Programmazione Orientata agli Oggetti in Linguaggio Java

Test e Correzione: JUnit

versione 2.0

Questo lavoro è concesso in uso secondo i termini di una licenza Creative Commons
(vedi ultima pagina)

SOME RIGHTS RESERVED creative commons

G. Mecca – Università della Basilicata – mecca@unibas.it

Sommario

- Introduzione
- Le Regole di JUnit
  - ⇨Fixture e SetUp
  - ⇨TestRunner e TestSuite
- Linee Guida
- Esempi di Test
- Installazione e Classpath

Introduzione

- JUnit
  - uno strumento per lo sviluppo di test in Java
  - principalmente test di unità (ma anche test funzionali)
  - e di regressione
  - uno strumento celebre, scritto da Kent Beck ed Erich Gamma, e poi riscritto per molti altri linguaggi
  - si tratta di un esempio di “framework”

Introduzione

- Framework
  - codice nato per facilitare lo sviluppo di applicazioni
  - tipicamente fornisce alcuni strumenti eseguibili (es: applicazione grafica per eseguire test)
  - e stabilisce delle regole di scrittura del codice per potere usare questi strumenti

Introduzione

>> TestStudente.java
>> junit.swingui.TestRunner

- Pillole di JUnit
  - ⇨un test viene scritto sulla base di opportune classi del package junit.framework
  - ⇨è necessario utilizzare metodi particolari per le asserzioni
- Gli strumenti di JUnit
  - ⇨due “testrunner”
  - ⇨junit.textui.TestRunner (testuale)
  - ⇨junit.swingui.TestRunner (grafico)


```
package it.unibas.mediapesata.test;

import junit.framework.*;
import it.unibas.mediapesata.*;

public class TestStudente extends TestCase {

    public void testMediaPesata() {
        Studente studente = new Studente();
        Esame esame1 = new Esame();
        esame1.setVoto(24);
        esame1.setCrediti(3);
        studente.addEsame(esame1);
        Assert.assertTrue("media p. 2", studente.getMediaPesata() == 24);
    }

    public void testMediaPesata2() {
        ...
    }
```

la classe di Test estende junit.framework.TestCase

il nome dei metodi di test deve cominciare con “test”

le asserzioni avvengono utilizzando il metodo asserTrue() ereditato da junit.framework.TestCase

Le Regole di JUnit

- Le regole di JUnit
  - una classe di test deve estendere la classe junit.framework.TestCase
  - ogni metodo di test deve essere una procedura pubblica (public void) il cui nome comincia con “test”
  - le asserzioni vengono fatte usando il metodo statico Assert.assertTrue() con due argomenti: una stringa che rappresenta il nome del test e l’asserzione da verificare

Le Regole di JUnit

- Il metodo Assert.AssertTrue()
  - se l’asserzione è vera registra sull’interfaccia l’esecuzione corretta del test (es: stampa “.”)
  - se l’asserzione è falsa, lancia junit.framework.AssertionFailedError
  - una eccezione di tipo errore che provoca il fallimento del test

Le Regole di JUnit

- Altri metodi per le asserzioni
  - ⇨Assert.assertEquals – vari sovraccarichi
    es: assertEquals(int valorePrevisto, int valoreReale)
    es: assertEquals(String prevista, String reale) ecc.
  - ⇨Assert.assertNotNull(Object rif)
  - ⇨Assert.assertNull(Object rif)
  - ⇨Assert.assertFalse(boolean asserzione)

Le Regole di JUnit

- Possibili cause di fallimento di un test
  - ⇨junit.framework.AssertionFailedError: eccezione lanciata dal framework nel caso in cui una asserzione fallisca
  - ⇨qualsiasi altra eccezione (java.lang.Throwable): durante l'esecuzione dei test si è verificato un problema inaspettato che ha causato l'eccezione

Le Regole di JUnit

- Quindi, al termine dell'esecuzione
  - ⇨il test runner riporta quattro numeri
  - ⇨numero di test eseguiti
  - ⇨numero di test completati con successo
  - ⇨numero di test completati ma falliti per via del fallimento di una asserzione
  - ⇨numero di test non completati a causa del verificarsi di una eccezione

