Capitolo 24
Design by refactoring

Con questo capitolo mostriamo di nuovo l'introduzione di un pattern nel design di un'applicazione come risultato di un processo di refactoring. Qui, però, non si tratta di un refactoring "preconfezionato", come quelli degli esempi precedenti, ma si tratta di modifiche "ad hoc", scaturite da necessità contingenti.

Refactoring non significa quindi solo applicare regole estratte da un elenco, ma anche applicare modifiche al codice seguendo necessità specifiche del problema che sfociano in mutamenti del design e, nel nostro caso, nell'applicazione di un pattern conosciuto.

Problema

Eseguiamo le nostre modifiche attraverso un problema già visto: la gestione uniforme delle monete. Più precisamente, vogliamo analizzare di nuovo la soluzione adottata per il doppio dispatching necessario a rendere trasparente l'utilizzo del metodo `add()` nelle classi `Money` e `MoneyBag`. La soluzione al problema del doppio dispatching vista nella parte del libro relativa al test è più che accettabile per un problema come quello proposto. Nonostante ciò, vale la pena valutare altre soluzioni, magari più eleganti e generiche, utili in contesti maggiormente complessi.

L'idea del doppio dispatching viene associata al pattern Visitor, proposto da Gamma e Beck [JUnit] per la soluzione del problema precedente. In realtà la soluzione al doppio dispatching è solo una conseguenza indiretta del pattern Visitor. Il suo scopo dichiarato è infatti un altro. Prima di introdurre il pattern Visitor, diamo però un'occhiata alla soluzione proposta in precedenza.

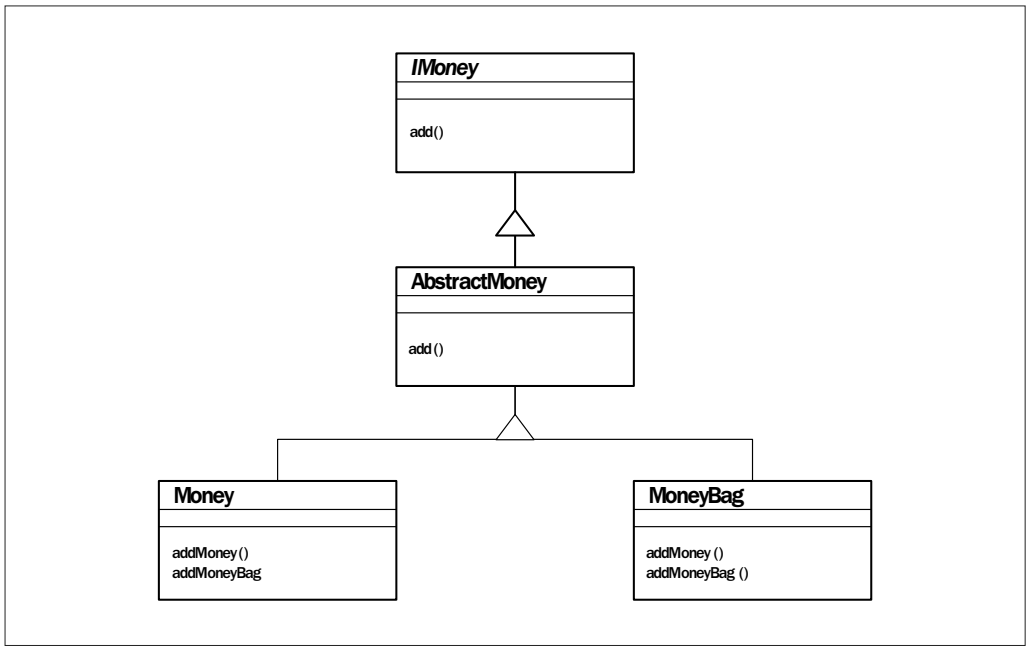

Figura 24.1 – Gerarchia di IMoney.

