

Verifica di sistemi sw orientati agli oggetti

Testing di sistemi OO: livelli

I livelli di testing cambiano nel caso di programmazione a oggetti:

- *Testing di metodo* (method testing): testing di una singola operazione di una classe
- *Testing di classe* (class testing, unit testing): testing di una classe nella sua globalità
- *Testing di integrazione* (integration testing, inter-class testing): testing delle interazioni tra più classi
- *Testing di regressione* (regression testing): testing eseguito quando una o più classi sono state modificate

Influenza delle caratteristiche OO sul testing

Incapsulamento: semplifica il testing di unità ma complica quello di integrazione perché l'interazione con altre classi può aggiungere nuove sequenze di invocazione di ciascun metodo (cioè nuovi sottografi al diagramma degli stati della classe)

Polimorfismo: richiede di selezionare un test set tale che tutti i tipi che possono essere associati dinamicamente a una variabile su cui viene invocato un metodo siano effettivamente esercitati da qualche caso di test (copertura delle invocazioni polimorfe)

Influenza delle caratteristiche OO sul testing (cont.)

Ereditarietà: come noto, le sottoclassi possono

- Aggiungere attributi e metodi, rispettando però il vincolo semantico
invariantesottoclasse $\rightarrow$ invarianteclasse
- Ridefinire i metodi, soddisfacendo però i seguenti vincoli semantici:
postcondizionisottoclasse $\rightarrow$ postcondizioniclasse
precondizioniclasse $\rightarrow$ precondizionisottoclasse



per ogni sottoclasse creata/modificata è necessario identificare accuratamente che cosa va sottoposto a testing, cioè

- operazioni aggiunte
- operazioni ridefinite
- operazioni ereditate, ma influenzate dal nuovo contesto

Influenza di altre caratteristiche sul testing

Eccezioni: modificano il flusso di controllo senza che sia presente un esplicito costrutto condizionale → nuove forme di copertura:

- Selezionare un test set tale che ogni possibile eccezione sia sollevata almeno una volta
- Selezionare un test set tale che ogni possibile percorso di esecuzione associato a eccezioni sia esercitato almeno una volta

Concorrenza: diverse sequenze di sincronizzazione possono dare risultati diversi → ogni caso di test deve specificare la sequenza di sincronizzazione da seguire, che deve poi essere forzata tramite

- modifiche del codice sorgente (es. delays)
- dispatcher programmabile di processi/thread

Testing di sistemi procedurali e di sistemi OO: differenze

Nella programmazione OO:

- Il test di metodo è meno difficile del test di unità dei sistemi procedurali perché i metodi sono piccoli
- Il test di integrazione è molto più gravoso perché, nei sistemi OO, la complessità è sospinta verso le interfacce fra componenti
- Per la stessa ragione, le misure di copertura del codice e gli strumenti che le supportano sono di minore valore
- Verificare la corrispondenza fra oggetti e metodi, da una parte, e i requisiti del documento dei requisiti, dall'altra, è gravoso
- Esistono pochi strumenti in grado di assistere nella generazione di casi di test
- La maggior parte delle metriche del codice (ad es. il numero ciclomatico) sono inutili perché definite per il codice procedurale

Testing di sistemi procedurali e di sistemi OO: differenze (cont.)

Sistemi procedurali	Sistemi OO
La componente base è la procedura e un test è fondato su input/output	La componente base è la classe e un test dipende dallo stato dell'oggetto istanza di tale classe (= configurazione dei valori degli attributi in un punto dell'esecuzione) → è possibile forzare l'oggetto nello stato desiderato prima di iniziare il test • mediante funzionalità offerte dal linguaggio (es. friend), oppure • modificando il codice sorgente, oppure • si utilizzano le specifiche per determinare le sequenze di operazioni (pubbliche) che portano la classe in un certo stato
Una chiamata di procedura è risolta staticamente (a parte i puntatori a funzione)	Il codice effettivamente eseguito è noto solo durante l'esecuzione (polimorfismo)

Raccomandazioni di Fowler

- Non scrivere codice finché non si ha ben chiaro in mente come verificarlo
- Appena scritto un pezzo di codice, scrivere anche i test relativi
- Non ritenere conclusa la stesura del codice fino a quando tutti i casi di test non sono stati verificati
- Conservare per sempre il codice di verifica ed eseguirlo, ogni volta che è necessario, attraverso una semplice riga di comando o la pressione di un pulsante sull'interfaccia
- Scrivere il codice di verifica in modo che mostri l'eventuale lista dei malfunzionamenti rilevati, già interpretati
- Eseguire test sia per le singole unità (scritti dagli sviluppatori e organizzati in package per verificare le classi), sia per le funzionalità (scritti da un gruppo separato dedicato al testing che guarda al sistema come a una “scatola nera”)

