Capitolo 17
JUnit

JUnit è un piccolo programma da utilizzare a parte, o da integrare nel proprio ambiente di sviluppo (IDE). Serve a facilitare lo sviluppo e l'organizzazione di test di unità. È stato sviluppato da Kent Beck ed Erich Gamma e ha avuto una rapidissima diffusione, grazie alla sua semplicità e al grande servizio che rende allo sviluppatore.

Gli stessi sviluppatori, nel loro libro *Contributing to Eclipse* [Gamma-2003], spiegando come realizzare un plug in che permetta l'integrazione di JUnit nella piattaforma di sviluppo Eclipse, riassumono gli scopi di JUnit e il suo contributo nell'attività di test con i seguenti punti (in parte già approfonditi nel capitolo precedente):

- Scrivere test deve essere semplice. Il tempo impiegato per scrivere test non deve richiedere più tempo dell'attività manuale di test che essi rimpiazzano.

- I test devono essere automatici. I test devono essere eseguiti e verificare i risultati senza nessun intervento umano.

- I test devono essere componibili. Se voglio eseguire contemporaneamente i miei test e quelli del mio collega, la cosa deve essere semplice. In altre parole JUnit deve mettere a disposizione la possibilità di combinare diverse sequenze di test (in seguito chiamate *suite* di test).

- I singoli test devono essere isolati. Il successo o l'insuccesso di un singolo test non deve influenzare il successo o l'insuccesso di altri test.

Utilizzo di assert

Il principio che sta alla base di JUnit è il meccanismo di *assert*. Con assert si intende un predicato che restituisce un valore booleano: vero o falso.

Un singolo test è rappresentato da un metodo che esegue una parte di codice del programma e si attende un certo risultato. Il risultato non viene stampato o visualizzato su schermo, ma viene inserito in una chiamata assert(), che lo confronta con quello atteso. Il controllo del risultato fa quindi parte del test stesso.

Nell'esempio iniziale, riportato nella Parte I del libro, abbiamo implementato alcuni metodi di test nel seguente modo:

```
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;
  }
}
```

Per l'espressione sci.getTipo() ci aspettavamo i valori Sci.CARVING in un primo tempo e Sci.NORMALE in un secondo tempo. Abbiamo confrontato esplicitamente questi valori e restituito *true* o *false* come valore della funzione di test.

In JUnit la cosa non è molto diversa, ma qui abbiamo a disposizione i metodi assert(), che ci permettono di mantenere più semplice il codice, senza necessità di realizzare il confronto in modo esplicito. Inoltre JUnit fa uso dei valori restituiti dalle chiamate assert() per mostrare i risultati in modo grafico.

Ecco come si presenta il test precedente implementato per JUnit.

```
public class SciTest …
{
  ...

  public void testSetTipo(){
    Sci sci = new Sci("Rossignol AX", Sci.CARVING);
    assertEquals(sci.getTipo(), Sci.CARVING);
```

```
        sci.setTipo(Sci.NORMALE);
        assertEquals(sci.getTipo(),Sci.NORMALE);
    }
}
```

L'interfaccia grafica di JUnit è molto semplice, ma efficace. Se il risultato non è quello atteso, JUnit mostra una barra rossa, che sta a significare la presenza di un problema, altrimenti continua con la barra verde iniziale. La figura 17.1 evidenzia l'avvenuta esecuzione di quattro metodi di test. Nessun metodo ha riscontrato un problema, perciò la barra è rimasta di colore verde.

I problemi sono di due tipi: *error* e *failure*. Una failure significa che un assert all'interno di un metodo di test ha restituito un valore `false`; si tratta quindi di un vero e proprio problema di test. Un error è invece un errore che viene riscontrato nel programma, come un'eccezione non catturata, un problema di memoria, e così via. È in un certo senso più grave, perché va oltre i controlli di errore presenti nel codice di test.

