Capitolo 19
Design by testing

Con questo capitolo introduciamo un concetto nuovo, che chiameremo "design by testing". Si tratta dell'evoluzione del design di un'applicazione partendo dal test. Non solo si parte dal test per sviluppare funzionalità, ma attraverso questo procedimento si definisce l'interfaccia verso l'esterno del sistema (API) e quindi parte del suo design. L'idea è già stata anticipata nel capitolo sul design. Per esprimerci in termini di UML, si tratta di una sorta di design che cresce e si evolve a partire dai casi d'uso.

Nel successivo capitolo 20 porteremo questo concetto all'estremo, trattando il tema del "test driven development" (TDD) divulgato da Kent Beck [Beck-2003], che propone una sorta di "design by testing" sistematico.

Adattamento del design

Torniamo al nostro problema. Per rendere compatibili i due tipi Money e MoneyBag, dobbiamo riunirli sotto un'unica interfaccia. Nel nostro caso potremmo definire una *interface* IMoney, implementata sia da Money che da MoneyBag.

L'unico metodo che per il momento serve dichiarare in IMoney è add().

```
public interface IMoney
{
  public IMoney add(IMoney aMoney);
}
```

Le due classi Money e MoneyBag dovrebbero unicamente modificare la loro definizione.

```
class Money implements IMoney
{…}

class MoneyBag implements IMoney
{…}
```

Il risultato dell'utilizzo di IMoney è quello di avere ottenuto un'interfaccia che ci permette di nascondere l'utilizzo di Money e MoneyBag. Lo scopo adesso è quello di rendere utilizzabile le nostre classi unicamente attraverso l'utilizzo di IMoney. L'utilizzatore non deve preoccuparsi riguardo a cosa corrisponda in realtà un oggetto IMoney. Naturalmente, il fatto che dietro questo semplice utilizzo ci siano due classi diverse che si combinano tra di loro significa che dobbiamo implementare le diverse combinazioni di add(), visto che ora a un Money possiamo aggiungere un altro Money oppure un MoneyBag. Stessa cosa per MoneyBag utilizzato come oggetto invocante. In totale abbiamo quindi quattro combinazioni di add().

Test

La realizzazione dei quattro metodi richiede alcune considerazioni di design. Iniziamo allora prima con un passaggio più semplice, che ci servirà in seguito alla definizione dei metodi. Iniziamo cioè con l'implementazione dei metodi di test, che oltre a essere meno complessi ci permettono di capire meglio quali saranno le combinazioni in gioco e come vogliamo che queste si comportino. Iniziamo con la somma tra due Money:

```
public void testAddTwoMoneys(){
  Money m12CHF = new Money(12, "CHF");
  Money m7EUR = new Money(7, "EUR");
  MoneyBag expected= new MoneyBag(new Money[]{m12CHF, m7EUR});
  assertEquals(expected, m12CHF.add(m7EUR));
}
```

Il metodo è chiaro. Vengono creati due elementi Money con divisa monetaria diversa. La loro somma deve generare come risultato un MoneyBag. Il secondo caso è quello dell'aggiunta di un Money a un MoneyBag:

```
public void testAddMoneyToMoneyBag(){
  Money m12CHF = new Money(12, "CHF");
  Money m7EUR = new Money(7, "EUR");
  Money m20USD = new Money(20, "USD");
  MoneyBag expected= new MoneyBag(new Money[]{m12CHF,m7EUR,m20USD});
  MoneyBag firstOperand= new MoneyBag(m12CHF, m7EUR);
  assertEquals(expected, firstOperand.add(m20USD));
}
```

Anche in questo caso la somma deve generare un MoneyBag. Da notare che l'ordine degli elementi non deve essere necessariamente rilevante, ma questo fatto è già stato verificato nel metodo testBagEquals().

