IL FRAMEWORK JUNIT

Di Mauro Lorenzutti

1. Cos'è JUnit?

JUnit è un framework open source per effettuare il testing in modo organizzato e semplice.
Come si può intuire dalla "J" è un framework scritto in Java e quindi gli esempi proposti saranno tutti in Java. Comunque JUnit è una istanza della classe xUnit, cioè esistono conversioni per un gran numero di linguaggi, fra i quali anche C#, C++ e Visual Basic.

È stato sviluppato da Erich Gamma e Kent Beck ed è scaricabile da
http://download.sourceforge.net/junit o dalla home page http://www.junit.org

2. Test unitari e test funzionali

JUnit è nato per la scrittura di test unitari.
I **test unitari** sono dei test che vanno a verificare la correttezza direttamente del codice, in ogni sua piccola parte. Questi test, che hanno avuto una grande diffusione all'interno della extreme programming, vanno scritti ed eseguiti dal programmatore stesso per ogni porzione di codice da lui sviluppata.
I **test funzionali**, invece, sono dei test che vanno a verificare che il software nella sua completezza funzioni correttamente. Questi test trattano il sistema come se fosse una scatola nera alla quale danno degli input e verificano la correttezza degli output. Devono essere ideati ed eseguiti da personale che non ha partecipato alla costruzione del software.

3. Perché non usare System.out.println

Un'alternativa all'uso dei test unitari qui presentati è sicuramente l'inserimento nel codice prodotto di comandi il cui unico scopo è di effettuare debugging. La soluzione più semplice per questo scopo è sicuramente l'inserimento di comandi tipo `System.out.println` atti a stampare variabili critiche per poterle analizzare. Questa soluzione presenta però almeno due grossi inconvenienti: innanzitutto è necessario andare a modificare il codice sorgente ogniqualvolta si voglia tracciare una nuova variabile, in secondo luogo è necessario analizzare l'output alla ricerca di eventuali errori manualmente e ad ogni esecuzione. Inoltre va sottolineato che per distribuire il progetto è necessario eliminare tutti i comandi inseriti nel codice per il testing e, soprattutto, qualora si debba apportare ulteriori modifiche ai sorgenti e si voglia testarli nuovamente è necessario reinserire tutti i comandi precedentemente cancellati.
Se per programmi brevi e semplici come quello analizzato può anche essere una soluzione valida, sicuramente per progetti di grosse dimensioni diviene difficile e laboriosa tale gestione. JUnit, invece, permette una maggiore strutturazione dei test, facilità nell'inclusione di nuovi test o nella eliminazione di altri e soprattutto un'immediata lettura dei risultati. Infatti, come vedremo, la risposta dopo l'esecuzione di una batteria di test in caso di successo è un semplicissimo `OK`. Questo velocizza non di poco l'esecuzione e la verifica dei test. Inoltre, strutturando correttamente le classi adibite al testing, sarà possibile distribuire il codice prodotto privato del codice di testing senza

apportare alcuna modifica e, se in un secondo momento si dovranno ripetere i test o aggiungerne di altri, non sarà necessario riscriverli ma semplicemente rieseguirli.
Pertanto, sebbene il lavoro richiesto per la stesura dei test sfruttando JUnit possa sembrare eccessivo ad una prima analisi, è assolutamente consigliato ricorrere a questa metodologia in quanto garantisce il corretto funzionamento del codice prodotto (ovviamente limitatamente ai test effettuati!).

4. Scrittura del primo test (corretto)

(*per la classe* `Importo` *vedere i sorgenti nel package* `importo_progetto`)
Innanzitutto è necessario creare una classe preposta all'esecuzione dei test (*nel package* `importo_test`):

```
package importo_test;
public class TestImporto extends TestCase{

private Importo i1;
private Importo i2;
```

Tale classe estende la sopraclasse `TestCase` definita all'interno del framework di JUnit e quindi è anche necessario effettuare l'import del framework stesso:

```
import junit.framework.*;
```

Inoltre è necessario importare tutte le classi che si vuole testare, classi che in questo esempio fanno parte del package `importo_progetto`:

```
import importo_progetto.*;
```

Quindi va definito un costruttore con un parametro di tipo `String` che non deve fare altro che richiamare il costruttore di `TestCase` con lo stesso parametro. In seguito verrà spiegato il perché di tale parametro.

```
public TestImporto(String name){
      super(name);
}
```

