# Il framework JUNIT

Di Mauro Lorenzutti

mercoledì 28 aprile 2004

## Scaletta

- JUnit
- Test unitari e funzionali
- Scrittura dei test e loro esecuzione
- Failure ed error
- Test di eccezioni
- Organizzazione dei test
- Installazione del framework



## Cos'è JUnit?

- Un framework open source per effettuare il testing in modo organizzato e semplice
- Una istanza di xUnit
- Sviluppato da Erich Gamma e Kent Beck
- http://www.junit.org



## Test unitari e test funzionali

- I **test unitari** sono dei test che vanno a verificare la correttezza direttamente del codice, in ogni sua piccola parte
- I **test funzionali** sono dei test che vanno a verificare che il software nel suo insieme funzioni correttamente
- **JUnit** è nato per la scrittura di test unitari



## Perché usare JUnit

- Elevata strutturazione dei test
- Facilità nell'aggiunta/rimozione di test
- **Immediata lettura dei risultati**
- Distribuzione del codice privo dei test senza necessità di modifiche
- Possibilità di riagganciare i test in un secondo momento



## Classe da testare 1/2

```
package importo_progetto;
public class Importo {
      private final boolean positivo;
      private final long euro;
      private final long cent;

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

## Classe da testare 2/2

```
public String toString(){
      String risultato = "";
      if (!positivo)
            risultato += "-";
      risultato = risultato + euro + "," + cent;
      return risultato;
      }
}
```



## Classe di Test

```
package importo_test;
import importo_progetto.*;
import junit.framework.*;
public class TestImporto extends TestCase{
  private Importo i1;
  private Importo i2;
  public TestImporto(String name){ super(name); }
  public static Test suite(){
      TestSuite suite = new TestSuite();
      suite.addTest(new TestImporto("testToString"));
      return suite;
  }
  public void setUp(){
      i1 = new Importo(true, 12, 16);
      i2 = new Importo(false, 1, 99);
  }
  public void tearDown(){}
  public void testToString(){
      assertEquals(i1.toString(), "12,16");
      assertEquals(i2.toString(), "1,99");
  }}
```



## `TestImporto` 1/4

- I test vengono inseriti in un package diverso

```
package importo_test;
```

- Devono essere importate tutte le classi del framework JUnit e le classi da testare

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

- La classe contenente i test deve ereditare da `TestCase`, una classe predefinita di JUnit

```
public class TestImporto extends TestCase{
```



## `TestImporto` 2/4

- Gli oggetti su cui verranno eseguiti i test

```
private Importo i1;
private Importo i2;
```

- Costruttore della classe

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

- Metodo utilizzato per inizializzare gli oggetti da testare

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



## `TestImporto` 3/4

- Metodo utilizzato per distruggere quanto inizializzato mediante `setUp` dopo l'esecuzione dei test

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

- Metodo che esegue i test

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



## `TestImporto` 4/4

- Metodo che crea una suite di test (insieme di test) da effettuare

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

Basato sul pattern Composite: una suite può contenere diverse TestSuite

# Esecuzione del test 1/2

- `junit.textui` specifica che i test devono essere eseguiti in modalità testuale. La classe `TestRunner` accetta come argomento una sottoclasse di `TestCase`

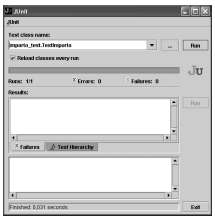

`OK` -> Nessun errore rilevato dai test ☺



# Esecuzione del test 2/2

- `junit.swingui` specifica che i test devono essere eseguiti in modalità grafica.

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





# Alla ricerca del bug 1/4

- Modifichiamo i test eseguiti al fine di scoprire l'esistenza di un bug nel codice:

```
class TestImporto …{
…
public void setUp(){
      i1 = new Importo(true, 12, 1);
   }
public void testToString(){
      assertEquals(i1.toString(), "12,01");
   }
…
}
```



# Alla ricerca del bug 2/4

- Eseguiamo nuovamente il test

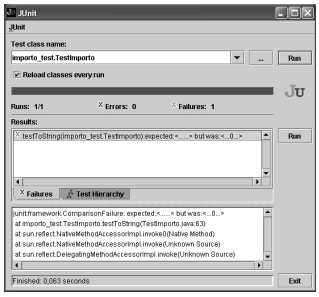
```
.F
Time: 0
There was 1 failure:
1)
testToString(importo_test.TestImporto)junit.framework.ComparisonFailure:
expected:<.....> but was:<...0...>
        at importo_test.TestImporto.testToString(TestImporto.java:70)
        at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
        at sun.reflect.NativeMethodAccessorImpl.invoke(Unknown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

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

- TEST FALLITO ☹ : L'asserzione di uguaglianza non è stata rispettata: non ha trovato nessuno 0. Ma perché?



# Alla ricerca del bug 3/4

- Eseguiamo i test in modalità grafica:




- Stessi risultati!
  Ma dov'è l'errore?



