

Corso di Ingegneria del Software

Testing con jUnit

Scrittura dei test di unità

- Il testing a livello di unità dei comportamenti di una classe dovrebbe essere progettato ed eseguito dallo sviluppatore della classe, contemporaneamente allo sviluppo stesso della classe
- **Vantaggi:**
 - Lo sviluppatore conosce esattamente le responsabilità della classe che ha sviluppato e i risultati che da essa si attende
 - Lo sviluppatore conosce esattamente come si accede alla classe, ad esempio:
 - Quali preconditioni devono essere poste prima di poter eseguire un caso di test;
 - Quali postcondizioni sui valori dello stato degli oggetti devono verificarsi
- **Svantaggi:**
 - Lo sviluppatore tende a difendere il suo lavoro ... troverà meno errori di quanto possa fare un tester!

Testing Automation

- L'automazione dell'esecuzione dei casi di test porta innumerevoli vantaggi:
 - Tempo risparmiato (nell'esecuzione dei test)
 - Affidabilità dei test (non c'è rischio di errore umano nell'esecuzione dei test)
 - Si può migliorare l'efficacia a scapito dell'efficienza sfruttando il fatto che l'esecuzione dei test è poco costosa
 - Riutilizzo (parziale) dei test a seguito di modifiche nella classe

Testing basato su “main”

- Scrivere un metodo di prova (“main”) in ogni classe contenente del codice in grado di testare i suoi comportamenti
 - Problemi
 - Tale codice verrà distribuito anche nel prodotto finale, appesantendolo
 - Come strutturare i test case? Come eseguirli separatamente?
- Per tutti questi motivi, cerchiamo un approccio sistematico
 - Che separi il codice di test da quello della classe
 - Che supporti la strutturazione dei casi di test in test suite
 - Che fornisca un output separato dall'output dovuto all'esecuzione della classe

Famiglia X-Unit

- La soluzione alle problematiche precedenti é data dai framework della famiglia X-Unit:
 - JUnit (Java)
 - È il capostipite; fu sviluppato originariamente da Erich Gamma and Kent Beck
 - CppUnit (C++)
 - csUnit (C#)
 - NUnit (.NET framework)
 - HttpUnit, HTMLUnit (Web Application)
 - JSUnit (JavaScript)
 - PHPUnit (Php)

Componenti di un test XUnit

- Una classe di test, contenente:
 - Un metodo *setup()* che viene eseguito prima dell'esecuzione di ogni test
 - Utile per settare precondizioni comuni a più di un caso di test
 - Un metodo *teardown()* che viene eseguito dopo ogni caso di test
 - Utile per resettare le postcondizioni
 - Un metodo per ogni caso di test

Struttura di un metodo di test

- **Inizializzazione precondizioni**
 - Limitatamente alle precondizioni tipiche del singolo caso di test, le altre potrebbero essere nel *setup*
- **Inserimento valori di input**
 - Tramite chiamate a metodi set oppure tramite assegnazione di valori ad attributi pubblici
- **Codice di test**
 - Esecuzione del metodo da testare con gli eventuali parametri relativi a quel caso di test
- **Valutazione delle asserzioni**
 - Controllo di espressioni booleane (*asserzioni*) che devono risultare vere se il test dà esito positivo, ovvero se i dati di uscita e/o le postcondizioni riscontrati sono diversi da quelli attesi

JUnit

- JUnit é un framework di unit testing per il linguaggio di programmazione Java.
- Plug-ins che supportano il processo di scrittura ed esecuzione dei test JUnit su classi Java sono previsti da alcuni ambienti di sviluppo
 - in particolare mostreremo il funzionamento del plug-in JUnit di Eclipse

Plug-in di Eclipse per JUnit

- Eclipse é dotato di plug-ins, di pubblico dominio, che supportano tutte le operazioni legate al testing di unità con JUnit, CppUnit, etc.. In particolare, essi forniscono dei wizard per:
 - Creare classi contenenti test cases
 - Automatizzare l'esecuzione di tutti i test cases
 - Mostrare i risultati dell'esecuzione dei casi di test
 - Organizzare i test cases in test suites

JUnit 4 Classi di Test

```
package mioProgetto;  
import static org.junit.Assert.*;  
import org.junit.Test;  
public class miaClasseTest {  
    @Test  
    public void testprimaClasseDiTest() {  
        ...Codice  
  
        assertEquals(valoreAtteso, valoreDaEsaminare);  
    }  
    @Test  
    public void testsecondaClasseDiTest() {  
        ...Codice  
  
        assertNull(valoreCheDeveEssereNullo);  
    }  
    ...  
}
```

import di classi ed
annotazioni JUnit

Nome della classe
di Test

Annotazione di metodo come test-case

Asserzione

JUnit: Osservazioni

- Tutte le classi di test che scriveremo avranno questa struttura
- Ovviamente le classi di test vanno progettate sulla base delle peculiarità della classe testata
- `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 (obbligatorio in Junit 3)

