;

    public Sci(String modello, int lunghezza, int tf){
      fModello = modello;
      fLunghezza = lunghezza;
      fTipoSci = tf;
    }

    public int getTipo(){
      return fTipoSci;
    }

    public void setTipo(int tipoSci){
      fTipoSci= tipoSci;
    }

    public String getModello(){
      return fModello;
    }

    public int getLunghezza(){
      return fLunghezza;
    }
}
```

I possibili tipi di sci, cioè i possibili valori per la variabile di istanza fTipoSci sono rappresentati dalle tre costanti di classe. Avremo quindi sci di tipo *carving*, sci per bambini e sci considerati "normali", cioè non facenti parte dei primi due insiemi. Un paio di sci non può avere contemporaneamente più di un tipo, anche perché il tipo serve a determinare il prezzo, che dev'essere unico.

Si tratta ora di decidere dove gestire l'informazione sui giorni di noleggio. A prima vista potrebbe sembrare un'informazione da inserire in Sci. L'oggetto Cliente che si riferisce a Sci ha così la possibilità di delegare a questo sia l'informazione sul prezzo sia i giorni di noleggio.

Questo approccio ha però due svantaggi. Da un punto di vista logico il numero di giorni di noleggio non ha molto a che vedere con il tipo di sci scelti; andrebbe quindi gestito separatamente. Inoltre se il negozio avesse più paia di sci uguali, cosa molto probabile, dovremmo prevedere un modo per associare le singole copie con gli oggetti in runtime, cosa che renderebbe il programma più complesso del dovuto.

Nemmeno la gestione nel cliente sembra la più appropriata, visto che gli verrebbe chiesto di gestire due tipi di informazione diversi e saparati, ma correlati tra di loro. Richiederebbe uno sforzo di sincronizzazione evitabile.

Si tratta di un problema molto simile a quello dell'iteratore, risolto proprio con l'utilizzo di un pattern (*Iterator*) che affronteremo nel capitolo specifico sui design pattern.

Nel nostro caso particolare la soluzione consiste nell'introdurre una classe intermedia (Noleggio) che permetta di migliorare la situazione precedente, evitando due cose: da un lato di gestire il numero di giorni di noleggio all'interno della classe Sci e, dall'altro, una doppia gestione di dati all'interno di Cliente.

Visto che Sci non contiene informazioni relative ai giorni di noleggio, è indipendente da questo tipo di relazione. Quindi, teoricamente, tutte le paia di sci uguali possono essere rappresentate dallo stesso oggetto. Questa relazione non è visibile a livello di codice di classe, poiché il riferimento è unidirezionale e va da Noleggio a Sci.

Ecco come appare la classe Noleggio.

```
public class Noleggio
{
  private Sci fSci;
  private int fGiorni;

  public Noleggio(Sci sci, int g){
    fSci= sci;
    fGiorni= g;
  }

  public int getGiorni(){
    return fGiorni;
  }
```

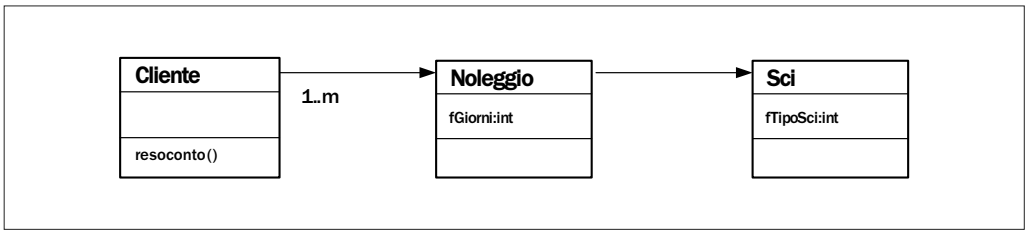

Figura 2.2 – Inserimento di Noleggio nel design.

```
  public Sci getSci(){
    return fSci;
  }
}
```

A questo punto siamo in grado di specificare la classe Cliente, che avrà una relazione diretta con elementi di tipo Noleggio. Questi elementi Noleggio fanno da tramite tra Cliente e Sci.

Anche nel caso di Cliente, come già con Sci, ci limitiamo a gestire le informazioni minime indispensabili. Qui utilizziamo una variabile fNome di tipo String come identificatore del cliente, oltre, naturalmente, alla lista di elementi Noleggio.

```
public class Cliente
{
  private String fNome;
  private List fNoleggi;

  public Cliente(String nome){
    fNome = nome;
    fNoleggi = new ArrayList();
  }

  public String getNome(){
    return fNome;
  }

  public void addNoleggio(Noleggio arg){
    fNoleggi.add(arg);
  }

  protected List getNoleggi(){
    return fNoleggi;
```

```
    }

    public String resoconto(){
      String testo = "Resoconto di noleggio per " + getNome() + "\n";
      ...
      return testo ;
    }
}
```

Il metodo resoconto() viene mostrato qui solo parzialmente. Si tratta del metodo utilizzato per eseguire il calcolo dell'ammontare da pagare per ogni cliente. Il valore restituito è una stringa contenente il testo finale da stampare su una fattura.

Funzionalità

L'unica funzionalità richiesta al programma è quella di stampare il resoconto delle fatture. Dalla struttura attuale delle classi vediamo che il metodo resoconto() è stato definito in Cliente.

Ecco una prima versione di 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()){
    double costo = 0;
    Noleggio ogniNoleggio = (Noleggio)iterNoleggi.next();
    switch(ogniNoleggio.getSci().getTipo()){  //somma ammontare per ogni paio
      case Sci.NORMALE: //25+15 ogni giorno dopo il 2. 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;
```

```
        }
        puntiAccumulati++;
        if ((ogniNoleggio.getSci().getTipo() == Sci.CARVING) &&
          (ogniNoleggio.getGiorni() > 1))    //extra bonus carving di >= 2 giorni
          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;
    }
```

