Capitolo 18
Sviluppare con JUnit

Prendiamo ora spunto da un esempio fornito nella distribuzione di JUnit [JUnit], per realizzare alcune possibili soluzioni e mostrare il metodo di sviluppo "code a little, test a little", possibile solo con l'ausilio di tool di test che permettono di mettere velocemente in relazione l'attività di sviluppo vera e propria con l'attività di test.

Il metodo "code a little, test a little" sta a significare che dopo aver scritto alcune righe di codice (una classe, un metodo) si deve subito passare allo sviluppo del metodo di test corrispondente.

Descrizione del problema

Il problema considerato consiste nell'implementare i calcoli di base (somma e, di conseguenza, sottrazione) tra monete di diverso tipo. I calcoli aritmetici tra elementi della stessa valuta sono semplici somme numeriche. Le cose diventano però più interessanti e complesse se ci sono più valute coinvolte. Non si può semplicemente sommare e sottrarre, perché il coefficiente di cambio non è fisso, e varia a seconda del momento in cui la conversione viene eseguita. D'altra parte non vogliamo che l'istante in cui viene eseguito il calcolo diventi determinante per l'intera gestione dei conti. Vogliamo infatti poter ottenere il valore di una certa combinazione di valute, espressa in un'unica divisa monetaria considerata al valore effettivo attuale, o a quello di ieri, o di un mese fa, e così via.

Caso semplice

Chiarito il problema, vediamo adesso di procedere passo per passo per arrivare a un abbozzo concreto di soluzione.

Prima codifica

Iniziamo col definire la classe Money, che gestisce il valore di una singola moneta. Per semplicità consideriamo il valore in int. L'informazione sulla valuta viene rappresentata da una stringa di 3 lettere che descrive in modo univoco ogni divisa monetaria.

```
public class Money
{
  private int fAmount;
  private String fCurrency;

  public Money(int amount, String currency) {
    fAmount= amount;
    fCurrency= currency;
  }

  public int getAmount() {
    return fAmount;
  }

  public String getCurrency() {
    return fCurrency;
  }

  …
}
```

Alla classe Money mancano ancora un paio di metodi. Il primo è quello che ci permette di eseguire la somma tra due valori della stessa moneta. Una somma di questo tipo dà come risultato un oggetto di tipo Money, che presenta come valore intero la somma dei due operandi.

```
public Money add(Money m) {
  return new Money(getAmount()+m.getAmount(), getCurrency());
}
```

È già utile a questo punto riscrivere per Money il metodo equals() ereditato da Object. È il metodo che servirà a determinare l'uguaglianza tra diverse istanze di Money.

```
public boolean equals(Object other) {
   return ((other instanceof Money) &&
      ((Money)other).getCurrency().equals(getCurrency()) &&
      ((Money)other).getAmount() == getAmount());
}
```

Test

Abbiamo già scritto codice a sufficienza per poter iniziare con il primo test. Si tratta della verifica dell'operazione equals() implementata per la classe Money. È importante verificarla subito, perché è quella che ci permetterà più avanti di confrontare vari ogetti di tipo Money. Ecco allora il metodo testEquals(), inserito in una classe di test che potremmo chiamare MoneyTest.

```
public class MoneyTest extends TestCase
{
   ...

   public void testEquals() {
      Money m12CHF= new Money(12, "CHF");
      Money m14CHF= new Money(14, "CHF");

      assertTrue(!m12CHF.equals(null));
      assertEquals(m12CHF, m12CHF);
      assertEquals(m12CHF, new Money(12, "CHF"));
      assertTrue(!m12CHF.equals(m14CHF));
   }
}
```

Il secondo metodo di test che possiamo inserire è quello che verifica l'operazione di somma tra due elementi Money della stessa valuta.

```
public void testSimpleAdd() {
   Money m12CHF = new Money(12,"CHF");
   Money m14CHF = new Money(14,"CHF");
   Money expected = new Money(26,"CHF");
   Money result = m12CHF.add(m14CHF);
   assertEquals(expected, result);
}
```