Soluzione precedente

Ricordo che si trattava di unificare in un'unica chiamata add(), visibile nell'interfaccia IMoney, i seguenti quattro casi possibili:

Money + Money
Money + MoneyBag
MoneyBag + Money
MoneyBag + MoneyBag

Considerando che sia Money che MoneyBag fanno parte della gerarchia IMoney, si vuole mostrare in interfaccia questa unica chiamata:

```
public IMoney add(IMoney m);
```

Come già visto nel capitolo riguardante il test, il problema non è il dispatching sull'elemento chiamante, che avviene in modo automatico, ma sul parametro, che nei linguaggi tradizionali

non viene considerato elemento di dispatching. Abbiamo così dovuto trattare in modo manuale questo caso, implementando in add() la scelta se considerare un oggetto di tipo Money oppure uno di tipo MoneyBag l'elemento passato come parametro.

```
public IMoney add(IMoney m){
  if (m instanceof Money){
    return addMoney((Money)m);
  }else{un
    return addMoneyBag((MoneyBag)m);
  }
}
```

Il metodo è unico, quindi in comune tra Money e MoneyBag. Siccome IMoney è un'*interface*, non potevamo implementare il metodo in IMoney. Abbiamo quindi optato per la soluzione di inserire una classe astratta tra IMoney e le due classi concrete, come mostrato nello schema di figura 24.1. Vedremo che la classe astratta intermedia risulterà utile anche per la soluzione con il pattern Visitor.

Il pattern Visitor

Questo pattern permette di modellare un'operazione da eseguire sugli elementi di un'intera struttura di oggetti. Visitor dà sufficiente indipendenza all'operazione da permettere la sua ridefinizione, senza dover modificare le classi degli elementi sui quali l'operazione viene applicata. In altre parole: le operazioni da eseguire sulla struttura di dati sono slegate dalla struttura stessa. Modifiche nelle operazioni o modifiche nella struttura rimangono indipendenti. Questo permette di mantenere la struttura dati meno complessa.

Esempio

Iniziamo con un esempio, in cui mostriamo la soluzione senza pattern e la soluzione con il pattern. Supponiamo di voler modellare l'organigramma di una ditta. Ogni elemento dell'organigramma può essere un'unità organizzativa, e quindi contenere altri elementi, in modo ricorsivo, oppure è un elemento finale, cioè una persona. La struttura si lascia descrivere in modo elegante con il pattern Composite.

Abbiamo un elemento principale, descritto come classe astratta o *interface*:

```
public interface Element
{
  public List getSubElements();
}
```

Da questo elemento derivano sia l'elemento contenitore (Container), sia l'elemento Person.

```
public class Container implements Element
{
  private String fContainerName;
  private List fSubElements;

  public Container(String organizationName, List subElements){
    fContainerName = organizationName;
    fSubElements = subElements;
  }

  public List getSubElements(){
    return fSubElements;
  }

  public String getContainerName(){
    return fContainerName;
  }
}

public class Person implements Element
{
  private String fName;
  private String fRole;

  public Person(String name, String address){
    fName = name;
    fRole = address;
  }

  public List getSubElements(){
    return null;
  }

  public String getName(){
    return fName;
  }

  public String getRole(){
    return fRole;
  }
}
```

Supponendo di voler visualizzare in modo testuale e sequenziale tutti gli elementi dell'organigramma, la prima possibilità sarebbe quella di prevedere la stampa dei singoli elementi nelle classi stesse coinvolte, cioè in Container e Person.

```
public class Container implements Element
{
  …

  public void printAll(){
    System.out.println("Container: " + getContainerName());
    Iterator iter = getSubElements().iterator();
    while(iter.hasNext()){
      Element element = (Element)iter.next();
      element.printAll();
    }
  }
}


public class Person implements Element
{
  …

  public void printAll(){
    System.out.println("Person: " + getName() + " " + getRole());
  }
}
```

Per rendere trasparente e pulita la chiamata al metodo di stampa, applicando il pattern Composite, è necessario aggiungere il metodo a Element.

```
public interface Element
{
  public List getSubElements();
  public void printAll();
}
```

Ecco un esempio di utilizzo di queste classi, con l'aggiunta dell'output che ne deriva:

```
public class Main
{
  public static void main(String[] args){
    List subElements1 = new ArrayList();
```

```
        List subElements2 = new ArrayList();
        subElements2.add(new Person("A. Rossi", "Capufficio"));
        subElements2.add(new Person("B. Bianchi", "Segretario"));
        subElements2.add(new Person("C. Verdi", "Contabile"));

        subElements1.add(new Person("D. Neri", "Direttore"));
        subElements1.add(new Container("Ufficio contabilità", subElements2));
        Container container = new Container("Agenzia Immobiliare", subElements1);
        container.printAll();
    }
}
```

