

Test Right-BICEP

- Consigli pratici su cosa testare: Right-BICEP
- **Right**: i risultati sono corretti?
 - Validare i risultati
 - I risultati attesi sono conformi a ciò che il metodo fa?
- Se non si riesce a stabilire cosa significa “right” allora i test non sono possibili e non è possibile sapere se il codice funziona bene
 - I requisiti potrebbero non essere stabili
 - Prendere delle decisioni per stabilire cosa è “right” e valutare le decisioni in seguito
 - Il codice di test può essere scritto in modo da prendere in ingresso dei dati che validano i risultati attesi, anziché includere tali dati al suo interno

Right-BICEP

- **Boundary condition** ovvero condizioni di confine (ai limiti)
 - Considerare gli estremi di range di valori, dati invalidi (come visto prima)
 - Dati formattati male (conforme ad un formato?)
 - Ordinamenti inattesi (ordinamento appropriato?)
 - Valori fuori da range consentiti (range?)
 - Considerare se il codice riferisce qualcosa di esterno (reference ok?)
 - Valori nulli o mancanti (il valore esiste?)
 - Dati duplicati, se non consentiti, e numero valori non consentito (cardinalità ok?)
 - Tutto sta avvenendo secondo l'ordine temporale desiderato (time)
 - Usare l'acronimo CORRECT
 - Conformance, ordering, range, reference, existence, cardinality, time

Right-BICEP

- Controllare relazioni **inverse**
 - Se un metodo fa qualcosa che ha un inverso, usare l'inverso
 - Es. quadrato e radice quadrata, inserimento e cancellazione
 - Alcuni errori sono comuni ad entrambe le operazioni (diretta e inversa)
- **Cross-check** (controllo incrociato) usando mezzi alternativi
- Se è possibile fare qualcosa in più modi controllare che i risultati siano consistenti per ciascun metodo usato
 - Stabilire fattori di consistenza

Right-BICEP

- Verificare condizioni di **errore**
 - Parametri invalidi, valori fuori da range, eccezioni, etc.
 - Fallimenti fuori dal nostro codice: out of memory, disco pieno, rete non disponibile, etc.
 - Simulare errori attraverso Mock Object
- **Performance**
 - Misurare le performance
 - Misurare come le performance cambiano al crescere degli input

Unit Test

- Uno unit test è un frammento di codice che esegue un altro frammento di codice
- Per verificare che il codice si comporta come ci si aspetta si usa una *assertion*, ovvero una chiamata di metodo che verifica se qualcosa è true
- `void assertTrue(boolean condition)` valuta se `condition` è true, se non è così il test ha trovato un errore
- `void assertEquals(int a, int b)` verifica se due int sono uguali
- I metodi assert registrano fallimenti o errori e li riportano

```
int a = 2;
...
assertTrue(a == 2);
...
assertEquals(a, 2);
```

E. Tramontana - Test JUnit - 21-Jan-09 5

JUnit

- Junit è un framework per il test di programmi Java

```
import junit.framework.TestCase;
public class TestCalculator extends TestCase {
    public TestCalculator(String name) {
        super(name);
    }
    public void testAddition() {
        Calculator calc = new Calculator();
        double result = calc.add(10, 50);
        assertEquals(60, result, 0);
    }
}
```

```
javac -cp junit.jar *.java
java -cp junit.jar junit.swingui.TestRunner TestCalculator
```

E. Tramontana - Test JUnit - 21-Jan-09 6

JUnit

- Quando si verifica un fallimento o un errore, l'esecuzione del metodo di test viene interrotta, ma verranno eseguiti gli altri metodi di test della stessa classe

```
assertEquals([String message], expected, actual)
assertEquals([String message], expected, actual, tolerance)
assertEquals("dovrebbe essere 3 e 1/3", 3.33, 10.0/3.0, 0.01)
```

- Altri metodi assert
 - `assertNull(Object object)`
 - `assertSame(expected, actual)`
 - `assertTrue(boolean condition)`
 - `assertFalse(boolean condition)`
 - `fail(String message)`

E. Tramontana - Test JUnit - 21-Jan-09 7

JUnit Framework

- Ogni classe che contiene test (`TestCalculator`) deve estendere la classe `TestCase` del framework JUnit
 - La classe `TestCase` fornisce i metodi assert usati
 - La classe `TestCalculator` contiene metodi i cui nomi iniziano con `test`
- Tutti i metodi che iniziano con `test` saranno eseguiti da JUnit
 - Il framework è basato sulla riflessione computazionale

```
public void testAdd() {
    assertEquals(60, 20+40);
    assertEquals(8, 4+4);
}
```

E. Tramontana - Test JUnit - 21-Jan-09 8

JUnit Composizione di test

- Possiamo creare una suite di test tramite un metodo statico chiamato `suite()`

```
import junit.framework.Test;
import junit.framework.TestSuite;
public class TestAll {
    public static Test suite() {
        TestSuite suite = new TestSuite("All tests");
        suite.addTestSuite(TestCalculator.class);
        suite.addTestSuite(TestClassA.class);
        ...
        return suite;
    }
}
```

```
java -cp junit.jar junit.swingui.TestRunner TestAll
```

E. Tramontana - Test JUnit - 21-Jan-09 9

JUnit

- Il metodo `addTestSuite(Class c)` aggiunge alla lista di metodi di test da eseguire tutti quelli della classe `c`
- Per resettare l'ambiente prima dell'esecuzione di test, la classe `TestCase` fornisce i metodi `setUp()` e `tearDown()` che possono essere ridefiniti per gestire l'ambiente
- Il metodo `setUp()` sarà chiamato (automaticamente) prima di eseguire ciascun metodo `test...` ed il metodo `tearDown()` sarà chiamato dopo l'esecuzione di ciascun metodo `test...`

E. Tramontana - Test JUnit - 21-Jan-09 10

JUnit in Eclipse

- Implementare casi di test con JUnit in Eclipse
 - Selezionare File -> New -> JUnit Test Case, selezionare JUnit 3, selezionare una classe ed i metodi, verrà generata una classe con i metodi `test...` corrispondenti ai metodi della classe che si vuole testare
- Per eseguire Run as -> JUnit Test
- Per misurare la copertura del codice (code coverage)
 - Installare il tool EclEmma da <http://update.eclEmma.org>
 - Una volta eseguiti i casi di test scritti con JUnit sarà fornita la misura di code coverage e saranno evidenziate le linee di codice eseguite

E. Tramontana - Test JUnit - 21-Jan-09 11