

Programmazione ad Oggetti

Qualità del codice:
introduzione alle tecniche di testing

Sommario

- Software ed errori
- Testing
- Introduzione a JUnit



Software ed Errori

- I primi errori con i quali ci scontriamo di solito sono *errori di sintassi*
 - Ci vengono indicati dal compilatore
- Successivamente incorriamo in *errori logici*
 - Il compilatore non ci può aiutare
 - Sono noti anche come “*bug*” (*bachì*)
- Alcuni errori logici non si manifestano immediatamente
 - Il software è estremamente complesso
 - Anche il software commerciale raramente è privo di errori

Errori di compilazione

- Il compilatore ci dà indicazioni precise e molto utili a correggere l'errore
- Il messaggio di errore del compilatore
VA LETTO E CAPITO

LINEA DI CODICE IN CUI E' STATO RISCONTRATO L'ERRORE

ERRORE RISCONTRATO

```
Diadia.java:27:invalid method declaration;  
return type required  
private creaStanze() {  
    ^
```

1 error

Errori a tempo di esecuzione

- Anche in questo caso abbiamo informazioni molto precise (dalla macchina virtuale)

```
Exception in thread "main"  
java.lang.NullPointerException
```

```
    at Diadia.vaiNellaStanza (Gioco.java:176)  
    at Diadia.processaComando (Gioco.java:117)  
    at Diadia.gioca (Gioco.java:71)  
    at Diadia.main (Gioco.java:209)
```

Motivazioni del Testing

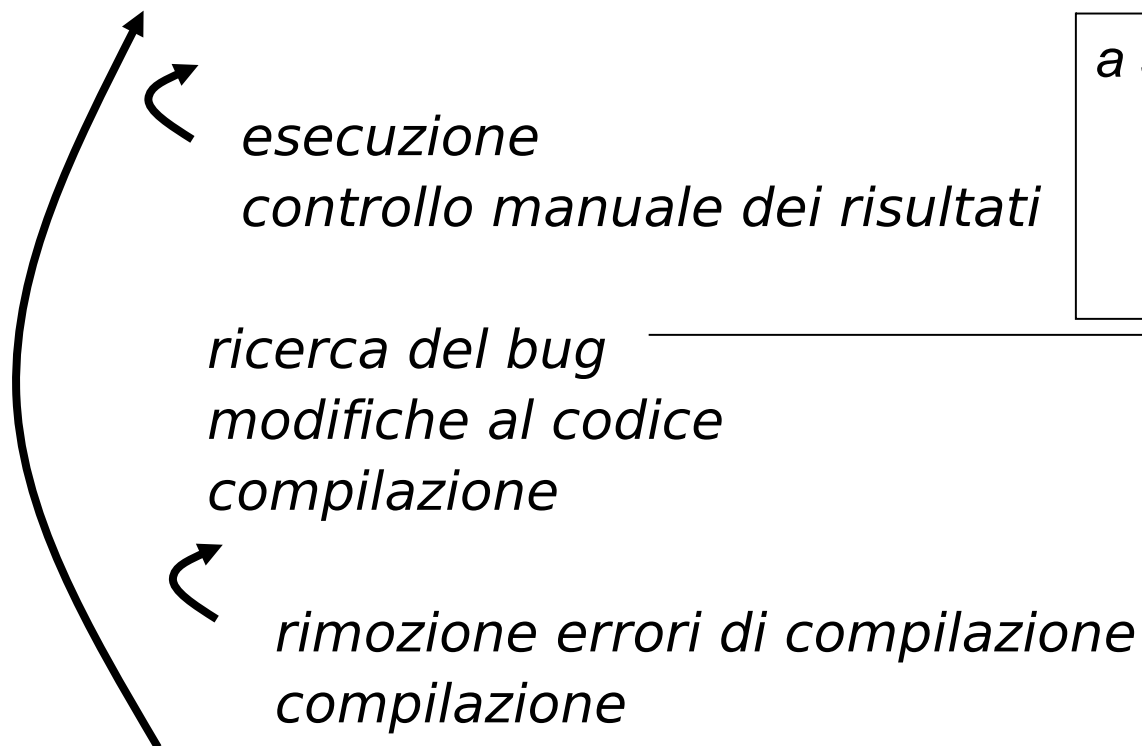
- I programmi sono descrizioni “statiche” a cui possono corrispondere molteplici esecuzioni “dinamiche”
- I compilatori moderni sono in grado di indicare esattamente posizione e motivo degli errori di compilazione
- Al contrario i compilatori non possono prevedere come evolverà l'esecuzione di un programma e non sono in grado di individuare gli errori dei programmatori (né possono sapere cosa intendevano fare)
- In sintesi:
 - il compilatore ci aiuta sugli aspetti statici (ad. es. analizzando i tipi)
 - il compilatore non dice nulla o quasi sugli aspetti dinamici