A questo punto è già possibile pensare ai test da effettuare. Per prima cosa vanno creati gli oggetti da testare, aprire le connessioni ai file necessari ecc. Tutte queste inizializzazioni vanno eseguite all'interno del metodo `setUp`, un metodo definito dalla classe `TestCase` ma lasciato vuoto. Il programmatore deve sovrascrivere tale metodo nella sua classe:

```
public void setUp(){
      i1 = new Importo(true, 12, 16);
      i2 = new Importo(false, 1, 99);
}
```

L'unico scopo di tale metodo è quindi di inizializzare quanto serve per il testing. Analogamente esiste un metodo preposto alla chiusura di tutte le connessioni aperte e alla distruzione di tutti gli oggetti creati appositamente per il testing: il metodo `tearDown`. In questo esempio tale metodo è vuoto poiché non c'è nulla da chiudere dopo la terminazione del testing se non i due oggetti `i1` e `i2` la cui eliminazione viene lasciata al Garbage Collector.

```
public void tearDown(){}
```

Non rimane che definire il test vero e proprio:

```
public void testToString(){
      assertEquals(i1.toString(), "12,16");
      assertEquals(i2. toString (), "-1,99");
}
```

Questo metodo è stato definito per effettuare due test sul metodo `toString` della classe `Importo`. Da notare soprattutto che il **nome del metodo** inizia con la stringa "test" seguito dal nome del metodo da testare. Questo nome non è affatto casuale ed in seguito verrà spiegato il perché di tale convenzione.
Analizzando le operazioni svolte dal metodo si notano due chiamate al metodo `assertEquals`. Tale metodo, definito sempre nel framework JUnit, verifica che i due argomenti a lui passati siano "uguali". Il concetto di uguaglianza è però, per così dire, ampio: nel caso si testi l'uguaglianza di due valori verrà verificata l'uguaglianza stretta (a == b), nel caso invece si testi l'uguaglianza di due oggetti verrà verificata la loro equivalenza (a.equals(b)).
Nel caso in cui almeno una delle due equivalenze non sia verificata, come vedremo più avanti, il test darà esito negativo.

Definiti i metodi di testing manca solamente da creare una suite contenente tutti i test che si vogliono effettuare alla prossima esecuzione:

```
public static Test suite(){
      TestSuite suite = new TestSuite();
      suite.addTest(new TestImporto("testToString"));
      return suite;
}
```

Come prima cosa viene creato un oggetto di tipo `TestSuite` che conterrà un elenco di tutti i metodi da richiamare durante la fase di testing. Nell'esempio qui proposto si può vedere come a tale oggetto venga aggiunto un solo test che riguarda, ovviamente, il metodo precedentemente definito `testToString`. Da notare la particolare modalità con cui tale metodo viene aggiunto all'elenco: viene creata un'istanza di `TestImporto` che si occuperà del testing del metodo il cui nome è passato come parametro al costruttore. Tale tecnica si basa sulla *Reflection*.

La classe `TestImporto` è ora pronta per l'esecuzione dei test.

5. Esecuzione del primo test in modalità testuale (corretto)

Per eseguire i test fin qui definiti è sufficiente sfruttare la classe `TestRunner` definita sempre all'interno di JUnit. Tale classe accetta come argomento una sottoclasse di `TestCase`. Pertanto il modo con cui verrà richiamata è:

```
java junit.textui.TestRunner importo_test.TestImporto
```

L'output di tale chiamata sarà:

```
.
Time: 0

OK (1 test)
```

Ovvero il test non ha rilevato nessun errore. Il tempo impiegato, pari a 0 secondi, non deve stupire data la banalità delle classi in gioco.
Da notare che prima di riportare il tempo impiegato è stato stampato un punto. Questo punto indica che è stato eseguito un unico test: se fossero stati eseguiti più test JUnit avrebbe stampato un punto dopo ogni test per avvisare il programmatore dell'avanzamento della fase di testing.

6. Scrittura del secondo test (errato)

Proviamo ora a modificare i test per scoprire un bug che affligge il codice fin qui prodotto:

```
public void setUp(){
      i1 = new Importo(true, 12, 1);
}


public void testToString(){
      assertEquals(i1.toString(), "12,01");
}
```

Una nuova esecuzione del test dovrebbe andare a buon fine ma, come vedremo, genererà invece un fallimento.