Fixture e SetUp

ATTENZIONE
al concetto di
fixture e di setUp()

- Il concetto di "fixture" e di setUp
  - ⇨tipicamente: tutti i metodi lavorano con uno o più oggetti della classe da verificare
  - ⇨trattandosi di test di unità, ogni metodo di test deve essere effettuato in modo isolato rispetto agli altri
  - ⇨e cioè su oggetti completamente nuovi e non intaccati dai test precedenti

Fixture e SetUp

- Una possibilità
  - ciascun metodo di test crea ex-novo gli oggetti su cui eseguire i test
  - svantaggio: molto codice duplicato inutilmente per la creazione; es: Studente
- Una possibilità migliore
  - definire un gruppo di oggetti condivisi tra i test (una “fixture” – impianto, infrastruttura)
  - e ricrearli ogni volta in modo automatizzato

```
import junit.framework.*;
import it.unibas.mediapesata.*;

public class TestStudente extends TestCase {

    private Studente studente;
    private Esame esame1;
    private Esame esame2;

    public void setUp() {
        studente = new Studente();
        esame1 = new Esame();
        esame1.setVoto(24);
        esame1.setCrediti(3);
        esame2 = new Esame();
        esame2.setVoto(27);
        esame2.setCrediti(9);
    }

    public void testMediaPesata1() {
        studente.addEsame(esame1);
        studente.addEsame(esame2);
        Assert.assertTrue("media p. 1", studente.getMediaPesata() == 26.25);
    }
    ...
```

il metodo di setUp() crea un nuovo studente e due esami che sono disponibili prima di ciascun test

Fixture e SetUp

- In JUnit
  - ⇨la fixture corrisponde alle proprietà della classe di test
  - ⇨viene definito un metodo public void setUp() che inizializza le proprietà
  - ⇨il metodo setUp() viene rieseguito dal framework prima di eseguire ciascun test
  - ⇨per avere sempre una fixture “fresca”

Fixture e SetUp

- Attenzione
  - ⇨alla differenza tra setUp() ed un costruttore
  - ⇨il costruttore può inizializzare le proprietà una volta sola (alla creazione)
  - ⇨setUp() viene richiamato dal framework prima di ogni test
  - ⇨è previsto anche un metodo tearDown() nel caso in cui la fixture richieda operazioni di “pulitura” dopo il test

Fixture e SetUp

- Esecuzione del test
  - ⇨il metodo runBare() di TestCase

```
public void runBare() throws Throwable {
    setUp();
    try {
        runTest();
    } finally {
        tearDown();
    }
}
```


Fixture e SetUp

>> TestStudente.java + stampe
>> java.textui.TestRunner

- Esempio: TestStudenteModificato
  - ⇨aggiungendo istruzioni di stampa in setUp() e nei metodi di test si ottiene

```
D:\codice\>java junit.textui.TestRunner
               it.unibas.mediapesata.test.TestStudente
.Sto eseguendo setUp()
Sto eseguendo testMediaPesata1()
.Sto eseguendo setUp()
Sto eseguendo testMediaPesata2()
.Sto eseguendo setUp()
Sto eseguendo testMediaPesata3()

Time: 0,06

OK (3 tests)
```

TestRunner e TestSuite

- Il concetto di testRunner
  - ⇨un testRunner è una classe java capace di eseguire i test di una classe di test
- junit.textui.TestRunner
  - ⇨esecutore testuale
  - ⇨uso: java junit.textui.TestRunner <classeDiTest>
- Esempio

  java junit.textui.TestRunner segmenti.test.TestPunto

TestRunner e TestSuite

- junit.swingui.TestRunner
  - ⇨esecutore grafico con barra di progresso verde/rossa
  - ⇨possibilità di analizzare l'albero dei test
  - ⇨possibilità di ricaricare la classe di test dopo avere effettuato modifiche
  - ⇨uso: java junit.swingui.TestRunner <classeDiTest>
- Esempio