Questo esempio mostra anche in modo chiaro la struttura generale di un metodo di test. Si parte da alcuni elementi di inizializzazione che costruiscono oggetti utilizzati nel test:

```
Money m12CHF = new Money(12,"CHF");
Money m14CHF = new Money(14,"CHF");
Money expected = new Money(26,"CHF");
```

Si passa poi all'utilizzo degli elementi da verificare, nel caso concreto alla chiamata del metodo add(), che evidentemente può anche essere inserita come espressione nella successiva chiamata a un metodo di assert, ma che qui viene separata per chiarezza:

```
Money result = m12CHF.add(m14CHF);
```

Da ultimo si esegue la sequenza di metodi assert() necessari a verificare la correttezza della funzionalità desiderata. Nel caso specifico si tratta di un'unica chiamata:

```
assertEquals(expected,result);
```

Inizializzazioni comuni

Abbiamo visto nel capitolo precedente, che esistono i metodi setUp() e tearDown() per le fixture dei test. Nel nostro caso potremmo utilizzare setUp() per le seguenti inizializzazioni, in comune tra i due metodi finora realizzati:

```
Money m12CHF = new Money(12,"CHF");
Money m14CHF = new Money(14,"CHF");
```

Il fatto che questi vengano in seguito utilizzati in più metodi di test significa che le variabili andrebbero dichiarate in modo globale e poi inizializzate in setUp(). Anche se questa è una prassi comune, evito qui di usare setUp() per non definire variabili globali da utilizzare nei test. Il motivo è quello di rendere i metodi di test più leggibili, evitando così al lettore di dover cercare nel resto del testo a che cosa si riferiscono le variabili usate.

Esecuzione

Esistono due modi per determinare quali sono i metodi di test e quali non lo sono: uno, per così dire, è esplicito e l'altro è implicito. In quello esplicito è necessario specificare quali sono i metodi di test, aggiungendoli a uno a uno alla lista di test, all'interno del metodo suite():

```
public static Test suite() {
  TestSuite suite= new TestSuite();
  suite.addTest(new MoneyTest("testEquals"));
  suite.addTest(new MoneyTest("testSimpleAdd"));
  return suite;
}
```

JUnit permette però di utilizzare un modo più semplice per raggruppare i metodi di test. Nella forma implicita, in cui viene sfruttato il meccanismo di reflection presente in Java, JUnit riconosce i metodi di test dal fatto che sono metodi public void, iniziano con test e non hanno parametri. In questo modo, più semplice, basta specificare in suite() qual è la classe di test. Il sistema è poi in grado di riconoscere i metodi e di eseguirli.

```
public static Test suite() {
  return new TestSuite(MoneyTest.class);
}
```

Con l'aiuto del metodo suite() è inoltre possibile combinare diverse sequenze di test, presenti in classi diverse, per essere eseguite tutte assieme. Nell'esempio che segue, il metodo suite() permette di eseguire i metodi di test della classe MoneyTest e della classe ApplicationTest.

```
public static Test suite() {
  TestSuite suite = new TestSuite();
  suite.addTest(MoneyTest.suite());
  suite.addTest(com.canoo.ApplicationTest.suite());
  return suite;
}
```

Utilizzo di più valute

Dopo aver verificato con i nostri test il funzionamento di base della classe Money, possiamo passare al caso più interessante, quello in cui vengono eseguiti calcoli tra elementi Money con valute diverse. Una delle condizioni iniziali è che non vogliamo utilizzare un cambio di riferimento unico. Questo ci obbliga a ritardare la conversione, registrando tutti gli elementi diversi.

Esempio

Facciamo un esempio. Supponiamo di dover sommare 10 franchi svizzeri (CHF) a 10 euro (EUR). Quale deve essere il risultato? Un contenitore ("borsa", *bag*) contenente i due valori separati. Se in questa borsa si introducono in un secondo tempo ulteriori 20 franchi il risultato deve essere una borsa contenente 30 franchi e 10 euro.