7. Esecuzione del secondo test in modalità testuale (errato)

Richiamiamo nuovamente il `TestRunner`:

```
java junit.textui.TestRunner importo_test.TestImporto
```

Stavolta l'output sarà:

```
.F
Time: 0
There was 1 failure:
1) testToString(importo_test.TestImporto)junit.framework.ComparisonFailure:
expected:<.....> but was:<...0...>
      at importo_test.TestImporto.testToString(TestImporto.java:70)
      at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
      at sun.reflect.NativeMethodAccessorImpl.invoke(Unknown Source)
      at sun.reflect.DelegatingMethodAccessorImpl.invoke(Unknown Source)

FAILURES!!!
Tests run: 1,  Failures: 1,  Errors: 0
```

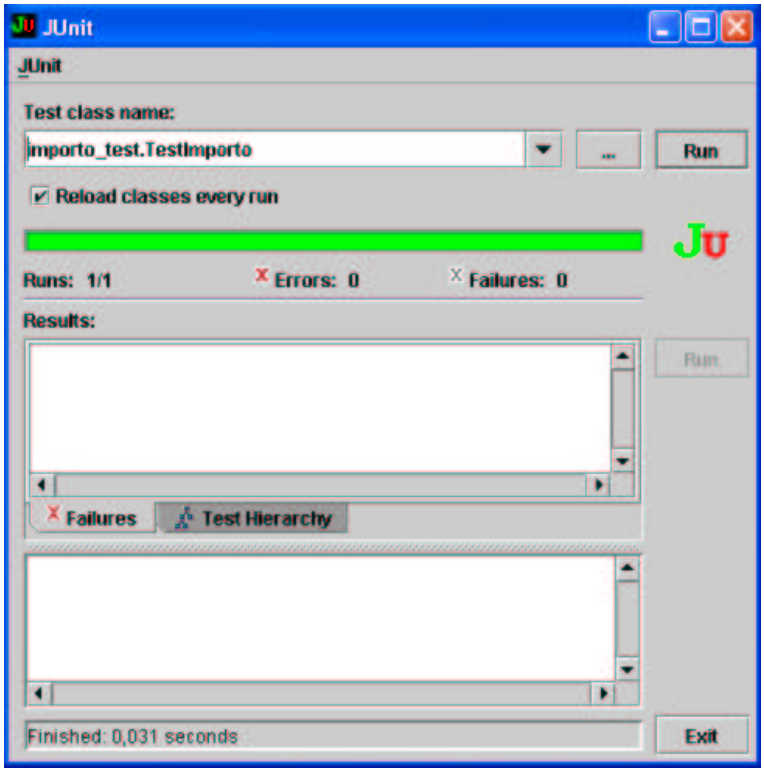
Come si può subito vedere il test è fallito. Il messaggio che riporta il motivo del fallimento, però, non è molto chiaro. Ciò che si può estrapolare da tale output è che il fallimento è stato causato dal metodo `testToString`, della classe `TestImporto`, in quanto il test di uguaglianza non si aspettava uno 0 all'interno della stringa come invece è stato richiesto.
In precedenza è stato sottolineato come il maggior vantaggio derivante dall'utilizzo di JUnit sia la velocità con cui è possibile verificare la correttezza dei test effettuati. Ovviamente, però, se i test non vanno a buon fine è necessario trovare il motivo di tale fallimento e quindi JUnit riporta un lungo e completo, ma forse complesso, messaggio di errore, uno per ogni test fallito.

8. Esecuzione del primo test in modalità grafica (corretto)

JUnit mette a disposizione del programmatore anche un'interfaccia grafica mediante la quale verificare il corretto funzionamento durante la fase di testing. Per utilizzare questa interfaccia grafica è sufficiente richiamare il `TestRunner` nel modo seguente:

```
java junit.swingui.TestRunner importo_test.TestImporto
```

A questo punto si aprirà la finestra che segue:

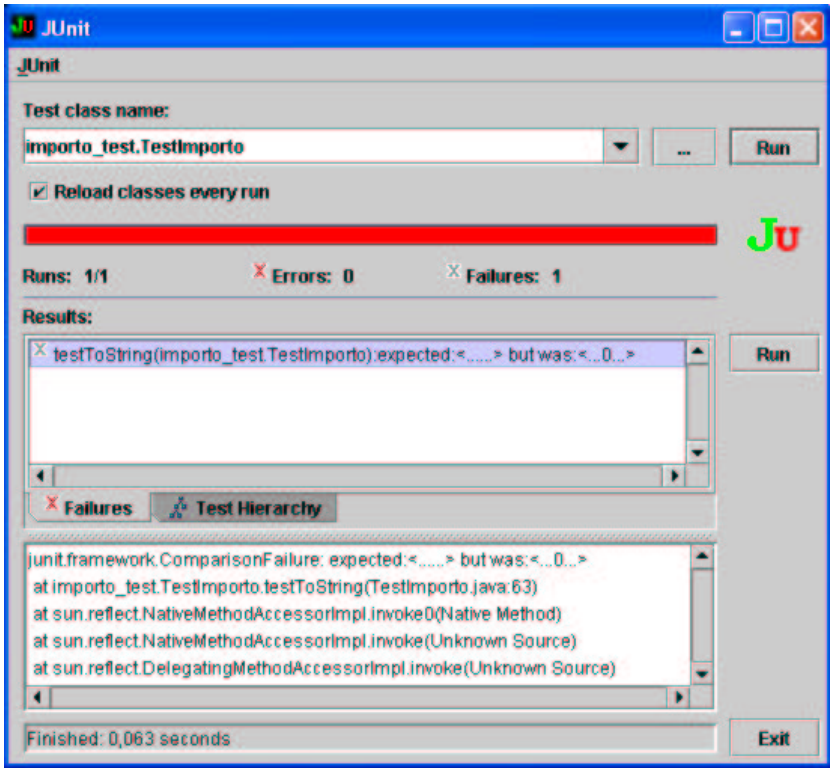


In questa finestra vengono presentati esattamente gli stessi dati riportati in forma testuale dopo il primo test. Infatti il test ora eseguito è quello precedente la modifica che portava a generare il fallimento e pertanto, come si può vedere, è giusto che non riferisca di alcun errore. Nella parte alta della finestra è presente una barra verde che indica la corretta esecuzione di tutti i test e che svolge la stessa funzione dei punti a livello testuale: è una pura e semplice barra di scorrimento.

Ciò che rende molto comoda da usare questa interfaccia è il fatto che non è necessario chiuderla dopo l'esecuzione di ogni test ma è sufficiente apportare le modifiche desiderate al codice sorgente e quindi cliccare nuovamente sul pulsante "Run". JUnit si farà carico di ricompilare le classi interessate ed eseguire il test tenendo conto delle modifiche apportate.

9. Esecuzione del secondo test in modalità grafica (errato)

Vediamo ora quale sarà l'output della finestra nel secondo test, quando cioè non è verificata l'uguaglianza. Da notare che in questo caso è sufficiente modificare il sorgente, come fatto in precedenza, e cliccare sul pulsante "Run" senza preoccuparsi di rilanciare JUnit.



Anche in questo caso le informazioni riportate sono le stesse. Ora però la barra di scorrimento è di colore rosso per evidenziare il fallimento del test.

10. Varianti di scrittura

Ovviamente questo non è l'unico modo per eseguire dei test sfruttando JUnit. Un'alternativa, che evita l'obbligo di richiamare JUnit da linea di comando, è di inserire nella classe di test, `TestImporto`, un metodo `main` all'interno del quale viene richiamato JUnit. Anche in questo caso è possibile optare per la soluzione testuale o per quella grafica. Nel codice sotto riportato viene richiamata la versione testuale:

```
public static void main(String[] args){
      junit.textui.TestRunner.run(suite());
}
```

Invece nel codice che segue si apre l'interfaccia grafica:

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

In questo caso per eseguire i test è sufficiente digitare nel prompt il seguente comando:

```
java importo_test.TestImporto
```

Verranno presentati i risultati dell'operazione di test nella modalità specificata nel `main`.

È possibile però ancora un'altra strada: si può fare a meno di costruire una suite di test. Come detto in precedenza il nome dei metodi che si occupano di effettuare il testing (es: `testToString`) deve

iniziare con la stringa “test”. Questa convenzione non è casuale: si può richiedere a JUnit di eseguire in automatico tutti i metodi il cui nome rispetta questa convenzione senza preoccuparsi di creare una suite di test. In questo caso è però necessario prestare attenzione al fatto che i metodi `setUp` e `tearDown` non verranno richiamati e quindi la creazione e distruzione di oggetti, connessioni a file ecc., devono essere implementati all’interno del metodo di test. Di seguito viene riportato un codice equivalente ai test sopra eseguiti ma che sfrutta questa caratteristica:

```
public class TestImporto2 extends TestCase{

	public void testToString(){
		Importo i1 = new Importo(true, 12, 16);
		Importo i2 = new Importo(false, 1, 99);
		assertEquals(i1.toString(), "12,16");
		assertEquals(i2.toString(), "-1,99");
	}
}
```

Questa semplice classe esegue gli stessi test della precedente. Anche in questo caso i comandi da digitare al prompt sono gli stessi ed è anche qui possibile inserire un metodo `main` contenente la chiamata a JUnit.