I Bug

- I bug sono errori nell'evoluzione dinamica di un programma su cui il compilatore non ha potuto prevedere e dire nulla
- Il debugging è completamente a carico del programmatore
- Il costo di debugging è ritenuto di gran lunga la componente principale nel costo dei moderni progetti software

Ciclo di Debugging

- Come si effettua il debugging di un programma che compila? Con estenuanti cicli:



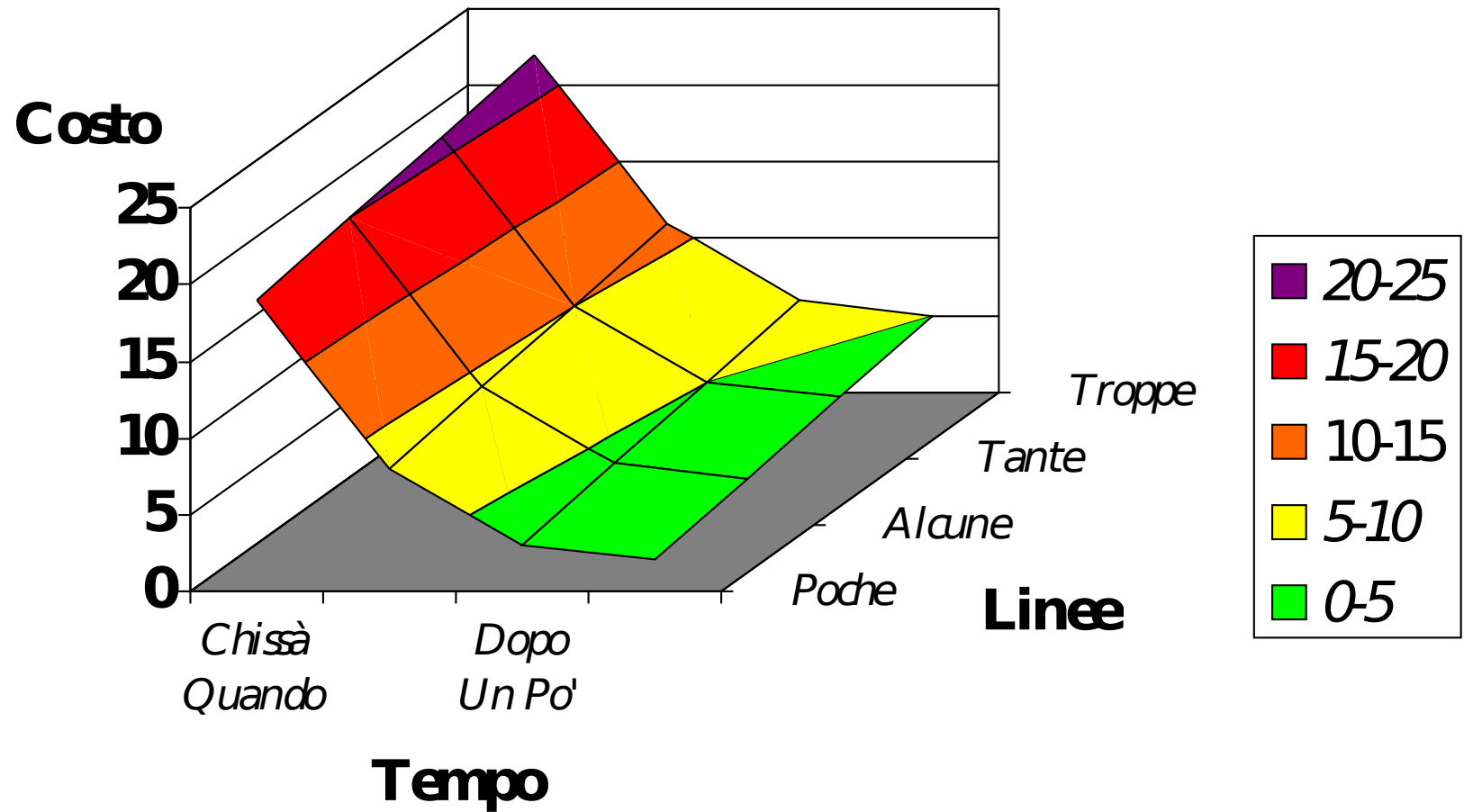
a sua volta può richiedere:

- *sessioni di tracing/logging*
- *sessioni con il debugger*

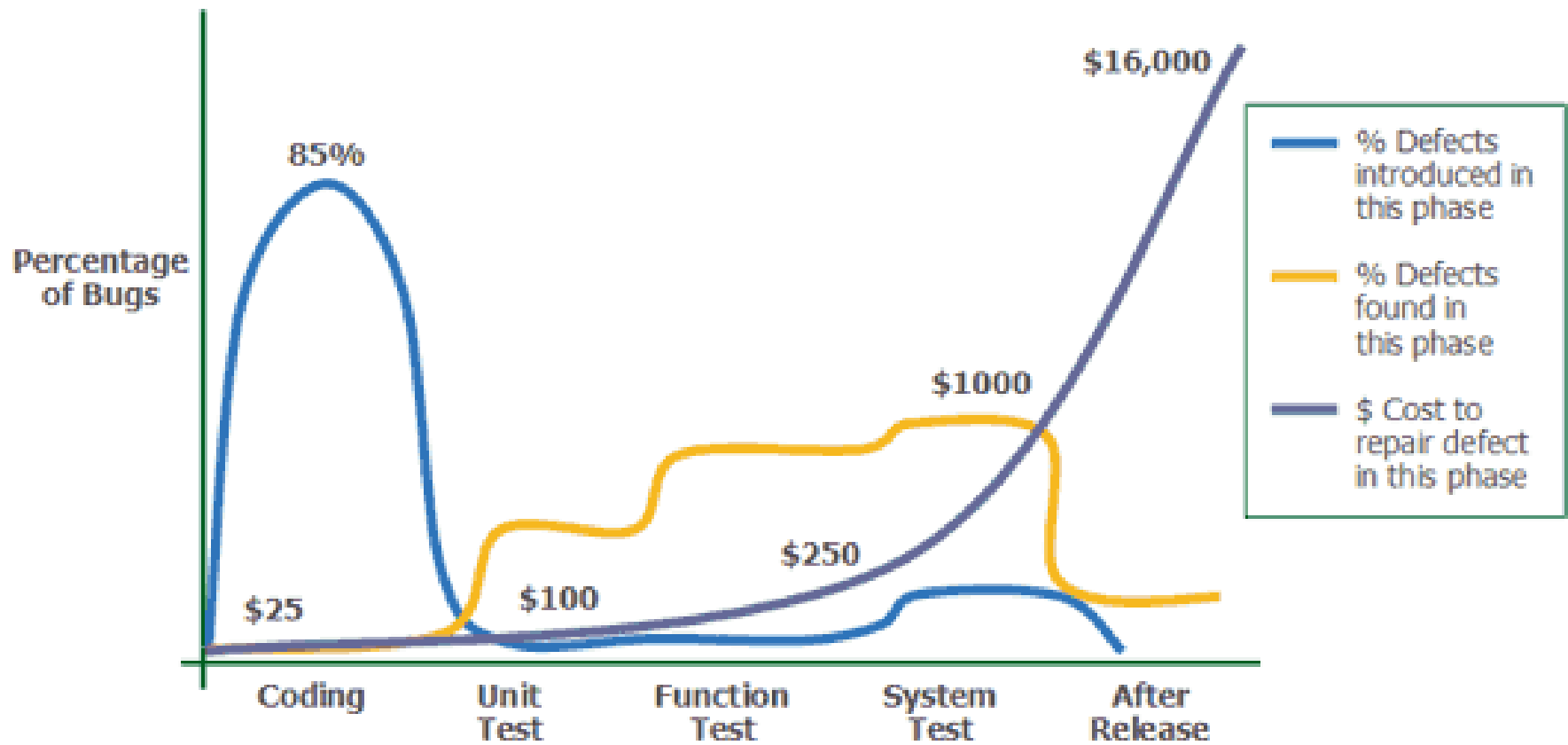
Costo del debugging

- E' ormai risaputo che i programmatori spendono la maggior parte del tempo per il debugging del codice
- E' anche noto che il costo della correzione di bug dipende da almeno due grandezze che ne determinano la *località*
 - le “dimensioni” del contesto
 - numero di linee di codice in cui il bug può annidarsi
 - il “tempo” che il bug impiega per manifestarsi
 - misura temporale di quanto dista la causa del bug (durante un'esecuzione del codice) ed il rilevamento dei suoi effetti

Costo di un Bug e “Località”



Costo di un bug



Obiettivi del testing

- Se ben progettati e mantenuti, i test aiutano a confinare i bug nella “zona verde” , ovvero con forti caratteristiche di località
 - i bug si manifestano immediatamente e palesemente
 - il difetto nel codice è confinato in un numero di linee non eccessivamente elevato e comunque ben identificabili
- Le esecuzioni che manifestano il bug non sono mai troppo lunghe e complesse

Conseguenze del Testing

- se il test ha successo si possiede una garanzia sul comportamento dinamico del codice
- se il test non ha successo il bug dovrebbe risultare facilmente localizzabile nella porzione sollecitata dal test
- se il test smette di funzionare in seguito ad una modifica del codice, con grande probabilità l'errore è dovuto alla modifica stessa e là va ricercato

Esercizio

- Supponiamo di voler testare il metodo **massimo** della classe **Sequenza** (quiz di preparazione)
 - Scriviamo in un documento di testo (.txt) diverse istanze dell'array di interi, per ogni sequenza scriviamo il massimo atteso
 - facciamo girare il programma su ciascuna sequenza e verifichiamo che il risultato sia quello atteso
- Osservazioni
 - Ovviamente è necessario scegliere con cura gli array di test
 - possiamo scrivere i test senza preoccuparci dell'algoritmo per il calcolo del massimo.
Conseguenza: possiamo scrivere i test prima di scrivere il programma!

Testo QUIZ

- Scrivere il codice del metodo `public int massimo()` che deve restituire il valore più grande presente nell'array `sequenza`.