Testing di unità di una classe

Difficoltà

Non si possono sfruttare le tecniche valide per i moduli procedurali perché

- la creazione di oggetti istanze della classe considerata può essere compiuta esternamente a tale classe
- il riuso della classe in altri contesti può portare a sequenze diverse di creazione degli oggetti e invocazione dei metodi

Soluzione

Testing basato sullo stato (state based testing): selezionare un test set tale che ogni transizione (corrispondente all'invocazione di un metodo) presente nel diagramma degli stati della classe sotto test venga esercitata almeno una volta; un test è superato se gli stati prossimi raggiunti sono quelli rappresentati nel diagramma degli stati

Testing basato sugli stati

Difficoltà

Il n° degli stati aumenta in modo combinatorio col n° degli attributi della classe

Soluzione

Anziché utilizzare un singolo (enorme) diagramma degli stati, considerare un insieme di diagrammi degli stati parziali

Concetto	Definizione
Stato parziale	Insieme di sottostati
Sottostato	Valore di un particolare attributo a un punto dell'esecuzione
Valore specifico di un sottostato	Valore singolo (di un attributo) a cui corrisponde una manipolazione individuale
Valori generici di un sottostato	Rappresentazione cumulativa di un gruppo di valori (di un attributo) a cui corrisponde una uguale manipolazione

Testing basato sugli stati (cont.)

Se un attributo p di un oggetto $o1$ è un puntatore, ci sono almeno 2 valori da associare a sottostati distinti di $o1$:

$p == \emptyset$
 $p \neq \emptyset$

Se p è il riferimento a un oggetto $o2$, ciascuno stato di $o2$ può indurre un nuovo sottostato di $o1$

Testing basato sugli stati: un esempio

```
class List {
    Item *head;
    Item *tail;
public:
    void append(Item *it);
    Item* remove();
};

class Item {
    char *data;
    Item *next;
    Item *prev;
};
```

Sottostati di tail:

S1 : tail == 0

S2 : tail != 0 && tail->prev == 0

S3 : tail != 0 && tail->prev != 0

Testing basato sugli stati: un esempio (cont.)

Transizioni:

t0: $\rightarrow S1$: constructor	def	} Variabile tail
t1: $S1 \rightarrow S2$: append	def+use	
t2: $S2 \rightarrow S3$: append	def+use	
t3: $S2 \rightarrow S1$: remove	def+use	
t4: $S3 \rightarrow S3$: append	def+use	
t5: $S3 \rightarrow S2$: remove [tail->prev->prev == 0]	def+use	
t6: $S3 \rightarrow S3$: remove [tail->prev->prev != 0]	def+use	