`@Test`

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

Assertzioni

- **Assertzione**

- affermazione che può essere vera o falsa
- I risultati attesi sono documentati con delle *assertzioni* esplicite, non con delle stampe che comunque richiedono dispendiose ispezioni visuali dei risultati
- Se l'assertzione 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 di esecuzione
- Le assertzioni sono utilizzate sia per verificare gli oracoli che le postcondizioni
 - Le assertzioni potrebbero essere utilizzate anche per verificare le precondizioni: se la precondizione fallisce, il test non viene proprio eseguito (e viene riportato come fallito)

Assertzioni

- 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()**
 - ...

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 di unità in pratica

```
public void testMetodoDaTestare() {  
    miaClasse mioOggetto = new miaClasse("par1", "par2");  
    assertEquals("par1", mioOggetto.getNome());  
    assertEquals("par2", mioOggetto.getParametro());  
}
```

- mettere un “frammento” del sistema in uno stato noto
 - il frammento comprende un solo oggetto **mioOggetto**
- inviare una serie di messaggi noti
 - nell'esempio solo la costruzione dell'oggetto, in generale ci potrebbero essere altre invocazioni di metodo sullo stesso
- controllare che alla fine il sistema si trovi nello stato atteso
 - tramite le asserzioni si controlla che il nome ed il parametro del comando siano quelli attesi

JUnit in ambiente Eclipse

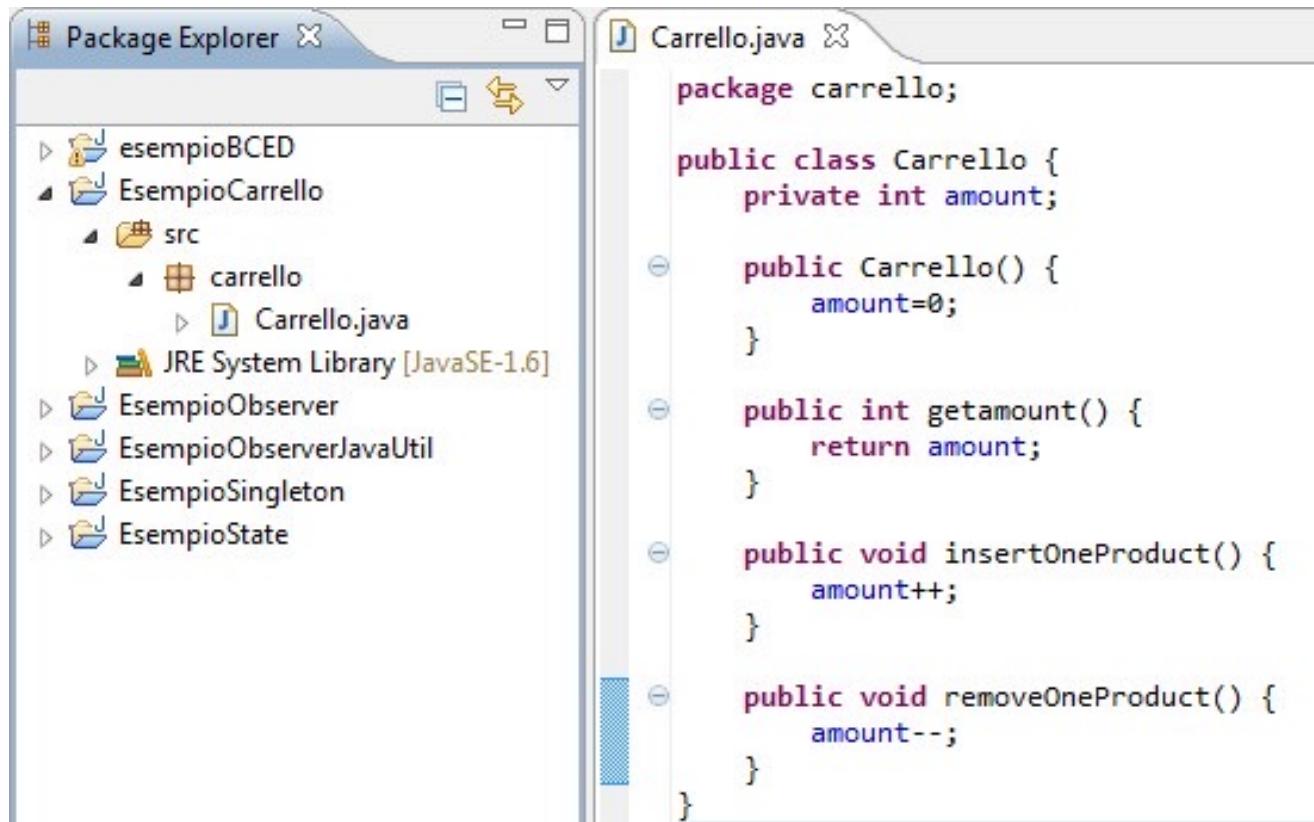
- Eclipse mette a disposizione wizard per velocizzare la scrittura di classi di test
- Visualizzazione grafica dei risultati dei test
- Possibilità di eseguire in un solo click un singolo test case