Codifica

La codifica parte con la creazione di una nuova classe rappresentante la "borsa" (MoneyBag). La classe non ha bisogno di essere pubblica, perché l'idea di MoneyBag dovrebbe essere quella di gestire i dati all'interno del programma, senza contatti diretti con l'esterno.

```
class MoneyBag
{
  private List fMonies= new ArrayList();

  MoneyBag(Money m1, Money m2) {
    appendMoney(m1);
    appendMoney(m2);
  }

  MoneyBag(Money bag[]) {
    for (int i= 0; i < bag.length; i++){
      appendMoney(bag[i]);
    }
  }

  void appendMoney(Money m){
    fMonies.add(m);
  }

  List getMoneyList() {
    return fMonies;
  }
  ...
}
```

Dobbiamo ora definire equals() anche per MoneyBag, in modo che si possa specificare un criterio di uguaglianza tra oggetti di tipo MoneyBag. Senza troppo osservare eventuali problemi di performance, ecco la soluzione più semplice per controllare l'uguaglianza tra due oggetti MoneyBag. Si tratta di verificare che gli elementi della lista del primo MoneyBag siano contenuti nella lista del secondo, non necessariamente nello stesso ordine.

```
public boolean equals(Object mb){
  if ((mb instanceof MoneyBag) && (fMonies.size() == ((MoneyBag)mb).getMoneyList().size())){
    List moneyListFromMoneyBag = ((MoneyBag)mb).getMoneyList();
    for(int i=0; i< fMonies.size(); i++){
      if (moneyListFromMoneyBag.indexOf(fMonies.get(i)) == -1){
        return false;
      }
    }
    return true;
  }
  return false;
}
```

Test

Appena implementato equals(), abbiamo già lo spunto per il prossimo test. Mostriamo il nuovo metodo testBagEquals(), ancora con tutte le inizializzazioni al suo interno, per il motivo spiegato in precedenza:

```
public void testBagEquals() {
  Money m12CHF = new Money(12, "CHF");
  Money m14CHF = new Money(14, "CHF");
  Money m7EUR = new Money(7, "EUR");
  Money m21EUR = new Money(21, "EUR");
  MoneyBag mb1 = new MoneyBag(m12CHF, m7EUR);
  MoneyBag mb1Bis = new MoneyBag(m7EUR, m12CHF);
  MoneyBag mb2 = new MoneyBag(m14CHF, m21EUR);

  assertTrue(!mb1.equals(null));
  assertEquals(mb1, mb1);
  assertEquals(mb1, mb1Bis);
  assertTrue(!mb1.equals(m12CHF));
  assertTrue(!m12CHF.equals(mb1));
  assertTrue(!mb1.equals(mb2));
}
```

Nuovo add()

Ora sappiamo cosa significa sommare due elementi Money con divisa monetaria diversa. Possiamo così cercare di modificare il metodo add() iniziale, tenendo in considerazione l'elemento currency, cioè confrontando la valuta, prima di eseguire eventualmente l'operazione.

Versione semplice (vista in precedenza):

```
public Money add(Money m){
  return new Money(getAmount()+m.getAmount(), getCurrency());
}
```

Primo tentativo "logico" di combinare monete diverse:

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

L'idea è semplice. Se le divise sono identiche, si effettua una semplice somma, come in precedenza. Se invece le divise monetarie sono diverse, si deve creare un `MoneyBag`. Non serve JUnit, in tal caso, per verificare che il metodo in questione contiene un errore poiché siamo già bloccati dal compilatore. Il metodo `add()` dichiara infatti di restituire un oggetto di tipo `Money`, ma nel secondo return restituisce un `MoneyBag` e tra i due, a livello di codice, non esiste nessuna relazione.

A questo punto si impone un redesign. Lo introduciamo per risolvere il problema posto da `add()` e lo trattiamo nel prossimo capitolo 19, in cui ci occuperemo più in generale di design che cresce e si modifica attraverso il test.

In questo capitolo…

Allo scopo di mostrare la metodologia di sviluppo "code a little, test a little", che è possibile, anzi auspicabile, con mezzi come JUnit, abbiamo sviluppato una soluzione per il problema della somma di monete, passando dalla somma semplice alla somma di due elementi con valute diverse.

Già dopo il primo metodo, siamo in grado di scrivere un test. È quello il momento migliore per farlo, perché sappiamo esattamente cosa c'è da verificare. Attraverso il test è possibile far crescere il design. Approfondiremo questo aspetto nel prossimo capitolo.