Output:

```
Container: Agenzia Immobiliare
Person: D. Neri Direttore
Container: Ufficio contabilità
Person: A. Rossi Capufficio
Person: B. Bianchi Segretario
Person: C. Verdi Contabile
```

Problema

Nella soluzione vista c'è un problema fondamentale che può diventare grave nel caso in cui arrivasse la richiesta di implementare vari tipi di visualizzazione della struttura: il codice di attraversamento e visualizzazione della struttura dati fa parte della struttura stessa.

Significa che se volessimo realizzare una nuova funzionalità di visualizzazione che ci stampi, ad esempio, unicamente i nomi delle persone, senza altre informazioni, dovremmo aggiungere un nuovo metodo alle singole classi e un nuovo metodo in interfaccia (Element).

Aggiunta di Visitor

Il pattern Visitor ci consente di isolare le operazioni in una gerarchia separata dalla struttura dati, permettendo l'aggiunta di nuove operazioni, senza intervenire sulla struttura stessa, che rimane quindi più stabile e chiara, senza metodi di attraversamento e visualizzazione che vanno aggiunti o tolti in continuazione.

Iniziamo a definire la gerarchia del pattern Visitor attraverso l'elemento interfaccia omonimo:

```
public interface Visitor
{
  public void visitContainer(Container container);
```

```
    public void visitPerson(Person person);
}
```

I metodi visit() delle sottoclassi dovranno implementare la funzionalità di visita, cioè di visualizzazione (ed eventualmente di attraversamento) della struttura dati. Dove vengono chiamati questi metodi? Qui sta il punto cruciale del pattern. L'oggetto Visitor viene passato alla struttura attraverso un metodo accept(), da implementare in ogni singola classe della struttura dati. In accept() verrà chiamato visit() dell'oggetto Visitor arrivato come parametro, eseguendo il dispatching sull'oggetto parametro.

Ecco le modifiche nelle classi della struttura dati. Element deve dichiarare accept(). Questo metodo rimarrà unico, indipendentemente dal numero di elementi concreti Visitor abilitati ad attraversare la struttura.

```
public interface Element
{
    public List getSubElements();
    public void accept(Visitor visitor);
}
```

Sia Container che Person dovranno poi aggiungere l'implementazione concreta del metodo.

```
public class Container implements Element
{
    …

    public void accept(Visitor visitor) {
        visitor.visitContainer(this);
    }
}


public class Person implements Element
{
    …

    public void accept(Visitor visitor) {
        visitor.visitPerson(this);
    }
}
```

Ed ecco ora un'implementazione di Visitor che realizza la funzionalità messa a disposizione precedentemente da printAll():

```
public class VisitorAll implements Visitor
{
  public void visitContainer(Container container){
    System.out.println("Container: " + container.getContainerName());
    Iterator iter = container.getSubElements().iterator();
    while(iter.hasNext()){
      Element element = (Element)iter.next();
      element.accept(this);
    }
  }

  public void visitPerson(Person person) {
    System.out.println("Person: " + person.getName() + " " + person.getRole());
  }
}
```

Ecco un esempio di utilizzo, con l'output che ne deriva, identico alla versione con printAll():

```
public class Main
{
  public static void main(String[] args) {
    List subElements1 = new ArrayList();
    List subElements2 = new ArrayList();
    subElements2.add(new Person("A. Rossi", "Capufficio"));
    subElements2.add(new Person("B. Bianchi", "Segretario"));
    subElements2.add(new Person("C. Verdi", "Contabile"));

    subElements1.add(new Person("D. Neri", "Direttore"));
    subElements1.add(new Container("Ufficio contabilità", subElements2));
    Container container = new Container("Agenzia Immobiliare", subElements1);
    container.accept(new VisitorAll());
  }
}
```

Output:

```
Container: Agenzia Immobiliare
Person: D. Neri Direttore
Container: Ufficio contabilità
Person: A. Rossi Capufficio
Person: B. Bianchi Segretario
Person: C. Verdi Contabile
```