Esempio

- Una classe per la gestione del carrello di un sito di E-Commerce
 - Nome della classe: **Carrello**
 - Metodi implementati:
 - **getAmount()** - restituisce il numero di elementi nel carrello
 - **insertOneProduct()** – incrementa di uno il numero di elementi nel carrello
 - **removeOneProduct()** – decrementa di uno il numero di elementi nel carrello

Tutorial JUnit in Eclipse

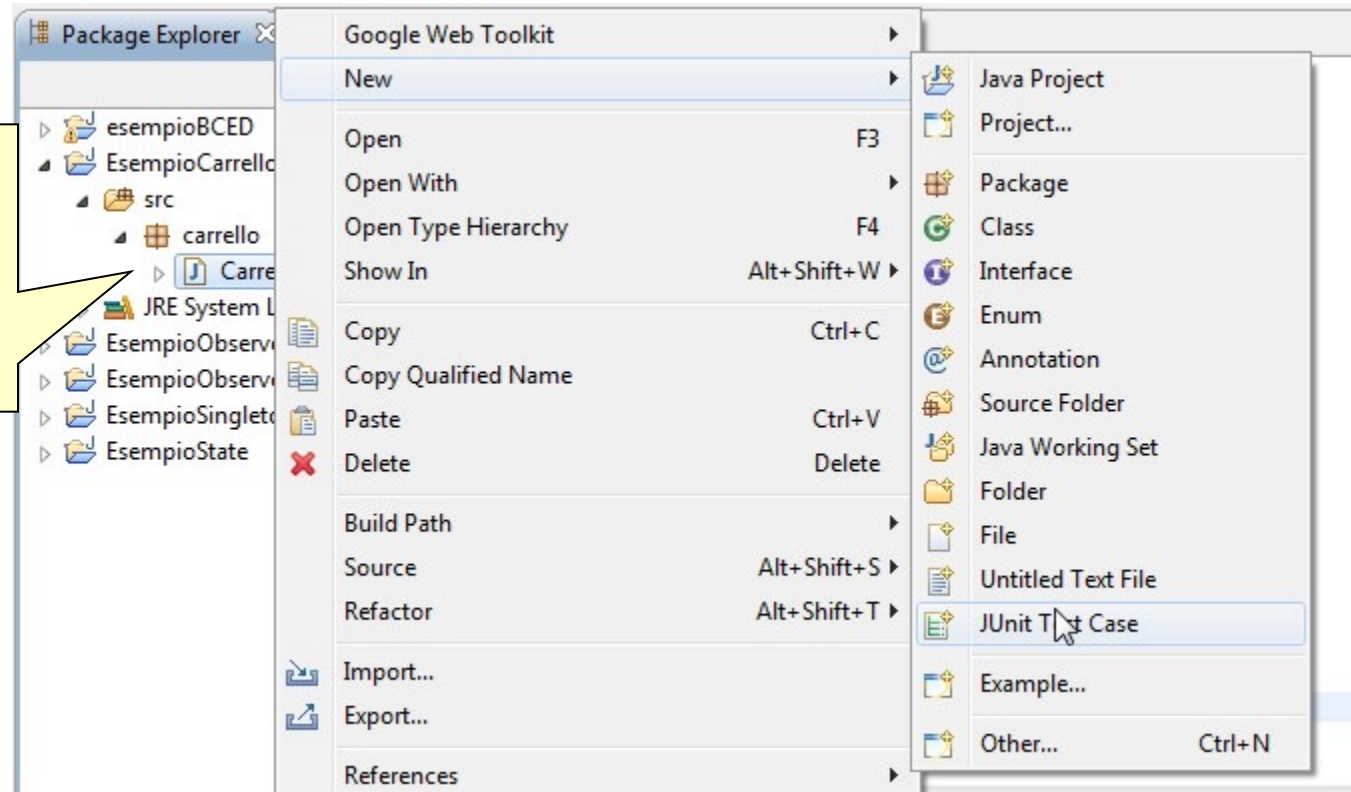
- Il codice della classe per la gestione di un Carrello.
 - Classe del Java Project “EsemplioCarrello” in Eclipse