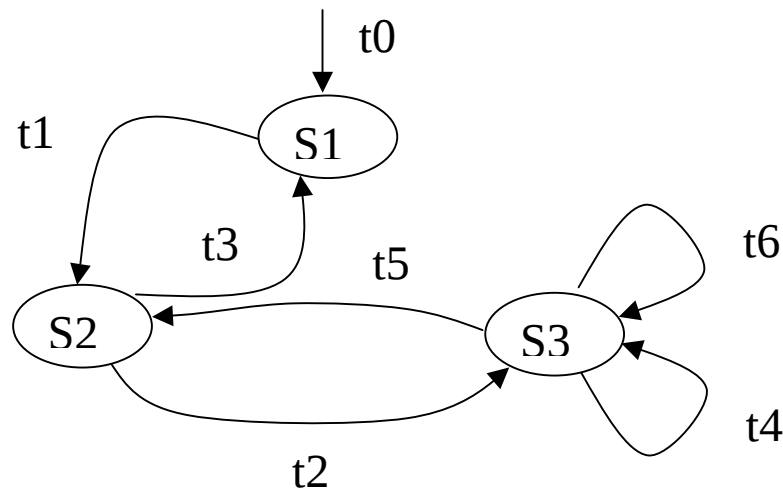
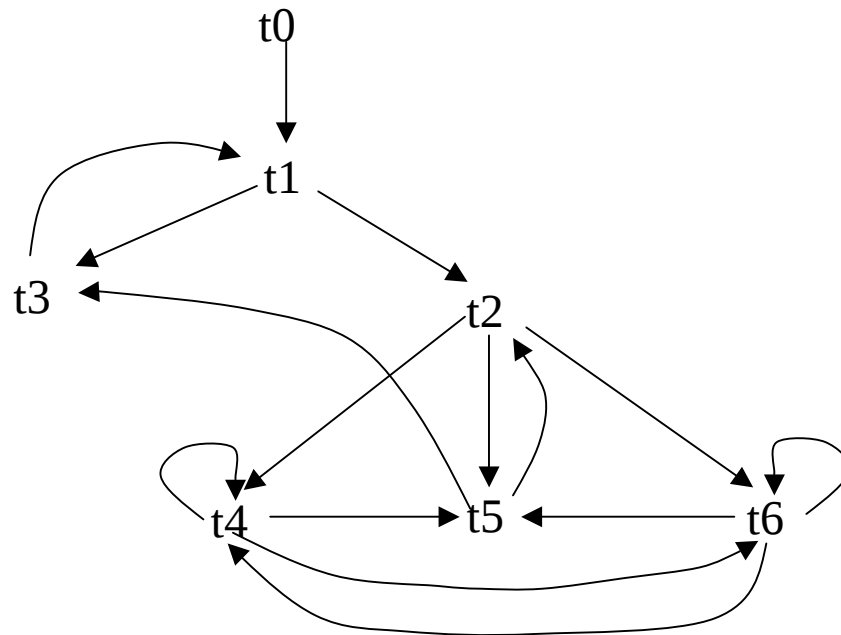


Diagramma degli
stati parziali della
classe List relativo
ai soli sottostati di
tail

Testing del flusso dei dati applicato al testing basato sugli stati

Il testing del flusso dei dati può essere applicato al testing basato sugli stati perché ciascuna transizione può definire e/o usare variabili; ciò richiede di derivare da ciascun diagramma degli stati un diagramma duale (sorta di CFG)

Esempio. Al diagramma degli stati di cui al lucido precedente corrisponde il seguente diagramma duale, da cui si ricavano i cammini def-uso della variabile `tail` elencati



(t0, t1)
(t1, t2); (t1, t3)
(t2, t4); (t2, t5); (t2, t6)
(t3, t1)
(t4, t4); (t4, t5); (t4, t6)
(t5, t2); (t5, t3)
(t6, t4); (t6, t5); (t6, t6)

Testing di integrazione

L'integrazione può sfruttare le dipendenze esplicitate dal diagramma delle classi, in particolare l'ordinamento parziale fra le stesse; in presenza di dipendenze cicliche, un passo di integrazione integra non già la singola classe ma un cluster contenente tutte le classi coinvolte

È possibile che alcuni malfunzionamenti si verifichino per legami dinamici (polimorfismo) che si attuano solo dopo l'integrazione → si utilizza l'Inter Class Control Flow Graph (ICCFG), in cui tutti i possibili binding sono rappresentati esplicitamente. Su questo si applicano tecniche di path-testing o data flow testing

Assertions (da Horstmann)

- Mechanism for warning programmers
- Can be turned off after testing
- Useful for warning programmers about precondition failure
- Syntax:

```
assert condition;    /* verifica che condition sia vera */
```

```
assert condition : explanation;
```

- Throws `AssertionError` if condition false and checking enabled
- `explanation` è una stringa (se è di tipo diverso, viene convertita in una stringa) che viene inserita nell'oggetto di tipo `AssertionError`
- Se l'asserzione non è verificata, l'esecuzione del programma termina e, nella maggior parte degli ambienti di esecuzione, viene visualizzato un msg d'errore (nome file, n° linea, `explanation`)

Assertions (da Horstmann, cont.)

```
public Message removeFirst()
{
    assert count > 0 : "violated precondition size() > 0";

    Message r = elements[head];

    . . .
}
```

- With Java SDK, during testing, run with

```
java -enableassertions MyProg
```

or (identically)

```
java -ea MyProg
```

Assertzioni in Java SDK

Sono state introdotte nella versione 1.4

Per compilare un programma che contiene asserzioni usando il compilatore 1.4 è necessario adottare l'opzione sotto indicata

```
javac -source 1.4 NomeProgramma.java
```