Si tratta di creare un iteratore sui vari noleggi di un cliente. Per ogni noleggio si deve calcolare il costo, sulla base dei giorni e del tipo di sci, e i punti di bonus accumulati, anch'essi calcolati con gli stessi criteri.

Un paio di sci "carving" costerebbe nel nostro esempio 25 CHF al giorno, un paio "normale" 25 CHF di base più 15 a partire dal terzo giorno, un paio di sci della categoria "bambini", una base di 15 CHF, moltiplicata per la lunghezza in metri più la stessa somma ogni giorno a partire dal quarto giorno. I punti sono invece incrementati di 1 per sci "normali" e "bambini", di 2 per gli sci di tipo "carving", tenuti per almeno due giorni.

Test

Nel capitolo specifico sul test di unità avremo modo di sottolineare il fatto che per ogni nuova classe va implementata una classe corrispondente di test che verifichi il funzionamento dei suoi metodi. Introdurremo anche l'utilizzo di un framework, JUnit, che semplifica il compito di chi scrive il codice di test.

Siccome a questo punto della spiagazione JUnit non è ancora conosciuto, ci limitiamo a vedere quali sono gli elementi che andrebbero verificati. Un primo principio importante consiste nel fatto che il test non solo deve eseguire del codice, ma dev'essere anche in grado di dimostrare che questo codice funzioni come ci si aspetta, senza stampare lunghe liste di output da controllare a mano: il controllo spetta al codice di test.

Nel nostro caso definiamo un certo numero di metodi di test per ogni classe da verificare. Ogni metodo di test sarà di tipo `boolean` e restituirà `true` nel caso in cui il test arrivi a buon fine, `false` nel caso in cui ci fossero problemi.

Vediamo dapprima come verificare il funzionamento minimo della classe `Sci`. In questo caso si tratta semplicemente di dimostrare che il metodo `setTipoSci()` modifichi in modo corretto l'informazione interna sul tipo di sci.

```
public class SciTest …
{
```

```
  ...
  public boolean testSetTipo(){
    Sci sci = new Sci("Rossignol AX", Sci.CARVING);
    if (sci.getTipo() != Sci.CARVING)
      return false;
    sci.setTipo(Sci.NORMALE);
    if (sci.getTipo() != Sci.NORMALE)
      return false;
    return true;
  }
}
```

Il controllo è banale, ma garantisce la verifica di consistenza tra la funzionalità di setTipo() e quella di getTipo(). Se una delle due verrà modificata in modo sbagliato in futuro, il test se ne accorgerà (*regression test*). La maggior parte dei test è semplice e funziona in questo modo, verificando una funzionalità singola. La somma di tutti i test serve a verificare la funzionalità completa del programma e soprattutto a garantirne la consistenza.

Vediamo in un secondo esempio come verificare il funzionamento dell'aggiunta di nuovi oggetti Noleggio alla lista gestita da Cliente.

```
public class ClienteTest…
{
  ...
  public boolean testAddNoleggio(){
    Cliente cliente = new Cliente("Mario Rossi");
    cliente.addNoleggio(new Noleggio(new Sci("Stöckli 2B", 170, Sci.NORMALE),3));
    cliente.addNoleggio(new Noleggio(new Sci("Völkl BX", 110, Sci.BAMBINI),5));
    List noleggi = cliente.getNoleggi();
    Noleggio singolo = (Noleggio)noleggi.get(0);
    if (!singolo.getSci().getModello().equals("Stöckli 2B"))
      return false;
    singolo = (Noleggio)noleggi.get(1);
    if (!singolo.getSci().getModello().equals("Völkl BX "))
      return false;
    return true;
  }
}
```

Programma principale di prova

Per vedere il funzionamento del codice implementato, basta scrivere una classe principale con alcuni esempi:

```
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);
      cliente.addNoleggio(noleggio2);
      System.err.println(cliente.resoconto());
   }
}
```

Ciò produce il seguente output:

```
Resoconto di noleggio per Pinco Pallino
   Stöckli 2B  55.0
   Völkl BX 49.5
Ammontare dovuto: Fr. 104.5
Ha guadagnato 2 punti di bonus
```

In questo capitolo…

Abbiamo presentato il problema che svilupperemo nei prossimi capitoli e abbiamo proposto un design iniziale che consenta di implementare la funzionalità richiesta. Abbiamo poi introdotto alcuni semplici princìpi di test, che approfondiremo più avanti, nella parte specifica dedicata al test di unità. Adesso siamo pronti per una prima verifica del codice.