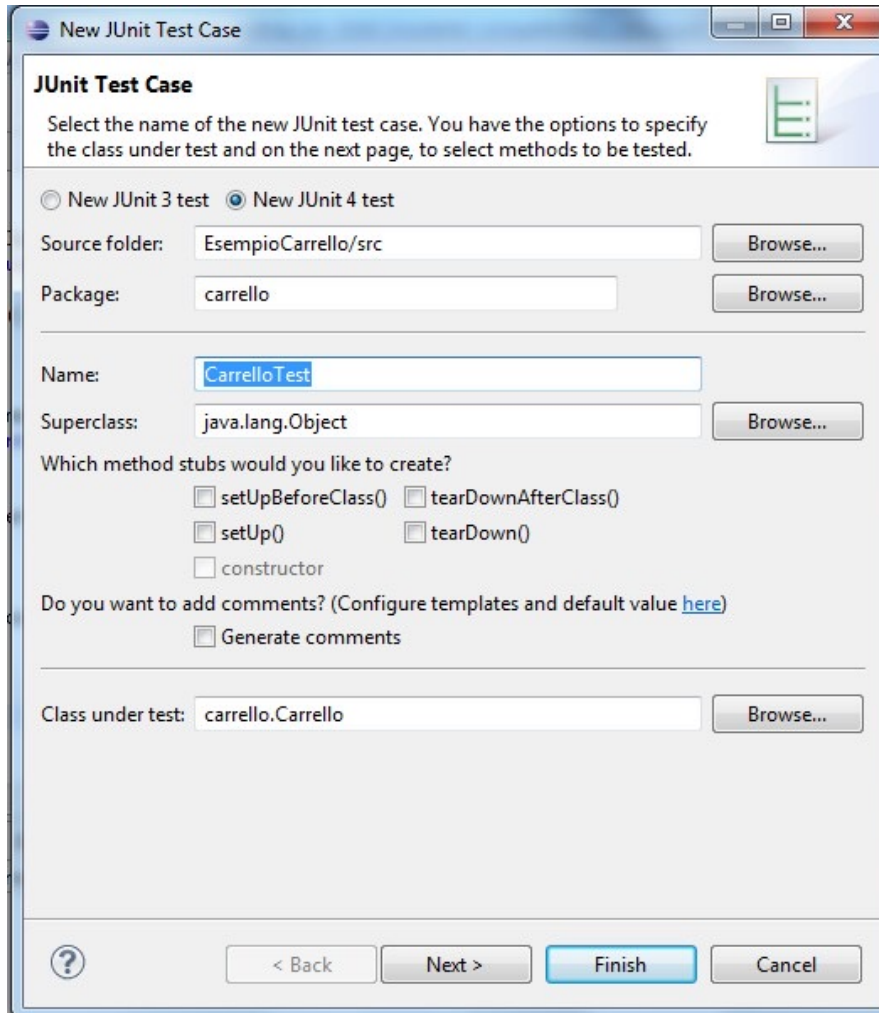
Tutorial JUnit in Eclipse

- Si vuole testare tramite JUnit il funzionamento dei metodi
 - insertOneProduct
 - removeOneProduct

