UNIT TESTING

Laboratorio di Algoritmi e Strutture Dati

INTRODUZIONE

- Test automatici:
  - "oggetti" software
  - obiettivo: verificare che una porzione di una applicazione sia corretta
- Oggetti testati UNIT
  - piccoli
  - autocontenuti
    - Es. classe, piccolo modulo, metodo
- UNIT TESTING:
  - test automatico di unit

UNIT TESTING

- Test automatici:
  - predispongono un input
  - invocano la unit
  - verificano che output (o eventuale side-effect) sia corretto
- Verifica mediante asserzioni
  - asserzione indica il risultato atteso
  - quando un'asserzione fallisce il test viene interrotto e viene restituito un messaggio d'errore

VANTAGGI dello UNIT TESTING

- Debugging:
  - testing indipendente di piccole unità di codice permette di isolare la porzione di codice in cui è presente il bug
  - soluzione del problema più semplice
- Qualità del codice:
  - disponibilità di una test suite che verifica automaticamente la correttezza del software crea la serenità necessaria perché il codice venga mantenuto e ripulito con la dovuta frequenza
  - unit piccole e ben focalizzate incoraggiano l'adozione di uno stile di programmazione migliore
- Documentazione:
  - unit test come fonte di documentazione per il codice
    - come si intende che le unit vengano utilizzate
    - relazione attesa tra input e output

Proprietà degli UNIT TEST

- i test devono essere focalizzati
  - uno unit test deve testare un singolo caso d'uso di una singola unit
  - un test che prende in considerazione più unit o più casi d'uso è meno utile
    - eventuali problemi sono più difficilmente ricollegabili a una porzione ristretta di codice

Proprietà degli UNIT TEST

- Esempi di test correttamente focalizzati:
  - test di un metodo di sort su un array vuoto
  - test di un metodo di sort su un array dato
  - test di un metodo di sort su un array con tutti elementi uguali
- Esempi di test non correttamente focalizzati:
  - test di un metodo di sort
  - test del funzionamento di più metodi contemporaneamente
- In genere un test poco focalizzato si riconosce perché contiene più asserzioni
  - Prima di scrivere un test di unità contenente più asserzioni è bene verificare l'opportunità di procedere in questa direzione

Proprietà degli UNIT TEST

- i test devono essere indipendenti
  - l'ordine di esecuzione dei test non deve influire sul loro risultato
  - JUnit impone questa caratteristica ricaricando l'intera classe in memoria prima di eseguire ciascun metodo di test e eseguendo i test in ordine casuale
- test **non** indipendenti:
  - un baco potrebbe essere nascosto dall'esecuzione di un test precedente
  - il debugging dei problemi diventa più complicato perché è necessario prendere in considerazione tutto il contesto

Proprietà degli UNIT TEST

- i test devono essere automatici
  - non devono richiedere l'intervento umano
  - un test che passa non deve generare nessun output degno di attenzione
  - un test fallito deve permettere di risalire velocemente alla porzione di codice da esaminare
  - Non scrivere nulla su console durante un test!

OSSERVAZIONE

- scrivere i test di unità è diverso dallo scrivere un programma di prova per il proprio software
- Unit test pensati per essere eseguiti automaticamente e non richiedere attenzione da parte dell'utente

UNIT TESTIG IN JAVA

- JUnit
  - libreria più usata per unit test in ambito Java
  - La versione più recente di JUnit è JUnit 5. Per semplicità, nell'esecuzione di questo laboratorio, suggeriamo l'uso della versione JUnit 4 (cui faremo riferimento qui)
  - test in JUnit = metodo marcato con l'annotazione @Test
  - Tali metodi possono usare un certo numero di funzioni messe a disposizione dalla libreria per verificare la correttezza del programma
  - JUnit provvederà ad eseguire i test in ordine casuale ricaricando la classe prima di ogni singolo test

JUnit: esempio

```
import org.junit.Test;
import static org.junit.Assert.assertArrayEquals;

public class TestSorting {
  @Test
  public void testSortEmptyArray() {
    MySortingAlgorithm sorter = new MySortingAlgorithm();
    int[] a = {};
    assertArrayEquals( new int[]{}, sorter.sort(a) );
  }

  @Test
  public void testSortNullArray() {
    MySortingAlgorithm sorter = new MySortingAlgorithm();
    int[] a = null;
    assertArrayEquals( null, sorter.sort(a) );
  }
}
```

JUnit compilazione ed esecuzione

- Al fine di poter compilare ed eseguire le classi di test è necessario aggiungere le librerie JUnit al classpath di Java.
- Potete:
  - trovare le librerie JUnit (junit e hamcrest-core) nella cartella Resources/Java/JUnit del repository Git
  - scaricarle dal sito di JUnit 4 (https://junit.org/junit4/)
- Esempio (Unix):

```
setenv CLASSPATH /usr/NFS/Linux/labalgoritmi/lib/junit-4.12.jar:/
        usr/NFS/Linux/labalgoritmi/lib/hamcrest-core-1.3.jar:.
```

UNIT TESTIG IN C

- purtroppo non esiste una libreria “standard”
- per questa ragione:
  - è ammesso utilizzare un programma ad-hoc per effettuare il test, a patto che si scrivano le funzioni di test prestando attenzione a quanto detto
  - è ammesso utilizzare librerie, ma sarà responsabilità dello studente/gruppo installarle e utilizzarle correttamente
- Semplice e molto usata:
  - Unity (http://www.throwtheswitch.org/unity/)
  - potete trovare Unity nella cartella Resources/C/Unity del repository Git

Esempio unit testing in C

```c
#include <assert.h>
#include <stdlib.h>
#include "my_sorter.h"

void test_sort_empty_array() {
  int a[] = {};
  assert( a == my_sorter(a, 0) );
}

void test_sort_null_array() {
  int* a = NULL;
  assert( NULL == my_sorter(a, 0) );
}

...

int main(int argc, char** argv) {
  test_sort_empty_array();
  test_sort_null_array();
  ...
}
```

Esempio unit testing in C con Unity

```c
#include "unity.h"

void test_massimo(void){
  TEST_ASSERT_EQUAL_INT(23, massimo(23,4));
  TEST_ASSERT_EQUAL_INT(44, massimo(44,44));
}

int main(void){
  UNITY_BEGIN();
  RUN_TEST(test_massimo);
  ...
  return UNITY_END();
}
```

```
gcc FileTest.c OtherFile.c unity/src/unity.c -o Test
```

```
./Test
```