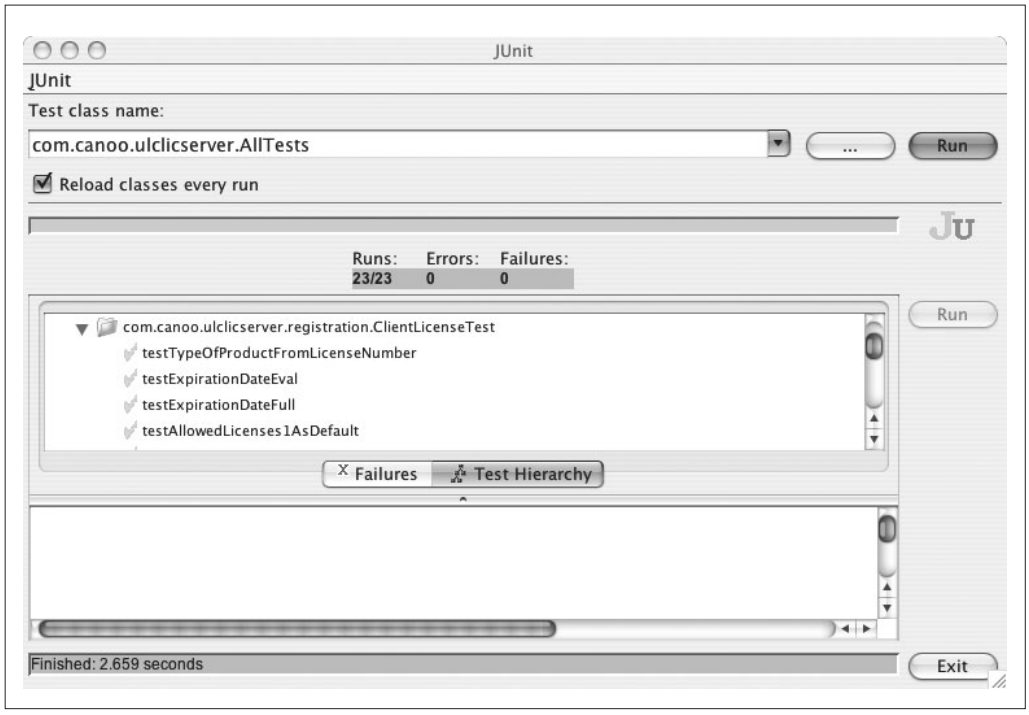

Figura 17.1 – Test riusciti, mostrati da JUnit.

Hello World

Vediamo come scrivere un primo test in JUnit con il programma "Hello World". In realtà si tratta di un "Hello World" realizzato in modo funzionale, proprio perché scrivere un programma in modo funzionale facilita la realizzazione di test.

```
public class HelloWorld
{
  String sayHello() {
    return "Hello World";
  }

  public static void main( String[] args ) {
    HelloWorld world = new HelloWorld();
    System.out.println(world.sayHello());
  }
}
```

Ed ecco come realizzare la classe di test HelloWorldTest per il programma appena definito.

```
public class HelloWorldTest extends TestCase
{
  public HelloWorldTest(String name) {
    super(name);
  }

  public void testSayHello() {
    HelloWorld world = new HelloWorld();
    assertTrue(world!=null);
    assertEquals("Hello World", world.sayHello());
  }
}
```

Come si può notare la realizzazione è molto semplice. Nello scrivere la classe di test bisogna unicamente considerare un paio di passaggi obbligati:

- La classe di test deve estendere la classe TestCase. Questa assicura l'esecuzione di alcuni meccanismi, tra cui l'implementazione dei metodi di assert e l'esecuzione dei metodi setUp() e tearDown(), che vedremo in seguito, realizzati con l'aiuto del pattern Template.

- La classe deve implementare un costruttore con un parametro stringa. Questo serve al sistema a runtime per chiamare in modo automatico i metodi. Si tratta dell'utilizzo di un pattern, il Pluggable Selector.

- Per permettere al sistema di distinguere i metodi di test da eventuali altri metodi ausiliari, i primi devono essere di tipo void, senza parametri, e il loro nome deve cominciare sempre con test.

- I migliori sistemi di sviluppo prevedono l'utilizzo di JUnit e la sua chiamata integrata nell'ambiente stesso. Se però si desidera utilizzare JUnit come elemento standalone, allora ci sono due modi per chiamare i test. Il primo prevede la definizione del metodo main() nella classe di test, l'altro l'utilizzo del metodo suite().

Chiamate

Ecco la chiamata con utilizzo del metodo main():

```
public static void main(String[] args){
  junit.swingui.TestRunner.run(HelloWorldTest.class);
}
```

La chiamata in questo caso sarebbe:

```
java HelloWorldTest
```

Un altro modo è la definizione del metodo suite() che definisce la sequenza di test da eseguire. È questo il metodo usato anche per combinare più sequenze di test da eseguire in un'unica chiamata, come avremo modo di vedere più avanti.

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

La chiamata dall'esterno diventa allora:

```
java junit.swingui.TestRunner HelloWorldTest
```

Architettura di JUnit

JUnit è un ottimo esempio di utilizzo di pattern. Nella documentazione disponibile con il prodotto, si può trovare il cosiddetto *cookbook*, cioè il libro contenente "la ricetta" del programma, nel quale viene data una descrizione chiara e dettagliata degli "ingredienti" dell'architettura utilizzando i pattern e la loro terminologia. Cercare di capire l'architettura di JUnit è quindi anche un ottimo esercizio per verificare se sono chiari i concetti legati ai pattern e, soprattutto, se si conoscono bene alcuni tra i pattern più utilizzati. Spiegata attraverso i pattern l'architettura di JUnit è abbastanza semplice.