Click con Tasto Destro sulla classe che si vuole testare
-New
-JUnit Test Case

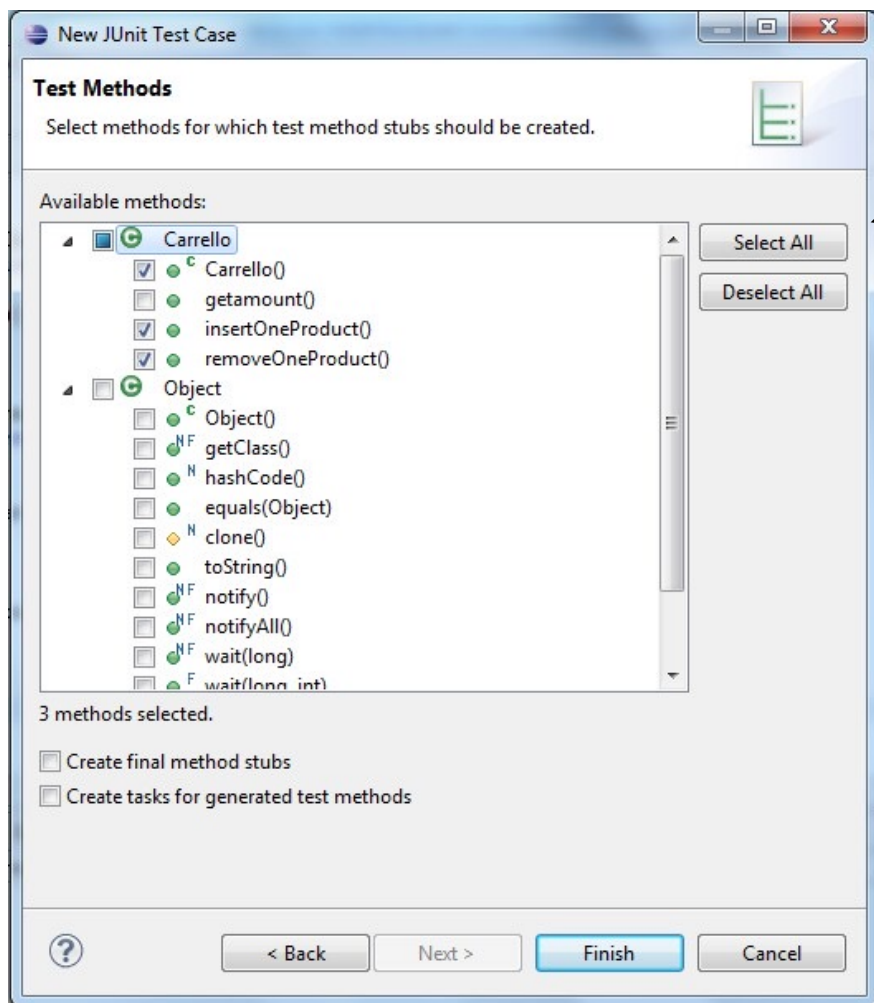


Tutorial JUnit in Eclipse



- Appare una nuova finestra che offre le seguenti funzionalità
 - selezionare la classe da testare
 - Creazione di fixtures
- Importante: Impostare JUnit 4 per avere uniformità di sintassi.

Tutorial JUnit in Eclipse



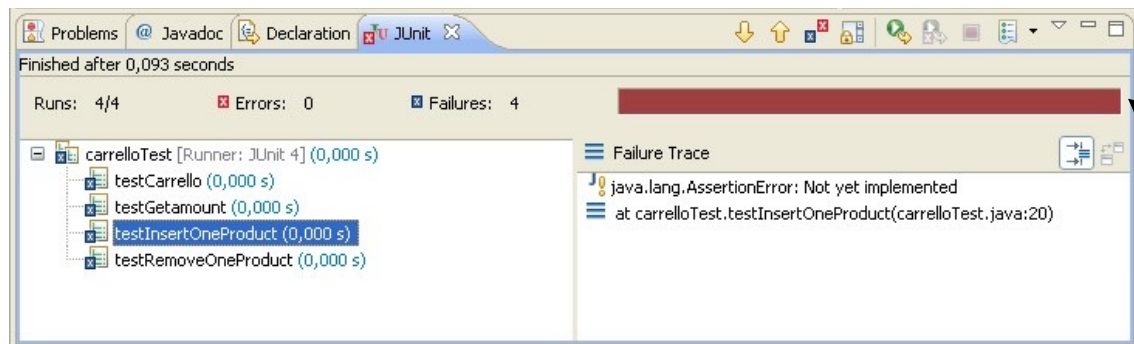
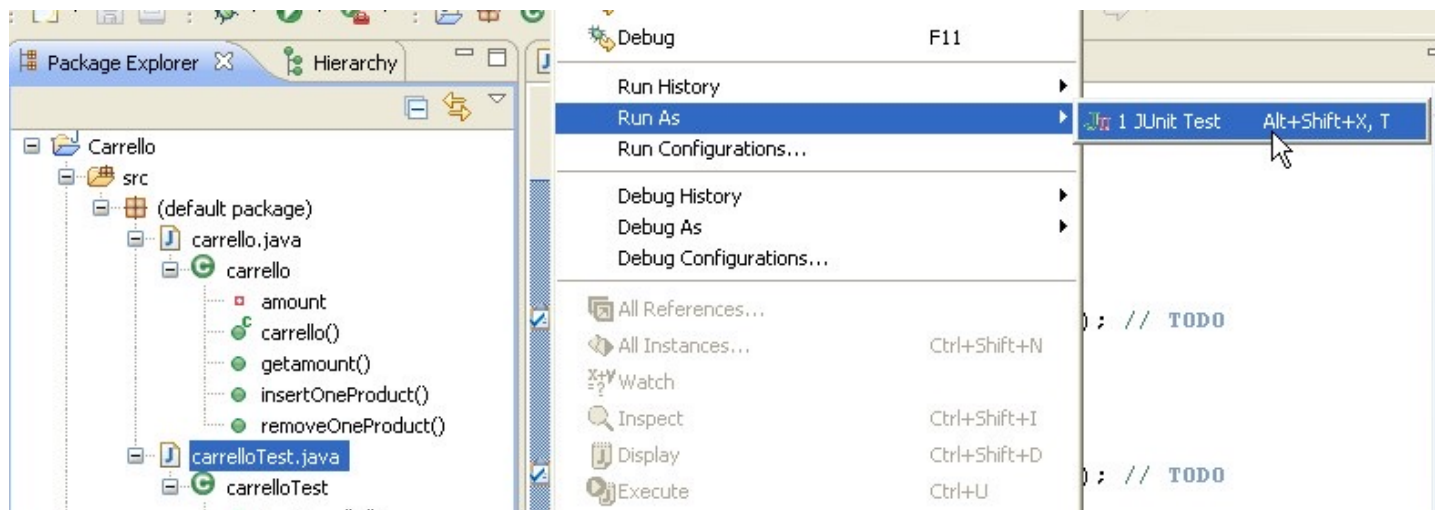
Si selezionano i metodi da voler testare

```
package carrello;  
  
import static org.junit.Assert.*;  
  
public class CarrelloTest {  
  
    @Test  
    public void testCarrello() {  
        fail("Not yet implemented");  
    }  
  
    @Test  
    public void testInsertOneProduct() {  
        fail("Not yet implemented");  
    }  
  
    @Test  
    public void testRemoveOneProduct() {  
        fail("Not yet implemented");  
    }  
  
}
```