```
public class Sequenza {  
    private int[] sequenza;  
  
    public Sequenza(int n){  
        sequenza = new int[n];  
    }  
  
    public int massimo(){  
        // scrivere il codice di questo metodo:  
        // deve restituire il valore piu' grande  
        // presente nell'array sequenza  
    }  
  
    public void setElemento(int indice, int valore) {  
        sequenza[indice] = valore;  
    }  
}
```

Codice di Test

- Nella pratica, accanto al *codice di produzione* si sviluppa sempre del *codice di test* il cui unico motivo di esistere è quello di verificare la correttezza a tempo di esecuzione del codice principale
- Il codice di test accompagna e supporta lo sviluppo del codice di produzione ma non fa parte del codice consegnato a fine progetto

Test Unitari Automatici

- Esistono diversi tipi di test
- Attenzione limitata ai
 - test-unitari (unit-test): si focalizzano su *frammenti* del sistema
 - test automatici: senza intervento umano
- Praticamente i test si codificano nel medesimo linguaggio di programmazione utilizzato per lo sviluppo (java)

Test Unitari - Unit Testing

- Test su *frammenti* di un sistema piuttosto che sull'intero sistema
- Concettualmente un test (unitario) si articola in questi passi
 - mettere un “frammento” del sistema in un stato noto
 - inviare una serie di messaggi noti
 - controllare che alla fine il sistema si trovi nello stato atteso

Automazione dei Test

- E' possibile eseguire test manuali
 - riportando i dettagli in un documento di testo e verificando che dopo ogni passo si arrivi allo stato atteso
- Ogni esecuzione di un test manuale richiede un considerevole sforzo sia per inserire l'input che per ispezionare visivamente i risultati
- L'automazione dei test è fondamentale

Una soluzione artigianale

- Una possibile soluzione
 - molto artigianale
 - ma automatica
- Scriviamo un programma in cui
 - inizializziamo un certo numero di oggetti con sequenze di test
 - invochiamo il metodo sotto test e verifichiamo che il risultato ritornato sia uguale a quello atteso