Abbiamo poi il caso contrario, cioè quello di un MoneyBag che viene aggiunto a un Money.

```
public void testAddMoneyBagToMoney(){
  Money m12CHF = new Money(12, "CHF");
  Money m7EUR = new Money(7, "EUR");
  Money m20USD = new Money(20, "USD");
  MoneyBag expected= new MoneyBag(new Money[]{m12CHF, m7EUR, m20USD});
  MoneyBag secondOperand= new MoneyBag(m12CHF, m7EUR);
  assertEquals(expected, m20USD.add(secondOperand));
}
```

Qui vale quanto già detto in precedenza. Infine abbiamo il caso di un MoneyBag aggiunto a un altro MoneyBag.

```
public void testAddMoneyBagToMoneyBag() {
  Money m12CHF = new Money(12, "CHF");
  Money m5CHF = new Money(5, "CHF");
  Money m20EUR = new Money(20, "EUR");
  Money m7USD = new Money(7, "USD");
  MoneyBag firstOperand = new MoneyBag(new Money[]{m7USD, m12CHF});
  MoneyBag secondOperand = new MoneyBag(new Money[]{m20EUR, m5CHF});
  MoneyBag expected = new MoneyBag(new Money[]{m12CHF, m7USD, m20EUR, m5CHF});
  assertEquals(expected, firstOperand.add(secondOperand));
}
```

Quest'ultimo caso è interessante anche perché ci rende attenti sul problema della semplificazione. Un Money di 12 CHF aggiunto a un MoneyBag esistente contenente tra le altre cose 5 CHF deve andare a formare un Money di 17 CHF che sostituisce il Money di 5 CHF all'interno del MoneyBag iniziale.

Realizzazione

Con i test abbiamo specificato esattamente cosa ci aspettiamo dall'applicazione.

La realizzazione di add() richiede un paio di spiegazioni supplementari. Il problema è quello di riuscire a gestire le 4 combinazioni nel modo più semplice possibile. In generale, partiamo da un oggetto di tipo IMoney, al quale vogliamo aggiungere un altro IMoney. Desideriamo poter chiamare il metodo nel modo più generico possibile, partendo da una definizione del genere:

```
public IMoney add(IMoney aMoney);
```

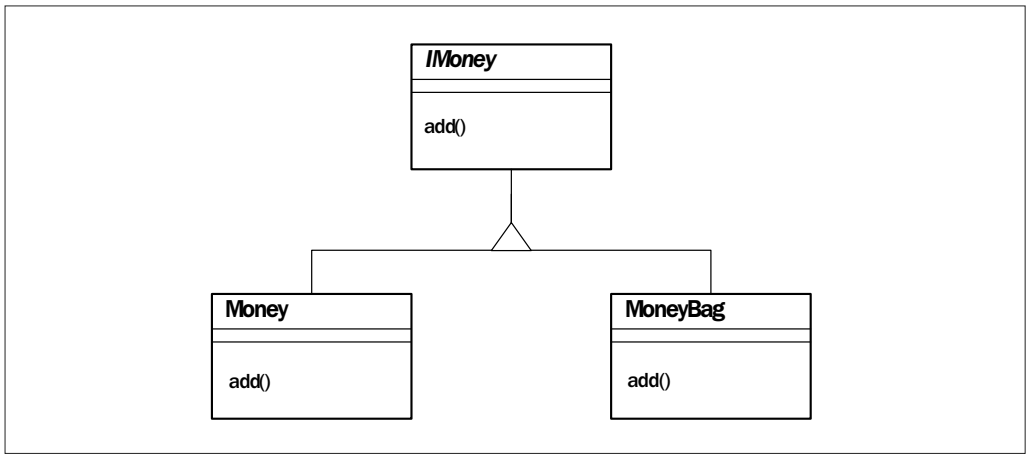

Figura 19.1 – Metodo add() dichiarato in IMoney.

Questo ci permetterebbe di avere un unico metodo in interfaccia, così come già dichiarato in IMoney. Vogliamo sia il sistema ad occuparsi di eseguire il metodo corretto, a seconda dell'oggetto che invoca e di quello passato come parametro.

Sappiamo che il *dispatching* associato al polimorfismo utilizzato dal sistema ci permette di eseguire una chiamata di questo tipo:

```
IMoney firstOperand = …
…
IMoney result = firstOperand.add(…);
```

L'oggetto firstOperand, definito come IMoney, può essere sia un Money che un MoneyBag; sarà il sistema a occuparsi di chiamare il metodo add() corretto. La cosa non funziona però con il parametro.

Dispatching semplice

Cerchiamo di approfondire un attimo il concetto. Analizziamo dapprima il caso più semplice e consideriamo di poter aggiungere a un IMoney (Money o MoneyBag), unicamente un oggetto semplice, cioè un Money.