Viene creato lo scheletro dei Test

Tutorial JUnit in Eclipse

- Avviare l'esecuzione dei test



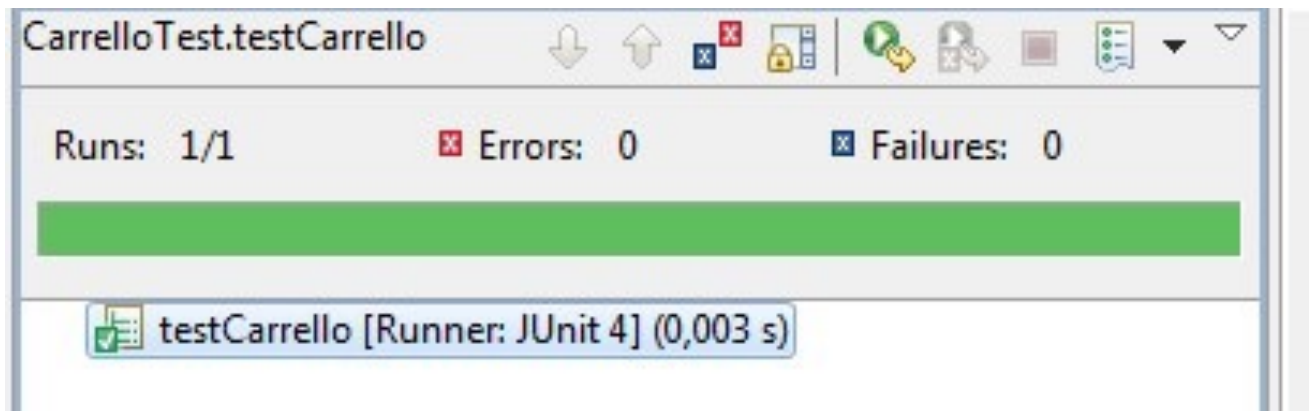
Un semaforo per ogni test
-Verde: test riuscito
-Rosso: test non riuscito

Esempio 1

- Verifica del costruttore

```
@Test
public void testCarrello() {
    Carrello carrello = new Carrello();
    int expected=0;
    assertEquals(expected,carrello.getamount());
}
```

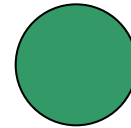
- Click tasto destro sulla classe di test testCarrello() → Run As → JUnit Test per eseguire il singolo test



Esempio 2

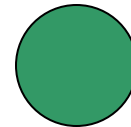
- Esempio di test che va a buon fine del metodo insertOneProduct()

```
@Test
public void testInsertOneProduct() {
    carrello Carrello=new carrello();
    int expected=3;
    Carrello.insertOneProduct();
    Carrello.insertOneProduct();
    Carrello.insertOneProduct();
    assertEquals(expected, Carrello.getamount());
}
```



- Esempio di test che va a buon fine del metodo removeOneProduct()

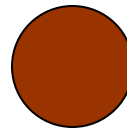
```
@Test
public void testRemoveOneProduct() {
    carrello Carrello=new carrello();
    int expected=1;
    Carrello.insertOneProduct();
    Carrello.insertOneProduct();
    Carrello.removeOneProduct();
    assertEquals(expected, Carrello.getamount());
}
```