  java junit.swingui.TestRunner segmenti.test.TestPunto

TestRunner e TestSuite

>> TestTutto.java
>> java.textui.TestRunner

- Il concetto di testSuite()
  - ⇨i testRunner accettano come argomento il nome di una singola classe
  - ⇨ma spesso è necessario eseguire tutti i test dell'applicazione: molte classi
  - ⇨in questo caso è necessario creare una "suite di test", ovvero una classe che raccoglie i test di varie classi
  - ⇨eseguendo la suite si eseguono tutti i test

TestRunner e TestSuite

```
package it.unibas.mediapesata.test;

import junit.framework.*;

public class TestTutto extends TestCase {

    public static TestSuite suite() {
        TestSuite suite = new TestSuite();
        suite.addTestSuite(TestEsame.class);
        suite.addTestSuite(TestStudente.class);
        return suite;
    }
}
```

per creare una suite di test
è possibile creare una ulteriore
classe di test

che definisce il metodo suite
e restituisce un riferimento
ad un oggetto di tipo TestSuite

alla suite di test vengono aggiunti i test delle classi di test;
NOTA la sintassi: TestXXX.class (>>)

Le Regole di JUnit

- Riassumendo: i concetti di JUnit
  - classe di test: sottoclasse di TestCase
  - metoto di test: public void testXXX()
  - asserzione: Assert.assertTrue(s, *ass*);
  - fixture: proprietà della classe
  - set up: public void setUp()
  - suite: classe di test che definisce il metodo public TestSuite suite() per raggruppare i test
  - test runner: classe che esegue i test

Le Regole di JUnit

- Attenzione
  - si tratta di una descrizione “meccanica” di JUnit
  - in realtà il framework è basato su una serie di concetti avanzati della prog. a oggetti
  - polimorfismo (>>)
  - riflessione (>>>)
  - design pattern (>>)
  - che verranno approfonditi in seguito

Linee Guida

- Nota
  - ⇨i test di regressione sono molto importanti
  - ⇨oltre che essere uno strumento per la correttezza e per la manutenzione
  - ⇨sono un modo per “ragionare” sul codice
- Nel seguito
  - ⇨alcune linee guida per la scrittura corretta di test

Linee Guida

- Linee guida n. 1
  - ⇨quali classi verificare ?
  - ⇨i test di unità sono particolarmente adatti per i componenti di modello e persistenza
  - ⇨meno adatti ad interfaccia e controllo
- Per questi ultimi
  - ⇨è necessario scrivere test funzionali
  - ⇨che vedremo successivamente

ATTENZIONE
i test servono solo quando c'è una reale logica applicativa

Linee Guida

- Linee guida n. 2
  - ⇨quali metodi verificare ?
  - ⇨verificare solo l'interfaccia dei componenti
  - ⇨verificare solo i metodi che possono contenere errori, e cioè quelli che hanno una qualche logica applicativa
  - ⇨es: non è il caso di verificare i metodi set e i get

Linee Guida

- Linea guida n. 3
  - ⇨quanti test per ogni metodo ?
  - ⇨per ogni metodo da verificare è necessario preparare un piano di test che copre le varie condizioni di utilizzo (condizioni particolari)
  - ⇨varie condizioni ordinarie (es: media con vari esami, media con un esame)
  - ⇨e condizioni scorrette (es: media senza esami)

Linee Guida

- Come verificare le eccezioni ?
  - ⇨bisogna eseguire il metodo in condizioni scorrette, e verificare che sollevi le eccezioni attese
  - ⇨in questo caso il test non deve effettuare asserzioni, ma limitarsi a catturare l'eccezione con una regione catch
- E se l'eccezione non si verifica ?
  - ⇨in questo caso il metodo si comporta in modo scorretto e il test deve fallire