Esempio soluzione artigianale (1)

```
public static void main(String[] args){
    Sequenza positivi;
    Sequenza negativi;
    Sequenza negEpos;
    Sequenza negEzero;
    Sequenza inPrimaPos;
    Sequenza inUltimaPos;

    positivi = new Sequenza(5);
    positivi.setElemento(0,1);
    positivi.setElemento(1,5);
    positivi.setElemento(2,8); // MAX!
    positivi.setElemento(3,3);
    positivi.setElemento(4,4);

    negativi = new Sequenza(5);
    negativi.setElemento(0,-6);
    negativi.setElemento(1,-1); // MAX!
    negativi.setElemento(2,-8);
    negativi.setElemento(3,-13);
    negativi.setElemento(4,-10);
```

Esempio soluzione artigianale (2)

```
negEpos = new Sequenza(5);  
negEpos.setElemento(0,100);  
negEpos.setElemento(1,-5);  
negEpos.setElemento(2,-80);  
negEpos.setElemento(3,1000); // MAX!  
negEpos.setElemento(4,10);
```

```
negEzero = new Sequenza(5);  
negEzero.setElemento(0,-1);  
negEzero.setElemento(1,0); // MAX!  
negEzero.setElemento(2,-80);  
negEzero.setElemento(3,-10);  
negEzero.setElemento(4,-10);
```

```
inPrimaPos = new Sequenza(5);  
inPrimaPos.setElemento(0, 1000); // MAX!  
inPrimaPos.setElemento(1, 0);  
inPrimaPos.setElemento(2, 80);  
inPrimaPos.setElemento(3,-10);  
inPrimaPos.setElemento(4,-10);
```

```
inUltimaPos = new Sequenza(5);  
inUltimaPos.setElemento(0, 1);  
inUltimaPos.setElemento(1, 0);  
inUltimaPos.setElemento(2, 80);  
inUltimaPos.setElemento(3,-10);  
inUltimaPos.setElemento(4, 1000); // MAX!
```

Esempio soluzione artigianale (3)

```
boolean esito = true;
esito &= (positivi.massimo() == 8);
System.out.println(positivi.massimo() == 8);
esito &= (negativi.massimo() == -1);
System.out.println(negativi.massimo() == -1);
esito &= (negEpos.massimo() == 1000);
System.out.println(negEpos.massimo() == 1000);
esito &= (negEzero.massimo() == 0);
System.out.println(negEzero.massimo() == 0);
esito &= (inPrimaPos.massimo() == 1000);
System.out.println(inPrimaPos.massimo() == 1000);
esito &= (inUltimaPos.massimo() == 1000);
System.out.println(inUltimaPos.massimo() == 1000);

System.out.println(esito);
}
```

Una soluzione artigianale

- La soluzione presentata, benché artigianale è automatica
 - dopo ogni modifica al metodo sotto test
 - possiamo far rigirare il programma di test e verificare se ci sono cambiamenti per evitare regressioni

Automazione dei Test

- I test devono essere:
 - automatici (per mantenere rapido il ciclo di feedback)
 - devono essere eseguiti molte volte al giorno
 - efficienti
 - devono essere convenienti rispetto alle ispezioni manuali
 - isolati e che garantiscano la località degli errori
 - dal fallimento di un test alla rimozione del bug deve trascorre poco tempo grazie alla località errori che rilevano
 - ed inoltre:
 - separati dal codice applicativo
 - eseguibili e verificabili separatamente
 - raggruppabili a piacimento in “suite”

Automazione dei test: JUnit

- Esistono vari strumenti per assistere il programmatore nel testing
- Il più noto e utilizzato è JUnit
(<http://www.junit.org>)
 - un framework per la scrittura di classi test