E ora, come esempio, una seconda implementazione di Visitor, usata per visualizzare unicamente i nomi delle persone presenti nell'organigramma.

```
public class VisitorPersons implements Visitor
{
  public void visitContainer(Container container){
    Iterator iter = container.getSubElements().iterator();
    while(iter.hasNext()){
      Element element = (Element)iter.next();
      element.accept(this);
    }
  }

  public void visitPerson(Person person) {
    System.out.println("Person: " + person.getName());
  }
}
```

Questo è l'output ottenuto utilizzando questa seconda implementazione di Visitor:

```
Person: D. Neri
Person: A. Rossi
Person: B. Bianchi
Person: C. Verdi
```

Nel nostro esempio abbiamo usato la variante di Visitor in cui il compito di implementare l'attraversamento della struttura dati viene delegato al Visitor stesso. Si parte cioè dal presupposto che la struttura possa essere attraversata in vari modi e che quindi ogni oggetto Visitor possa decidere come permettere tale attraversamento. L'alternativa è quella di lasciare alla struttura stessa l'onere di decidere come implementare l'attraversamento, unico, delegando a Visitor solamente il compito della visualizzazione. Le modifiche in questo caso sarebbero sia in visitContainer(), sia in accept() di Container. Ecco la variante implementata per VisitorAll:

```
public class Container implements Element
{
  ...
  public void accept(Visitor visitor) {
    visitor.visitContainer(this);
    Iterator iter = getSubElements().iterator();
    while(iter.hasNext()){
      Element element = (Element)iter.next();
      element.accept(this);
```

```
    }
  }
}

public class VisitorAll implements Visitor
{
  …
  public void visitContainer(Container container) {
    System.out.println("Container: " + container.getContainerName());
  }
}
```

Utilizzo di overloading

Come qualcuno avrà notato, non è indispensabile che i metodi di Visitor si chiamino tutti in modo diverso (visitContainer(), visitPerson()). Si potrebbe fare uso del meccanismo di overloading e chiamare tutti i metodi allo stesso modo, ridefinendo Visitor come segue:

```
public interface Visitor
{
  public void visit(Container container);
  public void visit(Person person);
}
```

Il motivo per dare a tutti un nome diverso è dovuto alla chiarezza del codice. Il metodo visitContainer() verrà chiamato all'interno dell'implementazione di accept() di Container, mentre il metodo visitPerson() verrà chiamato da accept() di Person. I due metodi risultano più chiari e meglio documentati.
Chiamare entrambi i metodi visit() potrebbe inoltre creare un motivo di confusione per un aspetto particolare. Infatti sia in Container che in Person l'implementazione di accept() risulterebbe essere uguale.

```
public void accept(Visitor visitor) {
  visitor.visit(this);
}
```

Ma attenzione! Non è uguale. Se lo fosse, si sarebbe tentati di implementare il metodo in una classe comune, magari trasformando Element da interfaccia a classe astratta, per contenere accept(), ma non sarebbe la stessa cosa. Infatti this diventa l'elemento discriminante per la chiamata agli overloading visit(), quindi deve essere specifico, non può essere genericamente un Element, bensì Container o Person. In altre parole: anche se il metodo sembra identico (e infatti lo è, almeno a livello di testo), va implementato in entrambe le classi, affinché l'interpretazione di this sia diversa.

Doppio dispatching con Visitor

Torniamo al nostro problema iniziale sulle monete, cioè quello di riuscire a chiamare add() in modo generico, senza dover specificare se gli elementi sono Money o MoneyBag. Siccome Java non prevede il doppio dispatching, il pattern Visitor rappresenta un'alternativa elegante per gestire in modo semplice combinazioni di Money e MoneyBag (in cui entrambe possono apparire come oggetto che invoca e come primo parametro).

Siccome sull'oggetto chiamante il dispatching è automatico, l'idea di base, presente in Visitor, è quella di utilizzare all'interno di add() il parametro come elemento che invoca il metodo addMoney(), rispettivamente addMoneyBag(). Vediamo le due implementazioni, quella presente nella classe Money e quella presente nella classe MoneyBag:

```
class Money implements IMoney
{
  …

  public IMoney add(IMoney m) {
    return m.addMoney(this);
  }
}
```

```
    class MoneyBag implements IMoney
    {
      …

      public IMoney add(IMoney m) {
        return m.addMoneyBag(this);
      }
    }
```

Come si può notare, in entrambe le implementazioni, in cui, dalla chiamata di interfaccia, si entra con un dispatching semplice, il parametro viene utilizzato per effettuare a sua volta la chiamata. Nel primo caso, essendo il primo oggetto un Money, la chiamata sarà addMoney(). Nel secondo caso, essendo il primo oggetto un MoneyBag, la chiamata sarà addMoneyBag().