Linee Guida

- Il metodo Assert.fail()
  - ⇨provoca forzatamente il fallimento di un test
- Schema di test per una eccezione
  - ⇨viene chiamato il metodo in una regione try
  - ⇨se viene sollevata l'eccezione, viene catturata con il catch e il metodo termina con successo
  - ⇨se NON viene generata l'eccezione, bisogna forzare il fallimento del test con Assert.fail()

```
package it.unibas.mediapesata.test;

import junit.framework.*;
import it.unibas.mediapesata.*;

public class TestStudente extends TestCase {

   ...

   public void testMediaPesata3() {
      Studente studente = new Studente():
      try {
         this.studente.getMediaPesata();
         Assert.fail();
      } catch (IllegalArgumentException e) {  }
   }
}
```

se l'eccezione non si verifica il test deve fallire (il metodo non si comporta in modo corretto)

questo è uno dei rari casi in cui è consentito lasciare vuoto il corpo del blocco catch

Linee Guida

ATTENZIONE
i test devono essere semplici

- Linea guida n. 4
  - ⇨che stile adottare per i metodi di test ?
  - ⇨i test sono codice aggiuntivo, potenzialmente soggetto a sua volta ad errori
  - ⇨è opportuno che i metodi di test siano semplici e brevi per aiutare ad identificare le cause di errore
  - ⇨e che contengano poche asserzioni ciascuno
  - ⇨idealmente: una asserzione per metodo

Linee Guida

- Linea guida n. 5
  - ⇨cosa fare se un test fallisce ?
  - ⇨è stato individuato un baco catturato dai test
  - ⇨il test fornisce una immediata indicazione del componente e del metodo errati
  - ⇨una volta individuato l'errore è necessario correggerlo
  - ⇨ovvero effettuare il "debugging" del componente con i metodi ordinari

ATTENZIONE
tutti i bachi devono essere catturati da un test prima di essere corretti

Linee Guida

- Linea guida n. 6
  - ⇨cosa fare in caso si scopra un errore non catturato dai test ?
  - ⇨è possibile che, nonostante i test di unità abbiano successo, attraverso i test funzionali venga riscontrato un errore nel codice
  - ⇨in questo caso, se possibile, prima di correggere l'errore, è opportuno introdurre un nuovo test che "catturi" l'errore (test che fallisce a causa dell'errore)

Esempi di Test

- Nel seguito
  - ⇨analizziamo alcune classi di test per i progetti visti fino a questo punto
- Idea
  - ⇨discutere le caratteristiche principali del processo di scrittura dei test
  - ⇨fornire degli esempi di riferimento

Esempi di Test

- Gli esempi iniziali
  - ⇨hanno logica applicativa molto semplice
  - ⇨ma sono stati comunque sviluppati utilizzando test di regressione
  - ⇨principalmente per ragioni didattiche
  - ⇨in effetti, la logica applicativa non è, nella maggior parte dei casi, sufficientemente complessa da giustificare test di regressione

Esempi di Test

- Esempio
  - ⇨Calcolatrice, TestCalcolatrice
  - ⇨Circonferenza, TestCirconferenza
  - ⇨Segmento, Punto, TestSegmento, TestPunto
- Ci sono però alcuni aspetti interessanti
  - ⇨che discutiamo nel seguito

>> test sugli esempi iniziali

Esempi di Test

>> TestCalcolatrice.java

- Confronto tra valori reali
  - ⇨nelle asserzioni sui valori reali bisogna fare attenzione alle approssimazioni introdotte dalla virgola mobile
  - ⇨una soluzione è sviluppare una funzione che sia in grado di confrontare valori reali con una certa approssimazione (es: FORTRAN)
  - ⇨ public boolean uguali(double a, double b);
  - ⇨in alternativa, è possibile usare il metodo assertEquals() che gestisce nella maggior parte dei casi correttamente le approssimazioni

Esempi di Test

>> TestCirconferenza.java

- Test di condizioni eccezionali
  - ⇨i test sono scritti sempre in modo da verificare sia condizioni normali di utilizzo
  - ⇨sia condizioni eccezionali
  - ⇨in altri termini, il codice viene sollecitato anche in condizioni in cui solleva eccezioni
  - ⇨questo è indispensabile in ogni caso