Questo caso, più semplice di quello previsto per la versione finale, genera solo due combinazioni: un metodo add(Money) per la classe Money e un metodo add(Money) per la classe MoneyBag. Entrambi i metodi restituiscono un oggetto del tipo comune IMoney.

```
public IMoney add(Money money){
  …
}
```

Il metodo può essere dichiarato in IMoney e chiamato in modo del tutto trasparente. Sarà il sistema a occuparsi di selezionare il metodo corretto, a seconda del tipo effettivo dell'oggetto invocante.

Dispatching doppio

Purtroppo la stessa cosa non vale con il parametro. Java implementa il dispatching semplice, quindi unicamente sull'oggetto invocante. In altre parole: sull'oggetto invocante viene applicato un dispatching automatico, mentre sui parametri il dispatching deve essere gestito dallo sviluppatore.

Torniamo al nostro caso iniziale, e reale, dei quattro metodi add(), due implementati in Money (uno con Money come parametro, l'altro con MoneyBag) e due implementati in MoneyBag. Ogni classe ha quindi due metodi del genere:

```
public IMoney add(Money m){
  …
}

public IMoney add(MoneyBag m){
  …
}
```

Ebbene non è pensabile avere in interfaccia (IMoney) una dichiarazione di metodo del genere

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

e aspettarsi che venga scelto automaticamente quello giusto tra i quattro metodi a disposizione.

In altre parole, in Java non possiamo sperare di avere un unico add() in interfaccia senza intervenire a controllare il tipo esatto dell'elemento parametro. Vogliamo un unico add() per non dover mostrare in interfaccia le classi Money e MoneyBag. Lo scopo, ricordiamolo, è infatti quello di lavorare, all'esterno, unicamente con IMoney.

Se non avessimo il problema di esporre Money e MoneyBag in interfaccia, basterebbe utilizzare il meccanismo di overloading. Definiremmo due metodi in interfaccia nel modo seguente:

```
public IMoney add(Money money);
public IMoney add(MoneyBag moneyBag);
```

Questi metodi andrebbero poi implementati sia in Money che in MoneyBag. Come detto, però, non è quello che vogliamo, anche perché ciò non ci permette di lavorare con l'astrazione IMoney, che abbiamo inserito proprio per non richiedere all'oggetto client di doversi occupare della differenza tra Money e MoneyBag. In questo caso, invece, ad ogni chiamata di add(), l'utilizzatore della API dovrebbe specificare il tipo esatto dell'operando da passare come parametro.

Analizzeremo due modi per trarci d'impaccio da questa situazione, permettendo di lavorare in interfaccia solo con oggetti IMoney. Il primo, che vedremo subito, implementa una sorta di dispatching manuale, il secondo, usato in casi più complessi, prevede l'utilizzo del pattern Visitor. Questo secondo caso verrà trattato più avanti, in un capitolo separato, nella parte del libro riguardante il refactoring, perché si tratta di un buon esempio di "design by refactoring".

CLOS e il doppio dispatching

Abbiamo detto che Java non permette il doppio dispatching. In realtà non sono molti i linguaggi che lo permettono. Uno di questi è CLOS (Common Lisp Object System), la versione standard object oriented di Lisp. In CLOS i metodi non sono legati a una classe, ma sono raggruppati in gruppi a formare le cosiddette *generic function*. Tutti i metodi della stessa generic function devono avere lo stesso numero di parametri. La definizione della generic function serve unicamente a mostrare in modo esplicito l'interfaccia verso l'esterno dei suoi metodi.

```
(defgeneric add (a b)
)
```

Questa generic function specifica il nome della funzione e il numero di parametri (senza tipo) richiesti dai metodi che implementano questa interfaccia.
Lisp non è un linguaggio *strong typed*, perciò un metodo non è obbligato a specificare il tipo dei suoi parametri. Lo fa per distinguere un metodo da un altro all'interno della stessa generic function. È prassi far corrispondere il primo parametro al tipo dell'oggetto che si ritiene l'oggetto *invoker*, per ricostruire a livello di metodo la gerarchia di classi a cui i metodi corrispondono. In CLOS un parametro viene tipizzato inserendolo in una lista il cui primo elemento rappresenta il nome del parametro stesso e il secondo il tipo a cui il parametro appartiene (nel nostro caso le classi Money o MoneyBag).

```
(defmethod add ((invoker Money) operand)
  …)

(defmethod add ((invoker MoneyBag) operand)
  …)
```