11. Differenze fra failure ed error

Nei test finora eseguiti, il framework ha rilevato dei fallimenti (*failure*) quando i test di uguaglianza da noi impostati non erano verificati, ovvero quando una o più delle asserzioni specificate risultavano false. In questo caso il framework avvertiva del fallimento del test.
Ma cosa succede se invece di un fallimento si verifica un’eccezione imprevista durante il test? In questo caso JUnit termina segnalando un errore (*error*), ovvero riporta il verificarsi di una eccezione non gestita. Ovviamente questo tipo di scorrettezza non è un fallimento nel senso visto poco sopra in quanto non si tratta di una uguaglianza non verificata. Per questo motivo viene classificata come *error* e non come *failure*.

12. Eccezioni

Proviamo ora a modificare i nostri test per generare una eccezione e studiarne il comportamento. Innanzitutto è necessario modificare il costruttore della classe `Importo` al fine di sollevare una eccezione qualora i valori passati siano negativi:

```
public Importo(boolean positivo, long euro, long cent){
	if (euro >= 0 && cent >= 0){
		this.positivo = positivo;
		this.euro = euro + (cent / 100);
		this.cent = cent % 100;
	}
	else{
		throw new NumberFormatException();
	}
}
```

In questo modo se vengono passati al costruttore un numero di euro o di cent negativi, il costruttore genera una eccezione di tipo *NumberFormatException*. Ora non rimane che testare come reagisce JUnit quando, nei metodi di test, viene richiamato il costruttore in questione con valori negativi.

13. Esecuzione del test sulle eccezioni

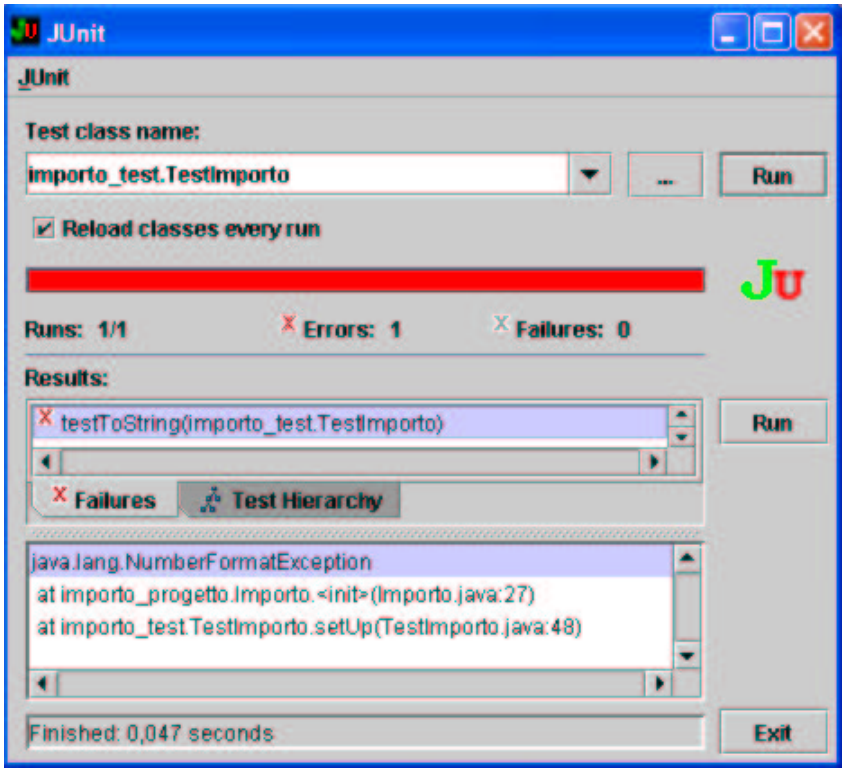
Per testare il verificarsi di un *error* è sufficiente modificare il metodo `setUp` per fargli richiamare il costruttore con un numero di euro negativo:

```
public void setUp(){
      i1 = new Importo(true, -12, 16);
      i2 = new Importo(true, 1, 99);
}
```