Esempio 3

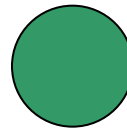
- C'è un test che ci permette di scoprire il malfunzionamento del metodo `removeOneProduct` ?

```
@Test
public void testRemoveOneProduct() {
    carrello Carrello=new carrello();
    int expected=0;
    Carrello.removeOneProduct();
    assertEquals(expected, Carrello.getamount());
}
```



Metodo dopo la correzione

```
public void removeOneProduct() {
    if (amount>0)
        amount--;
}
```



Fixtures

- Le fixture incoraggiano il riuso, definendo del codice che deve essere eseguito prima o dopo i test
- JUnit 4 ha reso le fixture esplicite con le **Annotazioni**
 - Codice fixture eseguito solo se necessario
 - Si può specificare che certe fixture siano eseguite prima o dopo di certi test
 - **codice riusato**

Annotazioni JUnit 4

- È possibile eseguire alcune azioni
 - Prima e dopo l'esecuzione di tutti i test di una classe di test
 - Prima e dopo l'esecuzione di un singolo test
- **@BeforeClass** definisce un metodo statico che viene eseguito prima di tutti i test contenuti in una classe di test
 - Il metodo "annotato" deve essere statico
- **@Before** (setUp in jUnit 3) definisce un metodo che viene eseguito prima di ogni metodo di test
- **@After** (tearDown in jUnit 3) definisce un metodo che viene eseguito dopo ogni metodo di test
- **@AfterClass** definisce un metodo statico che viene eseguito dopo che tutti i test contenuti in una classe di test sono stati eseguiti
 - Il metodo "annotato" deve essere statico

Osservazioni

- Si è utilizzato, finora, JUnit solo e soltanto per il testing di unità di singoli metodi
 - Può essere utile anche nel testing black box di sistemi completi
- JUnit supporta unicamente il testing, non il debugging
 - L'utilizzo di molte asserzioni può, però, portare ad indicazioni dettagliate sulle ragioni del successo del test case
- Si è data per scontata la correttezza del codice delle classi di test ma se avessimo voluto esserne sicuri al massimo avremmo potuto fare il test di unità delle classi di test stesse (ma il problema si sarebbe riproposto ricorsivamente!)
 - Il codice delle classi di test è comunque estremamente lineare e ripetitivo: la possibilità di sbagliare è ridotta!
 - La generazione del codice delle classi di test è automatizzabile in alcuni contesti

EclEmma

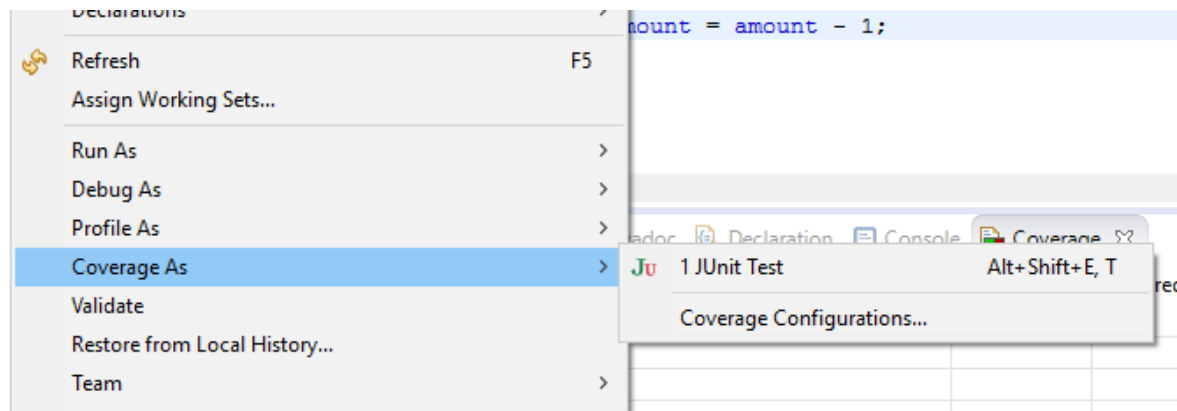
- EclEmma è un plug-in Open Source per Eclipse basato su Emma
 - <http://eclemma.org/>
- Si può installare seguendo le istruzioni alla pagina <http://eclemma.org/installation.html>
 - In particolare, nelle nuove versioni di Eclipse, è possibile scaricarlo dal Marketplace
 - Dal menù **Help**, selezionare **Eclipse Marketplace**
 - Ricercare **EclEmma**
 - Installa l'elemento "EclEmma Java Code Coverage"
 - Seguire il Wizard

Funzioni di EclEmma

- Valutare la copertura dei casi di test
- Mostrare direttamente nella finestra del codice sorgente le righe di codice coperte
- Generare statistiche riguardanti l'esito dei casi di test e report in formato HTML con tali risultati