Nel 2002 ho voluto riprendere l'architettura di JUnit per realizzare la stessa funzionalità in CLOS (Common Lisp Object System), la versione standard object-oriented di Lisp, poi presentata all'annuale "International Lisp Conference". C'erano due motivi per farlo.

> Il primo, pratico, è che avevo bisogno di scrivere, e gestire nel tempo, diverse serie di test per WordManager, scritto appunto in CLOS, prodotto per la gestione e il trattamento di informazione morfologica, progetto iniziato da Marc Domenig all'inizio degli anni Novanta, a cui ho partecipato con altri colleghi, sia durante gli anni in cui il progetto era portato avanti all'Università, sia dopo che abbiamo fondato Canoo. Il secondo motivo è che volevo mostrare ai miei studenti quanto potesse essere riutilizzabile un buon design: non solo per realizzare funzionalità simili nello stesso linguaggio, ma anche utilizzando un linguaggio completamente diverso.
> Una volta studiata l'architettura, il lavoro si è mostrato essere molto più semplice del previsto: in un solo giorno di programmazione ho potuto realizzare "clos-unit" [clos-unit] con interfaccia testuale. Non mi ritengo un programmatore particolarmente veloce. Il motivo del tempo ridotto di implementazione consiste sia nell'elevato grado di produttività di CLOS (l'implementazione di alcuni pattern, come il Pluggable Selector [Beck-1997], che in Java richiede parecchio lavoro, codice e utilizzo del meccanismo di Reflection, in CLOS risulta essere assolutamente banale), sia, soprattutto, e questo era quanto si voleva dimostrare, nell'alto livello di pulizia e descrizione del design, permessi dall'utilizzo mirato dei design pattern.

La classe Assert

La classe TestCase, da cui dobbiamo ereditare per realizzare una sequenza di test, estende a sua volta la classe Assert. Si tratta di un'eredità importante, perché da Assert vengono messi a disposizione tutti i metodi utilizzati nei predicati di test. Questo significa che la nostra classe di test ha direttamente a disposizione l'intera lista di metodi assert().

Il metodo assert() più importante è senza dubbio assertTrue(), perché è quello più generico, che permette di verificare in modo esplicito un'intera espressione booleana. Nell'esempio precedente l'abbiamo usato per controllare che il valore della variabile world fosse diversa da null:

```
assertTrue(world != null);
```

Ma avremmo anche potuto utilizzarlo per verificare l'uguaglianza tra la stringa restituita da world.sayHello() e quella attesa:

```
assertTrue(world.sayHello().equals("Hello World"));
```

Tutti i metodi di assert presentano una versione overloading con un parametro iniziale in più, in cui può essere inserito un messaggio sfruttabile dall'interfaccia utente:

```
assertTrue("Problema con l'istanza a null", world != null);
```

Oltre ad assertTrue(), esiste un'intera lista di metodi di assert, ognuno con un compito più specifico, usati per eseguire controlli di condizione, senza dover specificare l'espressione booleana in modo esplicito. Ecco l'elenco:

```
assertEquals(…)
```

assertNull(…)

assertNotNull(…)

assertFalse(…)

assertSame(…)

assertNotSame(…)

fail()

Inizializzazioni comuni

Nel caso in cui si desiderasse condividere elementi comuni di inizializzazione utilizzati in più metodi di test, JUnit prevede l'utilizzo dei metodi setUp() e tearDown(). Il metodo setUp() viene chiamato prima di ogni metodo di test, mentre tearDown() viene chiamato dopo ogni metodo di test. Vengono ereditati come metodi vuoti. Sovrascrivendoli si crea una *fixture* utilizzata in tutti i test.

Mostriamo la cosa con un esempio. Supponiamo di voler verificare il funzionamento di base della classe ArrayList, presente nelle librerie Java. Possiamo iniziare a scrivere il seguente metodo di test per valutare il funzionamento del metodo add().

```
public class ArrayListTest extends TestCase
{
   …

   public void testAddElement(){
      List list = new ArrayList();
      assertFalse(list.contains("myString"));
      list.add("myString");
      assertTrue(list.contains("myString"));
   }
}
```

Ora immaginiamo di voler scrivere il metodo testRemoveElement(). Siccome anche in questo metodo vogliamo usare un elemento ArrayList, possiamo fare in modo che questo venga dichiarato globale e inizializzato prima di ogni test nella chiamata setUp().