A questo punto è sufficiente richiamare JUnit sulla classe come fatto in precedenza e si ottiene il seguente output testuale:

```
.E
Time: 0
There was 1 error:
1) testToString(importo_test.TestImporto)java.lang.NumberFormatException
      at importo_progetto.Importo.<init>(Importo.java:28)
      at importo_test.TestImporto.setUp(TestImporto.java:51)

FAILURES!!!
Tests run: 1,  Failures: 0,  Errors: 1
```

In questo caso JUnit segnala la generazione imprevista di una eccezione avvertendo la presenza di un *error*. Scorrendo l'output si possono ricavare diverse informazioni, fra cui il tipo di eccezione sollevata (*NumberFormatException*) e il punto in cui è stata generata, sia nel metodo di test (`testToString`) sia nella classe `Importo` (`<init>` ovvero nel costruttore).
Gli stessi risultati vengono forniti qualora si opti per la modalità grafica di JUnit:



14. Test di corretto funzionamento delle eccezioni

Ovviamente questo non è il modo corretto per testare se il nostro codice gestisce correttamente le eccezioni. Se immaginiamo un progetto di dimensioni appena più grandi e che gestisce un numero non insignificante (come nell'esempio proposto) di eccezioni, è impensabile di dover scorrere l'output di JUnit per controllare che tutti i test che avrebbero dovuto generare delle eccezioni le abbiano effettivamente generate (e del tipo corretto). O comunque tale modalità di utilizzo di JUnit va contro la filosofia di JUnit stesso!
Pertanto, per risolvere questo problema, è disponibile in JUnit il metodo `fail(String)`. Questo metodo forza il fallimento della batteria di test proponendo in output la stringa passatagli come argomento. Tale metodo viene utilizzato nei test di eccezioni per far fallire i test qualora una eccezione attesa in un punto del codice non sia stata sollevata. Nella porzione di codice seguente è possibile vedere un modo corretto per testare le eccezioni:

```
public void testCostruttoreNegativo(){
	try{
		Importo i3 = new Importo(false, -10, 15);
		fail("Deve generare una eccezione");
	}
	catch(NumberFormatException nfe){}
}
```

In questo metodo, aggiunto alla classe `TestImporto` e il cui nome ovviamente inizia con "test", viene tentata la creazione di un oggetto di tipo `Importo` con valori negativi. A differenza di quanto fatto in precedenza, in questo caso si è racchiusa la creazione di tale oggetto all'interno di un `try-catch` al fine di catturarne l'eccezione generata. Da notare che dopo la creazione dell'oggetto è stata aggiunta la chiamata:

```
fail("Deve generare una eccezione");
```

Tale chiamata è necessaria qualora il costruttore della classe `Importo` non sollevi una eccezione di tipo *NumberFormatException* se gli vengono passati argomenti negativi. Infatti in questo caso il costruttore non funziona correttamente ed è necessario far fallire il test.
Proviamo ora ed eseguire nuovamente la batteria di test ma non prima di aver apportato le seguenti modifiche alla classe `TestImporto`:

- è necessario modificare il metodo `setUp` affinché non generi eccezioni:

  ```
  public void setUp(){
  	i1 = new Importo(true, 12, 16);
  	i2 = new Importo(false, 1, 99);
  }
  ```

- è necessario aggiungere alla `suite` il nuovo test (tale modifica non è richiesta se non si utilizza la suite ma si fanno eseguire in automatico tutti i metodi il cui nome inizia con la stringa "test", come nella classe `TestImporto2`):

  ```
  public static Test suite(){
  	TestSuite suite = new TestSuite();
  	suite.addTest(new TestImporto("testToString"));
  	suite.addTest(new TestImporto("testCostruttoreNegativo"));
  	return suite;
  }
  ```

A questo punto eseguiamo ancora i test ed otteniamo il seguente output:

```
..
Time: 0,015

OK (2 tests)
```

Innanzitutto si può notare come ora il test non sia fallito ma soprattutto non abbia riportato nessun errore. Ciò vuol dire che l'eccezione è stata correttamente sollevata dal costruttore e correttamente gestita dal metodo che la testava.
In secondo luogo è possibile rilevare la corretta esecuzione di due test, esattamente come specificato nella `suite`.

