Unit Testing: JUnit (xUnit) basics

Angelo Di Iorio

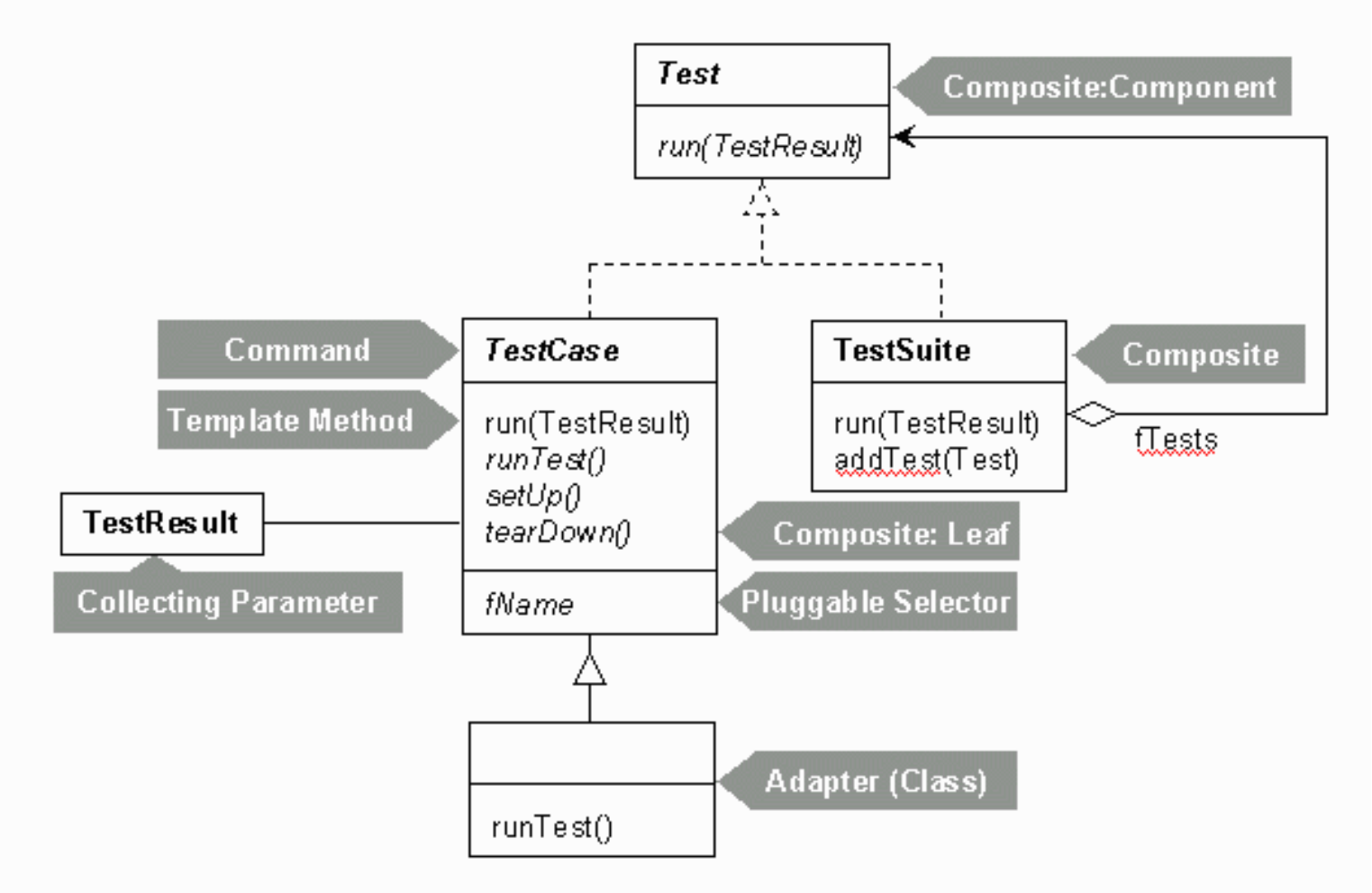
Tipi di test

- Esistono diversi tipi di test che permettono di verificare i requisiti funzionali di un sistema, a diversi livelli di granularità:
  - *Unit Testing*: unità individuali del sistema (classi singole, metodi, funzioni)
  - *Integration Testing*: gruppi di unità (module, component, sub-system)
  - *System Testing*: sistema completo
  - *Acceptance Testing*: per verificare che i requisiti dell'utente siano soddisfatti (casi d'uso)

Unit testing: JUnit

- JUnit è un framework Java per eseguire test di unità, scritto da Beck e Gamma
- Lo standard de facto per eseguire questo tipo di test, di cui sono stati fatti diversi porting: PHPUnit, PyUnit, Nunit
- Esistono inoltre diverse *estensioni* specializzate, che estendono classi e metodi della versione core:
  - *HTTPUnit*: testare il comportamento di un sito web (navigazione, form submission, cookie)
  - *DBUnit*: testare applicazioni database-driven (es. verificare risultati query, "congelare" database tra diversi test)
  - *XMLUnit*: testare conformità e validità di documenti XML e espressioni XPath
  - …

JUnit: struttura del framework



[da "JUnit: A Cook's Tour"]

TestCase

- I test sono organizzati in *TestCase*, a loro volta organizzati in *TestSuite*
- *TestCase* è una classe del framework da estendere per creare i test
  - Di solito ad ogni classe *XXX* è associato un test case chiamato *XXXTest*
- Ogni test richiama pochi metodi (meglio se uno solo) della classe da testare e verifica se l'output è quello atteso.
  - Il nome del test ha la struttura: textYYY dove YYY è un nome esplicativo del test che si sta eseguendo

