Test di unità

- scompongono il programma da testare in parti (unità) differenti
- ogni unità (classe o metodo) viene testata separatamente

jUnit

- jUnit è un framework per il test di unita per il linguaggio Java
- consentediseguireunparadigmadiprogrammazionetest-driven

Unit test case

- porzione di codice che assicura che un altra parte di codice (in genere un metodo) funziona come aspettato
- un test formale ben scritto è caratterizzato da un input noto e un output atteso stabiliti prima dell’esecuzione del test
- per ogni requisito (funzionalità implementate) devono esserci almeno due test (uno positivo e l’altro negativo)
- jUnit consente di utilizzare le annotazioni per identificare i metodi di test
- le asserzioni servono a confrontare i risultati ottenuti con quelli attesi

Caratteristiche di jUnit

JUnit fornisce le seguenti feature:

- Fixture:
  - è possibile settare uno stato predefinito degli oggetti prima di eseguire un test
  - l'obiettivo è di assicurare che l'ambiente nel quale i test sono eseguiti è noto e fissato in modo che i test siano ripetibili
  - vengono utilizzate le annotazioni (@Before (setting), @After (pulizia))
- Test runner:
  - è utilizzato per eseguire i test (trasparente all'utente)

Annotazioni

- @Before: codice eseguito prima di ogni caso di test per creare la situazione iniziale corretta
- @After: codice eseguito dopo ogni caso di test. Serve per ripulire lo stato dell’ambiente
- @Test: codice dei casi di test veri e propri

Asserzioni

- assertEquals(expected, actual): controlla che il valore restituito sia uguale (equals) a quello atteso
- se ci interessa l’identità dobbiamo usare assertSame
- assertTrue(actual)/assertFalse(actual): utili per delle condizioni booleane

Come deve essere un buon test d’unità?

- completamente automatizzato
- ogni caso di test deve coprire una sola funzionalità
- l’insieme dei casi di test deve coprire tutte le funzionalità dell’unità

Esempio 1

- vogliamo creare un metodo che converte dollari in franchi svizzeri (CHF)
- requisito base: dobbiamo essere capaci di moltiplicare un numero per un altro
- esempio: se il cambio Dollari-CHF è 2:1, 5 Dollari vengono convertiti in 10 franchi
- visto che siamo test driven partiamo dai test e non dagli oggetti

Best practice

- non utilizzare il costruttore del test case per settare il test case
- non assumere di conoscere l’ordine nel quale i test case vengono eseguiti (anche all’interno del singolo jUnit)
- non scrivere test case che abbiano side effect
- scrivere i test che leggono dati da locazioni del file system utilizzando path relativi
- memorizzare i dati che sono necessari per i test assieme ai test stessi
- assicurarsi che i nomi dei test siano time-independent

Best practice

- scegliere i nomi dei test nel modo corretto:
- il nome del test deve iniziare con la parola Test (esempio TestClassUnderTest)
- il nome dei metodi nel test case devono descrivere cosa viene testato (esempio testLoggingEmptyMessage())
- utilizza i metodi assert e fail di jUnit nel modo corretto per mantenere il codice pulito
- ad esempio:
  assertEquals("Should be 3 credentials", 3, creds); è più leggibile di assertTrue(creds==3);
- commentare i test con la Javadoc
- mantenere i test piccoli e veloci

Come testare metodi privati?

- è possibile testare metodi privati tramite i metodi pubblici che li utilizzano
- alternativamente è possibile utilizzare la reflection

Che cosa testare?

- in genere tanto più una parte di codice è visibile dall’esterno, tanto più deve essere testata