# Alla ricerca del bug 4/4

- Se il numero di centesimi è minore di 10 non viene aggiunto mai uno 0.

```
public Importo(boolean positivo, long euro, long cent){
      this.positivo = positivo;
      this.euro = euro + (cent / 100);
      this.cent = cent % 100;
}
public String toString(){
      String risultato = "";
      if (!positivo) risultato += "-";
      risultato = risultato + euro + "," + cent;
      return risultato;
}
```

## Varianti di scrittura 1/2

- È possibile inserire nel `main` della classe di test la chiamata al framework JUnit:

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

- È quindi possibile eseguire i test digitando semplicemente:

```
java importo_test.TestImporto
```



## Varianti di scrittura 2/2

- Non utilizzare la `suite` di test:
  - Nominare tutti i test da effettuare come `test`*nomemetodo_da_testare*
  - Non utilizzare suite, setUp e tearDown perché vengono ignorati
  - Eseguire normalmente (JUnit automaticamente testa tutti i metodi il cui nome rispetta la convenzione di cui sopra)

```
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");
      }
}
```

## Failure ed Error

- Si ha **failure** se i test di uguaglianza da noi impostati non vengono verificati.
- Si ha un **error** quando durante i test vengono sollevate delle eccezioni non gestite.
- Come si può testare la corretta gestione di un'eccezione?



## Generiamo un'eccezione 1/2

- Modifichiamo il costruttore della classe da testare affinché generi una eccezione nel caso venga passato un numero di euro o di centesimi negativo:

```
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();}
}
```

- Modifichiamo la classe di test affinché verifichi questa condizione:

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



## Generiamo un'eccezione 2/2

- Eseguiamo i test ed analizziamo l'output fornito da JUnit:

```
.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
```

- Si è verificato un **error**: l'eccezione è stata sollevata ma non gestita dalla classe di test.



## Testiamo le eccezioni 1/5

- Si utilizza il metodo **fail** che interrompe l'esecuzione dei test generando una failure e riporta in output la stringa passatagli come parametro.

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

## Testiamo le eccezioni 2/5

- Modifichiamo `setUp` e `suite` per eseguire correttamente i test:

```
public void setUp(){
 i1 = new Importo(true, 12, 16);
 i2 = new Importo(false, 1, 99);}
public static Test suite(){
 TestSuite suite = new TestSuite();
 suite.addTest(new TestImporto("testToString"));
 suite.addTest(new TestImporto("testCostruttoreNegativo")
 return suite;
}
```

- Eseguendo i test JUnit non rileva nessuna failure e nessun error ☺



## Testiamo le eccezioni 2/3

- Riportiamo ora il costruttore della classe `Importo` alla sua prima versione:

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

- In questo modo non viene generata alcuna eccezione.



## Testiamo le eccezioni 3/3

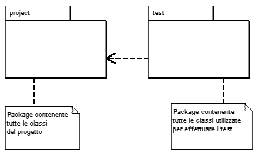
- Eseguendo ora i test, JUnit dovrebbe fallire a causa della chiamata al metodo fail:

```
..F
Time: 0
There was 1 failure:
1) testCostruttoreNegativo(importo_test.TestImporto)
    junit.framework.AssertionFailedError:
    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
```



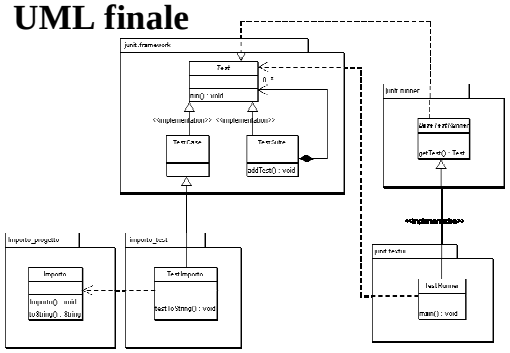
## Dove inserire le classi di test?

- Insieme al progetto (soluzione PESSIMA)
- In un package separato (soluzione IDEALE)
  - Distribuire il progetto senza i test.






## UML finale






## Non solo assertEquals!

**assertEquals(Object, Object)**
Asserts that two objects are equal.
**assertFalse(String, boolean)**
Asserts that a condition is false.
**assertNotNull(String, Object)**
Asserts that an object isn't null.
**assertNotSame(String, Object, Object)**
Asserts that two objects refer to the same object.
**assertNull(String, Object)**
Asserts that an object is null.
**assertSame(String, Object, Object)**
Asserts that two objects refer to the same object.
**assertTrue(String, boolean)**
Asserts that a condition is true.

## Installazione del framework

1. Scaricare l'ultima versione di JUnit da **http://www.junit.org** (si scarica un pacchetto .zip)
2. Installazione per Windows (per sistemi Unix-like è equivalente)
   1. Scompattare il file junit.zip in una cartella chiamata %JUNIT_HOME%
   2. Aggiornare il classpath:
      set CLASSPATH=%CLASSPATH%; %JUNIT_HOME%\junit.jar
      set CLASSPATH=%CLASSPATH%; %JUNIT_HOME%
3. Testare l'installazione:
   1. Lanciare i test in modalità testuale
      ```
      java junit.textui.TestRunner
      junit.samples.AllTests
      ```
   2. Lanciare i test in modalità grafica
      ```
      java junit.textui.TestRunner
      junit.samples.AllTests
      ```

## Riassumendo

- **Test unitari**: testare direttamente il codice
- Scrittura dei test e loro raggruppamento
- Modalità testuale e grafica
- **Failure** ed **error**
- Utilizzo del metodo **fail**
- Organizzazione dei test
- Installazione del framework



## 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.