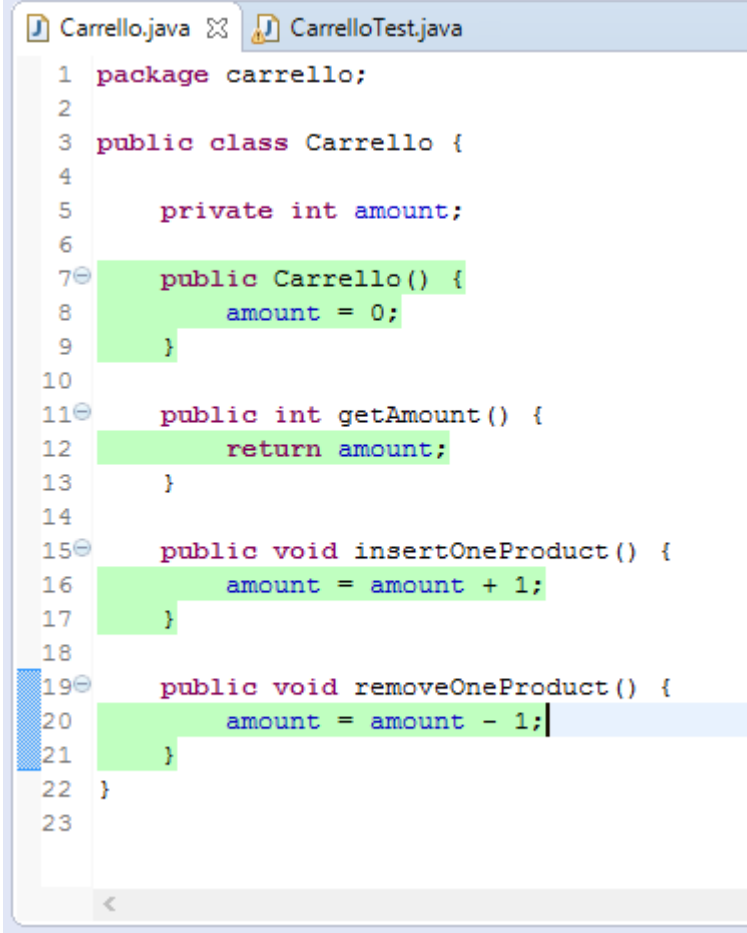
jUnit + EclEmma

- Per eseguire casi di test jUnit e valutarne la copertura con EclEmma
- Cliccare col tasto destro sulla classe di test
 - Dal menù contestuale selezionare «Coverage as...» e poi «JUnit Test»



Report di Copertura EcIEmma

- Dopo l'esecuzione dei test, nella finestra del codice saranno evidenziate
 - In verde, le linee di codice coperte
 - In giallo, le linee di codice coperte parzialmente
 - In rosso, le linee di codice non coperte



The screenshot shows an IDE with two tabs: Carrello.java and CarrelloTest.java. The Carrello.java file is open, displaying the following code with coverage highlights:

```
1 package carrello;  
2  
3 public class Carrello {  
4  
5     private int amount;  
6  
7     public Carrello() {  
8         amount = 0;  
9     }  
10  
11     public int getAmount() {  
12         return amount;  
13     }  
14  
15     public void insertOneProduct() {  
16         amount = amount + 1;  
17     }  
18  
19     public void removeOneProduct() {  
20         amount = amount - 1;  
21     }  
22 }  
23
```

The code is highlighted as follows: lines 7-9, 12, 16, and 20 are highlighted in green, indicating full coverage. Lines 11, 15, and 19 are highlighted in yellow, indicating partial coverage. Lines 1, 2, 3, 4, 5, 6, 10, 13, 14, 17, 18, 21, 22, and 23 are not highlighted, indicating no coverage.

Riferimenti

- Using JUnit in Eclipse, <http://www.cs.umanitoba.ca/~eclipse/10-JUnit.pdf>
- An Introduction to JUnit, <http://www.cs.toronto.edu/~cosmin/TA/2003/csc444h/tut/tut3.pdf>
- JUnit Testing Utility Tutorial, <http://supportweb.cs.bham.ac.uk/documentation/tutorials/docsystem/build/tutorials/junit/junit.html>
- Tools for automated software testing, http://www.elet.polimi.it/upload/picco/Teaching/softeng/slides/test_tools.pdf
- Introduzione a Test-First Design e JUnit, <http://www.lta.disco.unimib.it/didattica/Progettazione/lucidi/e09-testing+introJUnit.pdf>
- JUnit Testing Utility Tutorial, <http://supportweb.cs.bham.ac.uk/documentation/tutorials/docsystem/build/tutorials/junit/junit.html>
- Sito ufficiale di JUnit, <http://www.junit.org/index.htm>
- Progetto sourceforge di JUnit, <http://junit.sourceforge.net/>