Il funzionamento di chiamate è identico a quello di un qualsiasi linguaggio OO; ciò significa che il sistema esegue il dispatching sul parametro, visto che i metodi non sono legati alle classi. La cosa interessante è che non lo esegue solo sul primo, contenente l'oggetto invoker, ma anche sugli altri, realizzando di fatto un sistema multi-dispatching.
In CLOS basterebbe quindi implementare i quattro metodi add() con le diverse combinazioni di parametri. L'interfaccia sarebbe comunque generica, rappresentata dalla definizione della generic function. Ecco le quattro combinazioni:

```
(defmethod add ((invoker Money) (operand Money))
  …)
```

```
(defmethod add ((invoker MoneyBag) (operand Money))
  …)

(defmethod add ((invoker Money) (operand MoneyBag))
  …)

(defmethod add ((invoker MoneyBag) (operand MoneyBag))
  …)
```

Soluzione: dispatching manuale

Il dispatching manuale, nel nostro caso semplice, può essere considerato una soluzione accettabile. Si tratta di controllare il tipo del parametro che arriva ad add() e poi prendere la decisione su quale metodo specifico implementare.

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

Il metodo è unico, quindi in comune tra Money e MoneyBag. Siccome IMoney è un'*interface*, non possiamo implementare il metodo in IMoney. Abbiamo due possibilità: o far diventare IMoney una classe astratta, oppure inserire una classe astratta tra IMoney e le due classi concrete. Scegliamo quest'ultima variante e implementiamo il metodo in AbstractMoney.

L'implementazione dei veri metodi che eseguono la somma all'interno di Money e MoneyBag potrebbe essere fatta con overloading di add(), ma visto che questi metodi restano interni e non saranno mai visibili all'esterno, conviene per praticità e leggibilità del codice dare loro un nome più esplicito: addMoney() e addMoneyBag(). Ecco le loro implementazioni in Money

```
public IMoney addMoney(Money m){
  if (m.getCurrency().equals(getCurrency())){
    return new Money(getAmount()+m.getAmount(), getCurrency());
  }
  return new MoneyBag(this, m);
}

public IMoney addMoneyBag(MoneyBag b){
  return b.addMoney(this);
}
```

Ed ecco di seguito le versioni corrispondenti in MoneyBag. Da notare che MoneyBag deve mettere a disposizione un paio di costruttori in più rispetto a quelli già visti.

```
public IMoney addMoney(Money m) {
  return new MoneyBag(m, this);
}

public IMoney addMoneyBag(MoneyBag b) {
  return new MoneyBag(b, this);
}
```

Resto del codice

Ecco, per completezza, il resto del codice necessario al funzionamento corretto dei metodi addMoney() e addMoneyBag() in MoneyBag. Innanzitutto mancano i due costruttori usati nei metodi.

```
MoneyBag(Money m, MoneyBag mb){
```