JUnit: Test del metodo massimo

```
import static org.junit.Assert.*;  
import org.junit.Test;  
public class SequenzaTest {
```

import di classi ed
annotazioni JUnit

nome

Annotazione di metodo come test-case

```
@Test  
public void testMassimoPositivi() {  
    this.p = new Sequenza(5);  
    this.p.setElemento(0,1);  
    this.p.setElemento(1,5);  
    this.p.setElemento(2,8);  
    this.p.setElemento(3,3);  
    this.p.setElemento(4,4);  
    assertEquals(this.p.massimo(), 8);  
}
```

test-case

Asserzione

```
@Test  
public void testMassimoNegativi() {  
    ...  
    ...  
}
```

test-case

```
...  
}
```

JUnit: struttura classi di test

- Tutte le classi di test che scriveremo avranno questa struttura
- Ovviamente le classi di test vanno progettate sulla base delle peculiarità della classe testata
- Collochiamo la classe di test nello stesso package della classe che si sta testando
- Convenzione sui nomi basato sul suffisso:

Classe

Sequenza → **SequenzaTest**

Classe di Test

JUnit: struttura classi di test

- `import static org.junit.Assert.*;`
Serve per importare metodi (statici) e annotazioni del framework JUnit
- `@Test` è un'annotazione per marcare i metodi che si considerano di Test
- Non è (più) necessario ma è buona norma usare 'test' come prefisso del nome dei metodi di test

`@Test`

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

JUnit: Asserzioni

- Asserzione:
affermazione che può essere vera o falsa
- I risultati attesi sono documentati con delle *asserzioni* esplicite, non con delle stampe che comunque richiedono dispendiose ispezioni visuali dei risultati
- Se l'asserzione è
 - vera: il test è andato a buon fine
 - falsa: il test è fallito ed il codice testato non si comporta come atteso, quindi c'è un errore a tempo dinamico

JUnit: Asserzioni

- Se una asserzione non è vera il test-case fallisce
 - **assertNull()**: afferma che il suo argomento è nullo (fallisce se non lo è)
 - **assertEquals()**: afferma che il suo secondo argomento è **equals()** al primo argomento, ovvero al valore atteso
 - molte altre varianti
 - **assertNotNull()**
 - **assertTrue()**
 - **assertFalse()**
 - **assertSame()**
 - ...
- tutte sovraccariche ...

JUnit: asserzione `assertEquals()`

- `assertEquals(Object expected, Object actual)`

Va a buon fine se e solo se `expected.equals(actual)` restituisce `true`

`expected` è il valore atteso

`actual` è il valore effettivamente rilevato

- `assertEquals(String message, Object expected, Object actual)`

In questa variante si specifica un messaggio che il *runner* stampa in caso di fallimento dell'asserzione: molto utile per localizzare immediatamente l'asserzione che causa il fallimento di un test-case ed avere i primi messaggi diagnostici

Test unitario in pratica

`@Test`

```
public void testMassimoPositivi() {  
    this.p = new Sequenza(5);  
    this.p.setElemento(0,1);  
    this.p.setElemento(1,5);  
    this.p.setElemento(2,8);  
    this.p.setElemento(3,3);  
    this.p.setElemento(4,4);  
    assertEquals(this.p.massimo(), 8);  
}
```

- mettere un “frammento” del sistema in un stato noto
 - il frammento comprende un solo oggetto **Sequenza**
- inviare una serie di messaggi noti
- controllare tramite asserzioni che alla fine il sistema si trovi nello stato atteso

JUnit: compilare i Test

- Tutto sarà semplificato con Eclipse
- Nel classpath ci devono essere le librerie di JUnit (ad es. `junit-4.10.jar`).
- Supponiamo che queste siano nella directory `c:\java\lib\` :

```
javac -cp ".;c:\java\lib\junit-4.10.jar;c:\src" ComandoTest.java
```

Eseguire i test

- Per eseguire dei test è necessario usare una classe *runner* che trova ed esegue i test-case
- JUnit 4.x include come Runner
 - `org.junit.runner.JUnitCore`accetta come argomento una o più classi di test

```
$java -cp ".;c:\java\lib\junit-4.4.jar;c:\src\diadia"  
      org.junit.runner.JUnitCore diadia.ComandoTest
```

```
JUnit version 4.4
```

```
..... ←
```

```
Time: 0,066
```

```
OK (8 tests)
```

Un puntino per ogni test-case andato a buon fine

JUnit ed Eclipse

- L'uso di Junit è talmente diffuso, che è stato integrato negli IDE
- In Eclipse
 - creare una classe di test
 - eseguire una classe di test
 - barra verde: il test è andato a buon fine
 - barra rossa: il test è fallito