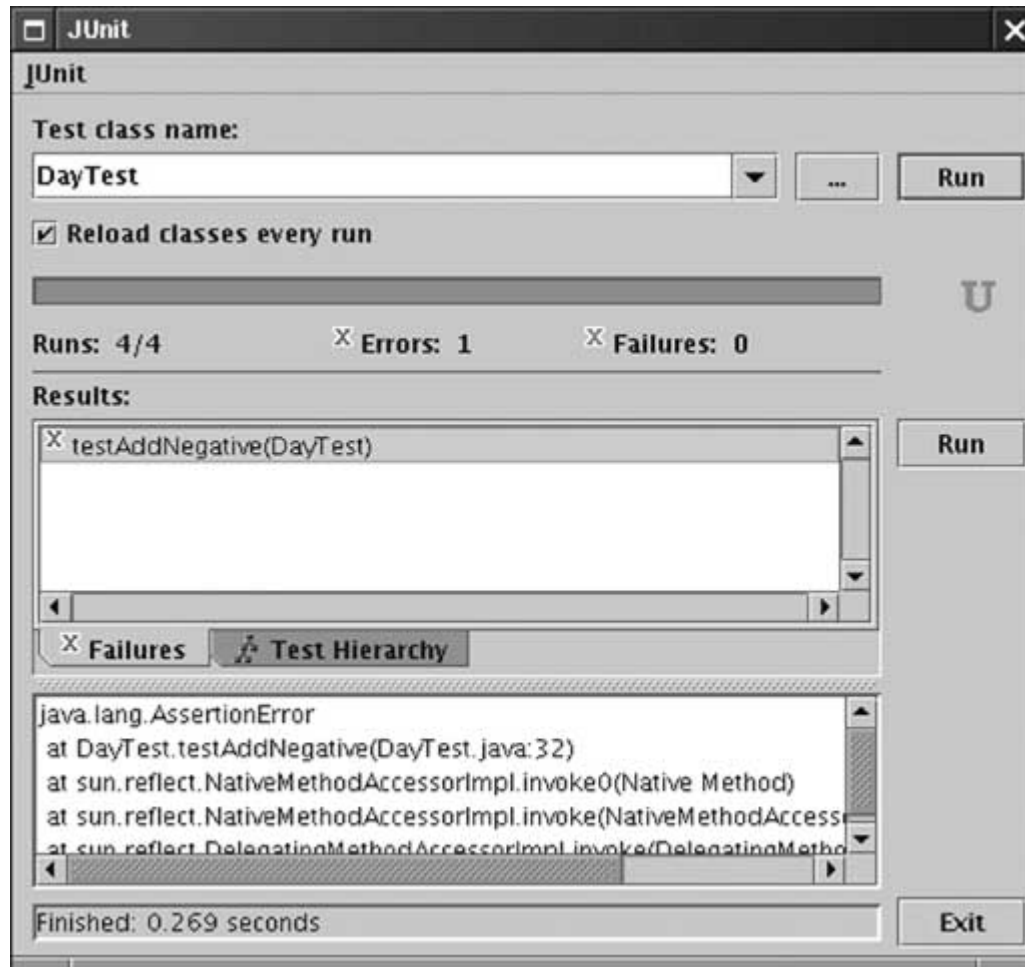
Unit Testing (da Horstmann)

- Unit test = test of a single class
- Design test cases during implementation
- Run tests after every implementation change
- When you find a bug, add a test case that catches it

L'esperienza dimostra che i programmatori sono molto meno riluttanti a migliorare la realizzazione di una classe quando dispongono di un insieme di prove che possono verificare le modifiche

JUnit (da Horstmann)

<http://junit.org>



JUnit (da Horstmann, cont.)

- Test class name = tested class name + Test (suggerimento)
- Test methods start with test