A seconda del tipo del nuovo oggetto chiamante, verrà attivata l'implementazione di addMoney() presente in Money o MoneyBag, rispettivamente l'implementazione di addMoneyBag() presente in Money o MoneyBag. Il tutto rispecchiando le quattro combinazioni esistenti.

Spostamento dei metodi

Sembra tutto chiaro e tutto bello, ma non è ancora finita. Se osserviamo l'implementazione dei due metodi (e, soprattutto, se proviamo a compilarli), ci accorgiamo di un problema. Chi

chiama addMoney() e addMoneyBag()? Non oggetti Money e MoneyBag, ma oggetti generici IMoney. Questo significa che IMoney dovrebbe contenere nella sua interfaccia questi due metodi.

```
public interface IMoney
{
  public IMoney add(IMoney aMoney);
  public IMoney addMoney(Money m);
  public IMoney addMoneyBag(MoneyBag mb);
}
```

Ora tutto compila e tutto funziona come ci si aspettava, ma c'è una contraddizione. Volevamo una soluzione pulita per il doppio dispatching e l'abbiamo ottenuta. Il doppio dispatching però ci serviva per poter mostrare un unico metodo add() in IMoney, ma con questa soluzione abbiamo dovuto inserire due ulteriori metodi addXYZ() in IMoney, anche se questi servono unicamente all'interno di add()… Come sbloccarci da questa situazione?

Sfruttiamo anche in questo caso una classe intermedia AbstractMoney, dove spostiamo i due metodi in questione. Lo scopo di AbstractMoney è un po' diverso da quello avuto nel dispatching manuale. In quel caso ci serviva per condividere l'implementazione di add() tra Money e MoneyBag. Qui, invece ci serve per dichiarare in un luogo neutro, in comune tra le due classi in questione, i metodi addMoney() e addMoneyBag(), senza doverli esporre in IMoney. Le implementazioni esistenti dei metodi rimangono però invariate, diverse nelle rispettive classi.

```
abstract class AbstractMoney implements IMoney
{
  abstract IMoney addMoney(Money m);
  abstract IMoney addMoneyBag(MoneyBag mb);
}
```

Ora dobbiamo solo fare in modo che le implementazioni di add() in Money e MoneyBag non si riferiscano più a metodi di IMoney, ma a metodi di AbstractMoney. Per ottenere questo, permettendo il passaggio dei parametri come IMoney, per restare generici, non resta che far uso di un "casting".

```
class Money extends AbstractMoney
{
  …

  public IMoney add(IMoney m) {
    AbstractMoney money = (AbstractMoney)m;
    return money.addMoney(this);
  }
}
```

```
class MoneyBag extends AbstractMoney
{
  …

  public IMoney add(IMoney m) {
    AbstractMoney money = (AbstractMoney)m;
    return money.addMoneyBag(this);
  }
}
```

In questo capitolo…

Questo capitolo è stato lo spunto per parlare di Visitor, un pattern particolarmente interessante e utile. Attraverso la sua chiamata "a ritroso" utilizzando come parametro l'oggetto arrivato, ci ha permesso di migliorare la nostra soluzione di `add()` tra due monete, implementando, di fatto, il doppio dispatching.

Il capitolo è inoltre servito per mostrare l'inserimento di un pattern nel codice esistente. Si tratta di un'evoluzione del design partita da un refactoring, quella che chiameremmo "design by refactoring", che è stato anche il tema centrale di questo libro. Sembra quindi appropriato terminare con questo capitolo.

La relazione diretta tra refactoring e design, soprattutto tra refactoring e design pattern, sta alla base della programmazione agile e permette l'applicazione pratica (e non solo teorica, come spero di aver mostrato con i parecchi esempi presenti nei vari capitoli) dell'evoluzione del design.