Esempi di Test

>> TestSegmento.java
>> TestPunto.java

- Metodi fabbricatori nei test
  - ⇨tipicamente, gli oggetti della fixture vengono costruiti nel metodo di setUp
  - ⇨questo va bene nel caso in cui siano sufficienti pochi oggetti per tutti i test
  - ⇨in alcuni casi, viceversa, servono molti oggetti diversi su cui effettuare test
  - ⇨è possibile aggiungere metodi di servizio alla classe di test che fabbricano questi oggetti sulla base di argomenti specificati di volta in volta
  - ⇨questi metodi vengono detti “fabbricatori”

Esempi di Test

- Test di eventi casuali
  - ⇨nel caso di giochi in cui avvengono scelte casuali, il test si complica
  - ⇨è impossibile prevedere nel test il comportamento dell'applicazione
- Alcuni strumenti
  - ⇨utilizzare un approccio "statistico"
  - ⇨ragionare per unità

Esempi di Test

- Un esempio: la partita a indovinaIlNumero
  - ⇨due problemi principali
  - ⇨effettuare test sul numero generato (1-100)
  - ⇨effettuare test sullo svolgimento della partita
- Nel primo caso: approccio statistico
  - ⇨vengono fatti generare 200 numeri e si verifica che sia sempre corretto
  - ⇨sufficientemente convincente che il codice si comporta sempre correttamente

Esempi di Test

- Nel secondo caso
  - ⇨il problema del gioco è che simulare in modo realistico una partita intera è un po' intricato
  - ⇨perchè le risposte dell'utente dipendono dai suggerimenti forniti – problema di ricerca
  - ⇨richiederebbe un metodo di test non banale
  - ⇨in questo caso la logica applicativa del test diventerebbe troppo complessa (richiederebbe test del test !) – da evitare

Esempi di Test

>> TestPartita.java
>> TestRecord.java

- Un possibile approccio: ragionare per unità
  - ⇨per convincersi che la partita si svolge correttamente, bisogna per cominciare verificare che ogni singolo passo si svolge correttamente
  - ⇨vengono verificati tutti i possibili casi di tentativo
  - ⇨per convincersi che il codice risponde bene a ciascuno di questi casi
  - ⇨ragionevolmente questo dovrebbe garantire il corretto funzionamento della partita

Esempi di Test

>> TestMano.java
>> TestPartita.java

- Esempio: la morra cinese
  - ⇨in questo caso, viceversa, è molto più semplice giocare una partita in un test
  - ⇨è sufficiente continuare a scegliere oggetti a caso finchè la partita non si interrompe
  - ⇨in questo caso devo comunque ragionare per unità
  - ⇨ma posso anche scrivere metodi di test che simulano l'intera partita

Installazione e Classpath

- L'installazione di JUnit
  - ⇨tipica procedura per le applicazioni scritte in Java
- Operazione n. 1
  - ⇨scaricare il pacchetto dal sito www.junit.org
  - ⇨distribuito in formato zip: junit3.8.1.zip
  - ⇨con licenza CPL ("Common Public License"), una licenza open source
  - ⇨da non confondersi con la più celebre GPL

Installazione e Classpath

- Operazione n. 2
  - ⇨decomprimere il file .zip in una cartella; es: c:\Programmi
  - ⇨la decompressione crea la cartella junit3.8.1 che chiameremo %JUNIT_HOME%
- Contenuto di %JUNIT_HOME%
  - ⇨il jar di junit: junit.jar
  - ⇨documentazione, sorgenti, javadoc, licenza, altro materiale

Installazione e Classpath

- A questo punto
  - ⇨in teoria il framework sarebbe già utilizzabile
- Ma...
  - ⇨riconsideriamo l'utilizzo dei test runner
  - ⇨supponiamo di dover eseguire i test della classe segmenti.test.TestPunto sulla classe segmenti.Punto
  - ⇨es: java junit.swingui.TestRunner segmenti.test.TestPunto