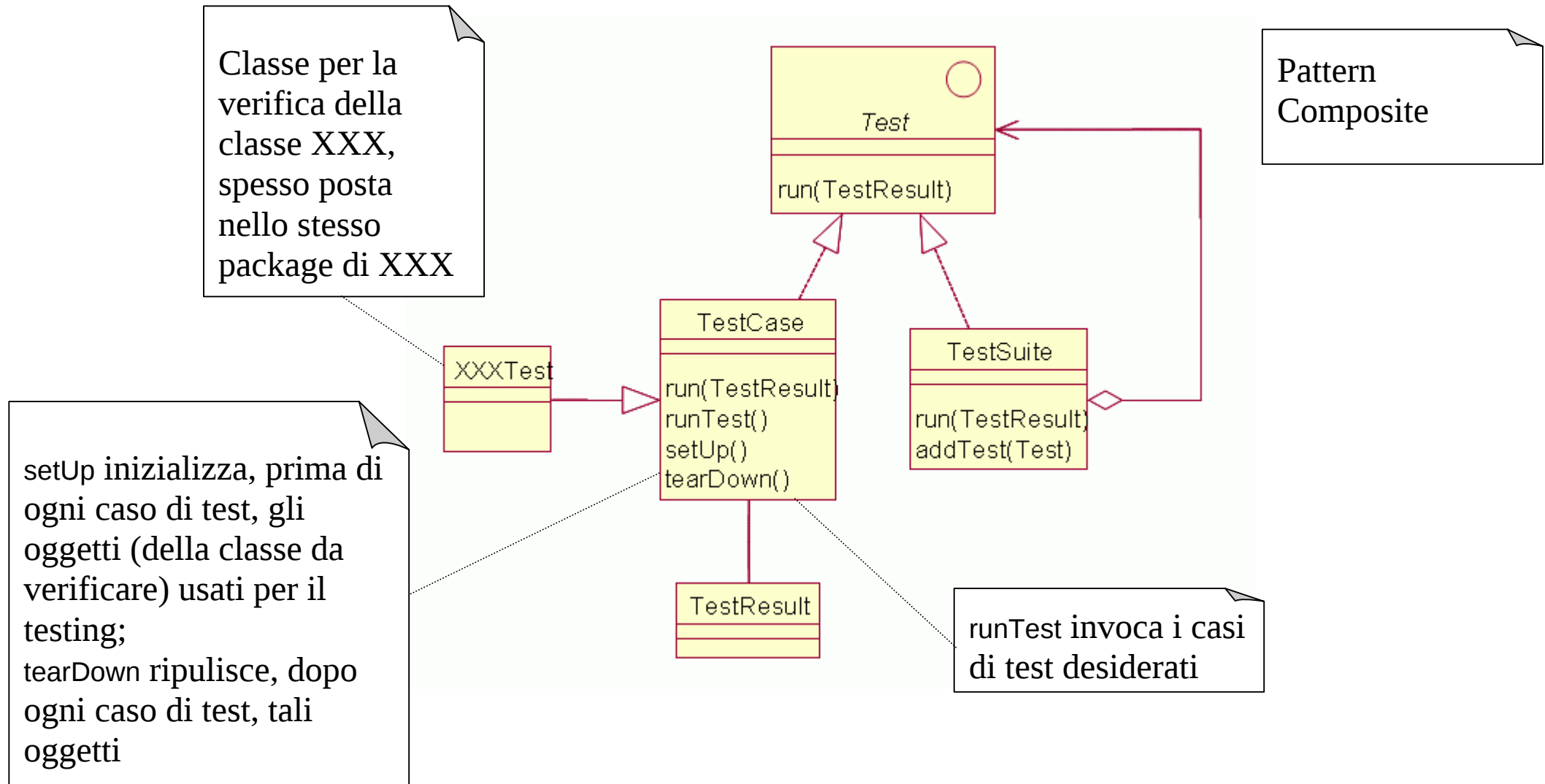
```
import junit.framework.*;

public class DayTest extends TestCase
{
    public void testAddDays() { ... }

    public void testDaysBetween() { ... }

    . . .
}
```

Diagramma delle classi di JUnit



Invocazione di casi di test con JUnit

```
public class XXXTest extends TestCase {
    public XXXTest (String testName) {
        super(testName);
    }
    public void testSomething ( ) {
        ...
    }
    public void testSomethingElse ( ) {
        ...
    }
    public void runTest ( ) {
        testSomething( );
        testSomethingElse( );
    }
}
```

Nota. Quella illustrata è una delle due varianti della modalità statica di invocazione dei casi di test con JUnit. Inoltre JUnit supporta anche una modalità dinamica.

JUnit (da Horstmann, cont.)

- Each test case ends with assertion
- Test framework catches assertion failures

Esempio: test del metodo AddDays della classe Day

```
public void testAddDays()  
{  
    Day d1 = new Day(1970, 1, 1);  
  
    int n = 1000;  
  
    Day d2 = d1.addDays(n);  
  
    assert d2.daysFrom(d1) == n;  
}
```

Uso di JUnit

Per compilare le classi di test occorre aggiungere il file `junit.jar` al percorso delle classi

```
javac -classpath .:junit.jar -source 1.4 NomeProgramma.java
```

Per usare l'ambiente grafico di JUnit è necessario eseguire il comando

```
java -classpath .:junit.jar -ea junit.swingui.TestRunner NomeProgramma
```