In questo esempio, però, il metodo `fail` può sembrare assolutamente inutile: viene sempre saltato in quanto il costruttore di `Importo` genera una eccezione che fa saltare al `catch` e quindi ignorare tutto quello che segue la chiamata al costruttore. Supponiamo invece di aver scritto male il costruttore e riportiamolo alla prima versione da noi realizzata:

```
public Importo(boolean positivo, long euro, long cent){
      this.positivo = positivo;
      this.euro = euro + (cent / 100);
      this.cent = cent % 100;
}
```

In questa versione, infatti, non viene sollevata alcuna eccezione nel caso gli argomenti siano negativi. A questo punto se eseguissimo nuovamente i test, secondo quanto detto finora, JUnit dovrebbe fallire a causa della chiamata al metodo `fail` successiva alla creazione di un oggetto `Importo` con valori negativi:

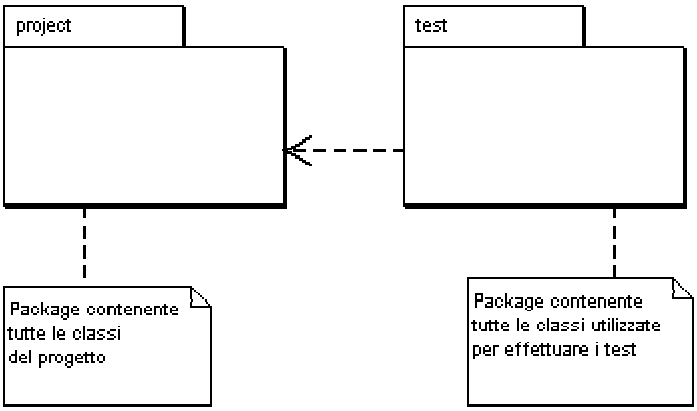
```
..F
Time: 0
There was 1 failure:
1)
testCostruttoreNegativo(importo_test.TestImporto)junit.framework.AssertionFailed
Error: Deve generare una eccezione
      at importo_test.TestImporto.testCostruttoreNegativo(TestImporto.java:78)
      at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
      at sun.reflect.NativeMethodAccessorImpl.invoke(Unknown Source)
      at sun.reflect.DelegatingMethodAccessorImpl.invoke(Unknown Source)

FAILURES!!!
Tests run: 2,  Failures: 1,  Errors: 0
```

Come si vede JUnit effettivamente termina segnalando un fallimento su due test eseguiti. Dall'output fornito si può risalire alla causa. Ovvero il metodo `testCostruttoreNegativo`. Inoltre si può vedere come effettivamente venga riportata la stringa passata come argomento al metodo `fail` ("Deve generare una eccezione").
È necessario sottolineare come JUnit sia terminato segnalando un fallimento e non un errore. Infatti in questo caso non è stata sollevata alcuna eccezione ed è proprio questo che ha portato ad un malfunzionamento nel codice.