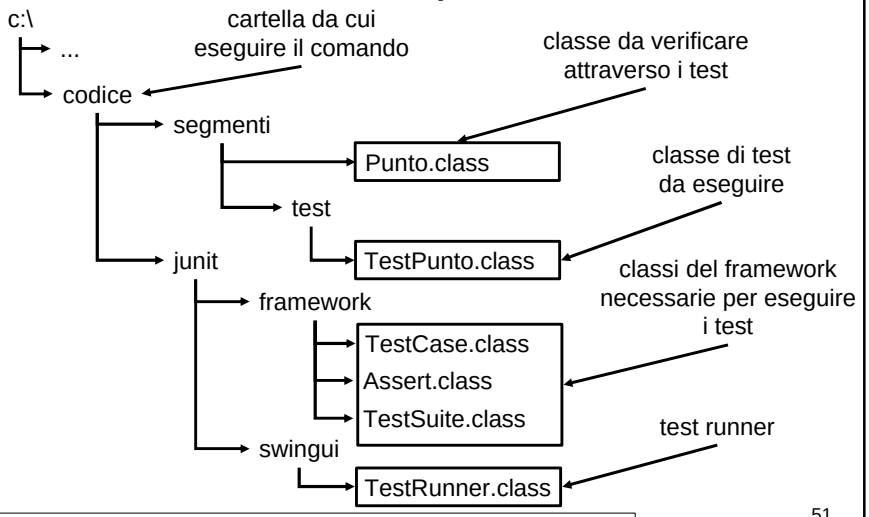
Installazione e Classpath

- Esecuzione di
  - ⇨java junit.swingui.TestRunner segmenti.test.TestPunto
- Il processo
  - ⇨la macchina virtuale chiede al classloader il caricamento della classe junit.swingui.TestRunner
  - ⇨a questo punto il TestRunner chiede al classloader il caricamento della classe di test

Installazione e Classpath

- Questo vuol dire che
  - ⇨sia il TestRunner.class che TestPunto.class devono essere raggiungibili tramite il classpath
- Una soluzione
  - ⇨utilizzare il classpath standard (.)
  - ⇨ed organizzare le cartelle in modo che le classi siano effettivamente raggiungibili
  - ⇨scompattando junit.jar (jar xvf junit.jar)

Installazione e Classpath



Installazione e Classpath

- Lo svantaggio di questa soluzione
  - ⇨dovrei scompattare junit.jar in tutte le cartelle in cui sono contenuti package da verificare
  - ⇨e questo stesso problema si porrebbe per qualsiasi altro framework analogo
- Sarebbe molto meglio
  - ⇨tenere junit sotto forma di jar in %JUNIT_HOME%
  - ⇨evitando di doverlo copiare dappertutto

Installazione e Classpath

- Due possibili soluzioni
  - ⇨cambiare permanentemente il classpath del sistema
  - ⇨specificare temporaneamente il classpath da utilizzare per la compilazione e l'esecuzione dei test (>>)
- Esempio: in c:\codice
  - ⇨javac –classpath .;c:\Programmi\junit.jar junit.swingui.TestRunner segmenti.test.TestPunto

Riassumendo

- Introduzione
- Le Regole di JUnit
  - ⇨Fixture e SetUp
  - ⇨TestRunner e TestSuite
- Linee Guida
- Esempi di Test
- Installazione e Classpath

Termini della Licenza

- This work is licensed under the Creative Commons Attribution-ShareAlike License. To view a copy of this license, visit http://creativecommons.org/licenses/by-sa/1.0/ or send a letter to Creative Commons, 559 Nathan Abbott Way, Stanford, California 94305, USA.

- Questo lavoro viene concesso in uso secondo i termini della licenza “Attribution-ShareAlike” di Creative Commons. Per ottenere una copia della licenza, è possibile visitare http://creativecommons.org/licenses/by-sa/1.0/ oppure inviare una lettera all’indirizzo Creative Commons, 559 Nathan Abbott Way, Stanford, California 94305, USA.