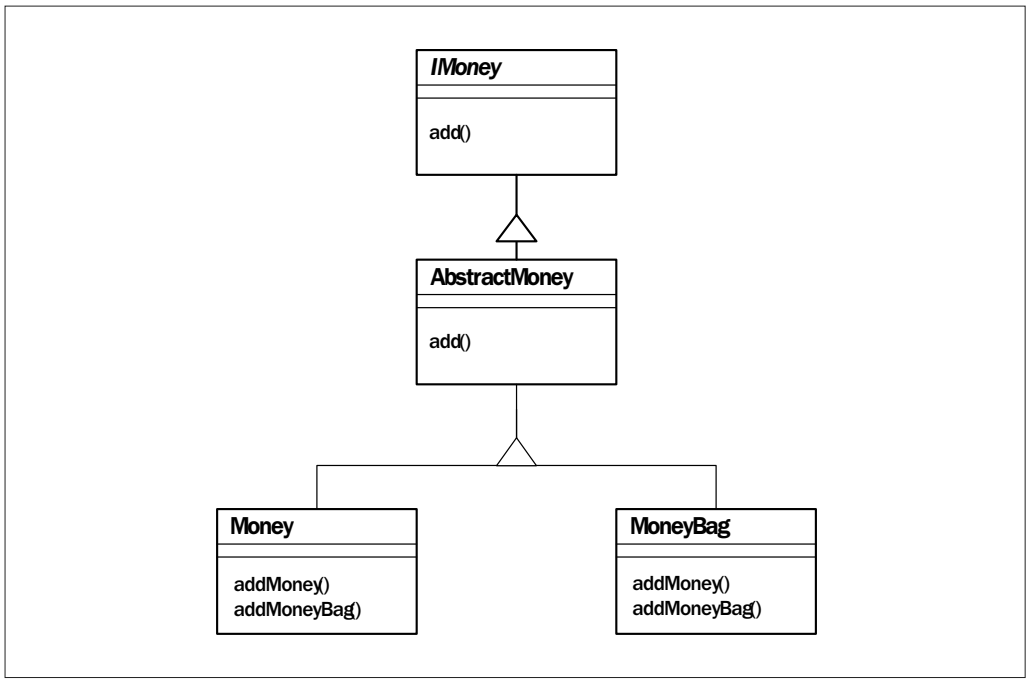

Figura 19.2 – Classe astratta intermedia.

```
    appendMoneyBag(mb);
    appendMoney(m);
  }

  MoneyBag(MoneyBag mb1, MoneyBag mb2){
    appendMoneyBag(mb1);
    appendMoneyBag(mb2);
  }
```

I due costruttori sono simili e non fanno altro che chiamare i metodi appendMoneyBag() e appendMoney(), responsabili di aggiungere sequenze di Money nella lista gestita dall'oggetto MoneyBag che esegue il metodo.

```
void appendMoneyBag(MoneyBag mb) {
  Iterator iter = mb.getMoneyList().iterator();
  while (iter.hasNext()){
    appendMoney((Money) iter.next());
  }
}
```

Questo metodo esegue un'iterazione sul MoneyBag arrivato come parametro e aggiunge uno alla volta tutti gli elementi Money. La scelta di aggiungere gli elementi uno alla volta è dettata dalla necessità di eseguire due controlli.

- Il primo serve a evitare di inserire nella lista un elemento con valore zero.
- Il secondo serve invece a verificare se esiste nella lista un Money della stessa valuta di quello che passiamo. In caso affermativo i due Money vengono fusi in un unico elemento e poi aggiunti alla lista attraverso una chiamata ricorsiva ad appendMoney().

Ecco il codice che ne deriva.

```
void appendMoney(Money m){
  if (m.getAmount() != 0){
    int position = findMoneyWithSameCurrency(fMonies, m.getCurrency());
    if (position == -1){
      fMonies.add(m);
    }else{
      Money existingMoney = (Money) fMonies.remove(position);
      appendMoney((Money)m.addMoney(existingMoney));
    }
  }
}
```

```
private int findMoneyWithSameCurrency(List monies, String currency){
  for (int i = 0; i < monies.size(); i++){
    Money money = (Money) monies.get(i);
    if (money.getCurrency().equals(currency)){
      return i;
    }
  }
  return -1;
}
```

Semplificazioni

Quando a un MoneyBag contenente una sequenza di 12 franchi e 20 euro sottraiamo 12 franchi, ci aspettiamo di ottenere non un MoneyBag, ma un unico Money di 20 euro. Si tratta di controllare quando un MoneyBag contiene un unico elemento (considerando il fatto che un elemento con valore zero venga comunque già eliminato).

Vediamo il metodo di semplificazione.

```
private IMoney simplify(){
  if (fMonies.size() == 1){
    return (IMoney)fMonies.get(0);
  }
  return this;
}
```

Le chiamate vanno aggiunte dopo ogni operazione di somma in MoneyBag.

```
public IMoney addMoney(Money m){
  return new MoneyBag(m, this).simplify();
}

public IMoney addMoneyBag(MoneyBag b){
  return new MoneyBag(b, this).simplify();
}
```

Ed ecco la verifica con il corrispondente metodo di test:

```
public void testSimplify(){
  MoneyBag firstOperand = new MoneyBag(new Money[]{new Money(12,"CHF"), new Money(20,"EUR")});
  Money expected = new Money(20, "EUR");
  assertEquals(expected, firstOperand.add(new Money(-12, "CHF")));
}
```

In questo capitolo…

In questo capitolo abbiamo ancora lavorato alla soluzione del problema delle valute, introducendo nel contempo modifiche al design iniziale. L'interfaccia esterna (API) dell'applicazione è stata specificata partendo dalle chiamate ad `add()`, definite nel test.

Nel prossimo capitolo vedremo un uso sistematico di questa tecnica. Vedremo cioè come realizzare nuove funzionalità partendo sempre dal test.

Nel corso del capitolo abbiamo anche avuto l'opportunità di capire cosa si intende con doppio dispatching e come questo può essere risolto "manualmente", mantenendo semplice la API dell'applicazione.