15. Dove inserire i test

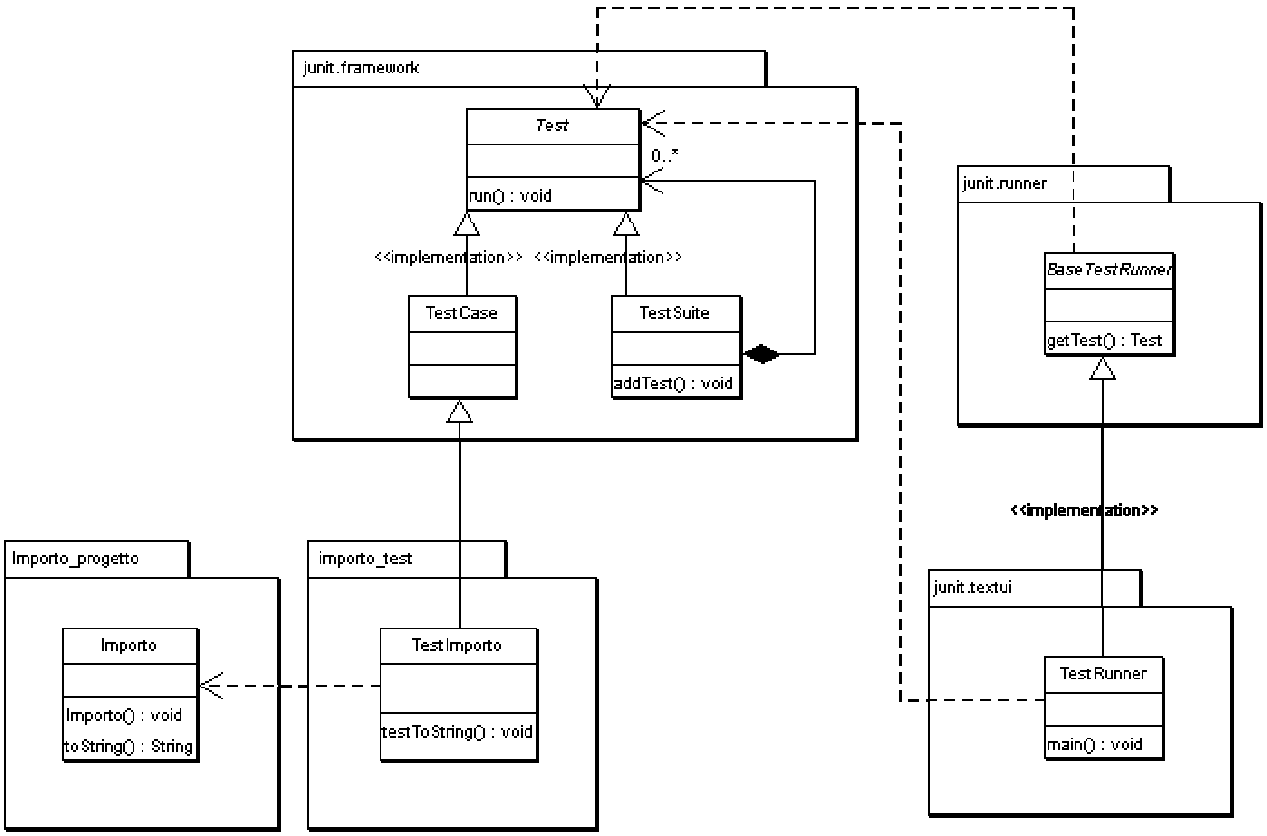
Una domanda che può sorgere a questo punto è: dove vanno inserite le classi di test? Ovviamente non ci sono restrizioni in questo senso in quanto è possibile inserirle nella stessa cartella del

progetto o in una diversa. Una buona soluzione è però la creazione di un package separato da quello del progetto. In questo modo sarà poi possibile **distribuire il progetto privato dei test senza dover apportare alcuna modifica**.



Per gli esempi qui prodotti sono stati creati due package: uno per il progetto (`importo_progetto`) ed uno per i test (`importo_test`). In tutte le classi di test è stato sufficiente importare il package `importo_progetto` per accedere liberamente alle sue classi. Qualora si volesse distribuire il progetto principale sarà pertanto sufficiente la distribuzione di `importo_progetto`.

Di seguito è riportato lo schema UML del progetto utilizzato per questa presentazione:



16. Installazione del framework

a. Scaricare l'ultima versione di JUnit da http://www.junit.org (si scarica un pacchetto .zip)
b. Installazione per Windows (per sistemi Unix-like è equivalente)
   b.1 Scompattare il file junit.zip in una cartella chiamata %JUNIT_HOME%
   b.2 Aggiornare il classpath:

```
set CLASSPATH=%CLASSPATH%; %JUNIT_HOME%\junit.jar
set CLASSPATH=%CLASSPATH%; %JUNIT_HOME%
```

c. Testare l'installazione:
   c.1 Lanciare i test in modalità testuale
   java junit.textui.TestRunner junit.samples.AllTests
   c.2 Lanciare i test in modalità grafica
   java junit.textui.TestRunner junit.samples.AllTests

   Se non funziona correttamente verificare di aver aggiornato il CLASSPATH.

Bibliografia

Blaine Simpson, JUnit HowTo, http://www.junit.org/news/article/index.htm

FAQ di JUnit, http://junit.sourceforge.net/doc/faq/faq.htm

Martin Fowler, *Refactoring, Improving the Design of Existing Code*, ADDISON-WESLEY 2000.

Paolo Perrotta, *Testare il codice con JUnit 1*, IOProgrammo n° 66, Edizioni Master, febbraio 2003.

Paolo Perrotta, *Testare il codice con JUnit 2*, IOProgrammo n° 67, Edizioni Master, marzo 2003.

Paolo Perrotta, *Extreme Programming abbracciare il cambiamento*, IOProgrammo n° 68, Edizioni Master, aprile 2003.