JUnit: Fixture

- Per facilitare la scrittura dei test-case, spesso è molto comodo creare degli oggetti/valori che possano essere utilizzati da tutti i test-case che lo desiderano
- Spesso la porzione di codice che si occupa di mettere l'unità da testare in uno stato noto può essere condiviso tra diversi test-case
- Le *fixture* sono oggetti/valori in uno stato iniziale noto ospitati in variabili d'istanza che le classi di test predispongono allo scopo

Fixture e JUnit

- Attraverso l'annotazione `@Before` è possibile indicare al runner quali metodi vanno eseguiti *prima di ogni* invocazione di test-case
- Tipicamente questi metodi inizializzano le fixture

Fixture

```
import static org.junit.Assert.*;
import org.junit.Before;
import org.junit.Test;
```

```
public class SequenzaTest {
    private Sequenza positivi;
    private Sequenza negativi;
```

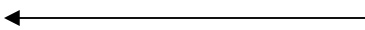
Fixture



@Before

```
public void setUp() {
    this.positivi = new Sequenza(5);
    this.positivi.setElemento(0,1);
    this.positivi.setElemento(1,5);
    this.positivi.setElemento(2,8);
    this.positivi.setElemento(3,3);
    this.positivi.setElemento(4,4);
```

*Metodo eseguito prima di ogni
invocazione di test-case*



```
    this.negativi = new Sequenza(5);
    this.negativi.setElemento(0,-6);
```

...

```
}
```

@Test

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

@Test

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

```
}
```

Fixture

```
...
public class SequenzaTest {
    ...
    @Test
    public void testMassimoPositivi() {
        assertEquals(this.positivi.massimo(), 8);
    }

    @Test
    public void testMassimoNegativi() {
        assertEquals(this.negativi.massimo(), -1);
    }

    ...
}
```


Testing Continuo

- Il testing deve essere una attività associata allo sviluppo
- Lo sviluppo dei test avviene progressivamente e continuativamente assieme allo sviluppo del codice principale
- Motivazioni principali
 - La rimozione precoce degli errori riduce i costi di sviluppo
 - Si costruisce contestualmente al codice principale un ambiente di test
 - I test possono essere riutilizzati durante la manutenzione del software ad esempio per evitare regressioni
 - Si accumulano *batterie di test* che sono importanti per lo sviluppo e la manutenzione del codice quanto il codice principale

Testing

- Chi scrive test si costringe nel ruolo del programmatore-utilizzatore e si focalizza sulla semplicità di utilizzo del proprio codice
- Per questo motivo il testing aiuta a cambiare la prospettiva di visione sul proprio codice, a concentrarsi sulle interfacce delle proprie classi e sulla distribuzione delle responsabilità
- Tipicamente il codice di qualità è più semplice da testare e viceversa
- Esistono metodologie di sviluppo che portano all'estremo questa affermazione: TDD.

Test Driven Development

- Promuove l'uso dei test non solo per le ragioni tradizionali ma anche come strumento di progettazione
 - i test guida lo sviluppo verso codice che sia semplice, facilmente testabile e di qualità
- Predica la scrittura dei test-case **prima** della scrittura del codice testato

Sviluppo guidato dai test

- Scrivendo il codice di test prima del codice stesso siamo costretti a:
 - chiarire quali sono i metodi visibili all'esterno perché il codice di test è trattato esattamente come qualsiasi altro codice “cliente” esterno alla classe
 - chiarire la semantica dei metodi
 - pensare ai possibili errori e chiarire il comportamento atteso in loro presenza
 - cercare di semplificare al massimo l'utilizzo del codice

Esercizi

- Scrivere (con Eclipse) una classe di test Junit per la classe **Persone** (dal Quiz di preparazione alla prima verifica)
- In particolare testare il metodo `int contaOmonimiDi(String nome)`
- Scrivere il codice del metodo `int contaOmonimiDi(String nome)`
- Eseguire la classe di test Junit (se il test fallisce, correggere il metodo sotto test e far girare nuovamente la classe di test)

```
public class Persone {  
    private String[] nomi;  
  
    public Persone(int n) {  
        this.nomi = new String[n];  
    }  
  
    public int contaOmonimiDi(String nome) {  
        // metodo da scrivere  
    }  
  
    public void aggiungiNome(int indice, String nome){  
        this.nomi[indice] = nome;  
    }  
}
```