Prima di tutto modifichiamo la classe come segue:

```
public class ArrayListTest extends TestCase
{
```

```
    private List fList;

    protected void setUp(){
      fList = new ArrayList();
    }
    …

    public void testAddElement(){
      assertFalse(fList.contains("myString"));
      fList.add("myString");
      assertTrue(fList.contains("myString"));
    }
}
```

Ora possiamo scrivere ulteriori test che utilizzano la lista. Questa verrà ricreata in setUp() prima della chiamata di ogni metodo di test.

```
    public void testRemoveElement(){
      fList.add("myString");
      assertTrue(fList.contains("myString"));
      fList.remove("myString");
      assertFalse(fList.contains("myString"));
    }
```

Il fatto che setUp() e tearDown() vengano chiamati in modo indipendente ad ogni test garantisce l'assenza di effetti collaterali. Il metodo tearDown() è utile quando si ha a che fare con risorse esterne (file, connessioni a un database, e così via) che vanno rilasciate dopo l'utilizzo in ogni singolo test.

Condivisione e chiarezza nel codice

L'utilizzo di setUp() per centralizzare l'inizializzazione degli elementi comuni porta ad avere metodi di test più corti e quindi un codice più sintetico. Il meccanismo a disposizione di JUnit garantisce inoltre l'assenza di effetti collaterali, perché setUp() viene chiamato prima di ogni metodo di test. Viene così eliminato un possibile svantaggio che si avrebbe nel definire le variabili come globali. Questo non significa però necessariamente più chiarezza. Se è vero che le *fixture* aiutano nella leggibilità di insieme, è altrettanto riconosciuto che le ridondanze permettono di capire meglio ogni singolo metodo, anche in modo isolato dal contesto (i metodi sono, per così dire, *self contained*). Non è quindi automatico centralizzare le inizializzazioni in setUp().

Se a questo aggiungiamo il fatto che a volte l'aver definito inizializzazioni comuni in setUp() condiziona la codifica di ulteriori metodi di test (nel senso che si tende a scrivere nuovi test in modo che utilizzino tutti le stesse inizializzazioni, a costo di limitarne gli effetti), ecco che i vantaggi di setUp() vengono pesantemente relativizzati. In ogni modo sono una scelta interessante che il framework di test ci permette.

Ridurre il test all'essenziale

Ridurre il codice di test all'essenziale significa principalmente limitarsi a verificare ciò che va effettivamente verificato, senza introdurre troppe ridondanze.

Ridurre il codice di test all'essenziale vuol però anche dire fare in modo che il test sia sufficientemente veloce. Se il test dura pochi secondi, verrà richiamato spesso e il programma rimarrà sotto controllo. Se il test dura alcuni minuti, la tentazione di non chiamarlo così regolarmente aumenterà.

Un elemento di peggioramento della performance durante l'esecuzione dei test consiste nell'integrazione del proprio codice con elementi esterni. Spesso si ha bisogno di dati da un database, oppure si deve caricare un grosso file in memoria, oppure ancora interagire con un server in rete.

Quando lo scopo del test non è l'integrazione con l'esterno, ma unicamente la verifica di funzionalità proprie del programma locale, è utile ricorrere alla tecnica dei *mock objects*.

Vengono chiamati in inglese mock objects gli oggetti che sostituiscono gli oggetti reali durante il test. In Java sono solitamente oggetti con funzionalità limitata che implementano interfacce esistenti. Questi oggetti hanno un ruolo importante nella realizzazione di test, perché vengono usati per sostituire oggetti reali che richiederebbero installazioni o inizializzazioni troppo onerose, oggetti che rallenterebbero l'esecuzione del test, oppure, più semplicemente, oggetti che non esistono ancora, ma che verranno realizzati più avanti nel progetto, magari da altre persone.

Un caso tipico è quello dell'oggetto mock che sostituisce l'oggetto che ha il compito di dialogare con un database. Per non attendere che il database esista e sia perfettamente funzionante, è utile costruire un oggetto mock che ne simuli il funzionamento attraverso risposte a chiamate di metodi, implementati a partire da interfacce previste per l'interazione con il database stesso. Questo permette anche di stabilire in anticipo se l'interfaccia tra database e programma è completa o richiede modifiche.

Esistono librerie (ad esempio jmock.org) che permettono la generazione di oggetti mock a partire da interfacce esistenti.

In questo capitolo…

In questo capitolo abbiamo introdotto l'utilizzo pratico di JUnit, alla base del quale ci sono i metodi di assert, perfettamente integrati nel framework, che causano il cambiamento di colorazione della barra grafica da verde a rossa, in caso di errore. Eventuali *fixture* comuni da condividere tra diversi test possono essere centralizzate in setUp() e tearDown().