Esempio: testiamo la classe Word

```
public class Word {

    // costruttore
    public Word(String s){}

    // stampa parola
    public String print(){}

    // conta lettere
    public Integer countLetters(){}

    // conta vocali
    public Integer countVowels(){}

    // inverti: ABCDE -> EDCBA
    public void reverse(){}
}
```

Struttura di base

```
import junit.framework.TestCase;
import org.junit.Test;

public class WordTest extends TestCase {

    @Test
    public void testPrint() {
      fail("Not yet implemented");
    }

    public void testCountLetters() {…}

    public void testCountVowels() {…}

    public void testReverse() {…}
}
```

Struttura di un test

- La struttura di un test è (e deve essere) molto semplice:
  1. Creare o ottenere le istanze della classe da testare
  2. Invocare il metodo da testare
  3. Verificare il risultato tramite asserzioni

Il test ha esito SUCCESS se tutte le asserzioni hanno successo, altrimenti fallisce.

Oltre alle condizioni FAILURE ci sono gli ERRORI che sono invece imprevisti e indicano un problema, eventualmente nello stesso test.

Struttura di base: testare *print()*

```
@Test
public void testPrint() {

    // crea istanza di Word
    String s = "Phone";
    Word p = new Word(s);

    // invoca metodo print
    String printedWord = p.print();

    // verifica output
    assertEquals(s, printedWord);              //success
    assertEquals("iPhone", printedWord);       //failure

}
```

Testare *countLetters()*

```
@Test
public void testCountLetters() {

    Word w0 = new Word("");
    assertEquals((Integer) 0, w0.countLetters()); //success

    String s1 = "Phone";
    Word w1 = new Word(s1);
    assertEquals((Integer) 5, w1.countLetters()); //success
    assertEquals((Integer) 15, w1.countLetters()); //failure

    assertEquals(15, w1.countLetters()); //error

}
```

Testare *countVowels()*

```
@Test
public void testVowels() {

    Word w0 = new Word("");
    assertEquals((Integer) 0, w0.countVowels()); //success

    String s1 = "Phone";
    Word w1 = new Word(s1);
    assertEquals((Integer) 2, w1.countVowels ()); //success

}
```

Asserzioni

- Le asserzioni indicano i “punti di controllo” in cui si verifica che il risultato ottenuto sia quello atteso
- Ne esistono di diversi tipi, con struttura *assertXXX(msg, v1, v2)* dove:
  - *XXX* è il tipo di controllo
  - *msg* è un messaggio di debug opzionale
  - *v1* e *v2* sono i valori da controllare

Asserzioni: esempi

- *assertEquals(integer expected, integer actual)*
- *assertEquals(double expected, double actual)*
  - *assertEquals(3.04, 1.52*2);*
  - *assertEquals(3.04, 1.52);*
- *assertNull(object object)*
  - *assertNull(null);*
- *assertNotNull(object object)*
  - *assertNotNull(null);*
- *assertTrue(boolean condition)*
  - *assertTrue(3 == 4)*
  - *assertTrue(3 == 3)*
- *assertFalse(boolean condition)*
  - *assertFalse(3 == 4)*
  - *assertFalse(3 == 3)*
- *fail()*
- …

Test Suite

- I *TestCase* possono essere raggruppati in *TestSuite*
  - manutenzione di test omogenei
  - chiarezza e organizzazione logica (i test sono utili anche come documentazione!)
  - esecuzione di tutti i test contenuti nella test-suite via Junit
- Il framework definisce una classe `TestSuite` e un metodo statico `suite()` per aggiungere dinamicamente tutti i metodi testXXX alla test-suite tramite reflection Java

Test Suite: esempio

```
import junit.framework.Test;
import junit.framework.TestSuite;

public class AllCalculatorTests {

    public static Test suite() {
      TestSuite suite = new TestSuite("Test for Calculator");
      //$JUnit-BEGIN$
      suite.addTestSuite(CalculatorTest.class);
      suite.addTestSuite(CalculatorZeroTest.class);
      //$JUnit-END$
      return suite;
    }
}
```

Fixtures

- E' comune avere diversi test sugli stessi oggetti.
- Per evitare di duplicare il codice di creazione e reset dei dati di test si usano le *fixture*.
- JUnit definisce inoltre due metodi `setUp()` e `TearDown()` eseguiti rispettivamente prima e dopo ogni metodo `testXXX()`
- Esistono poi due metodi statici metodi `setUpBeforeClass()` e `TearDownBeforeClass()` eseguiti all'inizio e alla fine dell'intero TestCase

```
public class WordFactorizedTest extends TestCase {

    private Word w0;
    private Word w1;

    @Before
    public void setUp() throws Exception {
    this.w0 = new Word("");
    this.w1 = new Word("Phone");
    }

    @After
    public void tearDown() throws Exception {
    }

    @Test
    public void testCountLetters() {
    assertEquals((Integer) 0, this.w0.countLetters());
    assertEquals((Integer) 5, this.w1.countLetters());
    }
}
```

Annotazioni

- Dalla versione 4 JUnit usa le annotazioni Java per caratterizzare i metodi ed indicare ad esempio:
  - che il metodo è usato per testare altri metodi: `@Test`
  - che il metodo deve essere eseguito prima/dopo ogni test: `@Before/@After`
  - che il metodo deve essere eseguito prima/dopo l'intero test-case: `@BeforeClass/@AfterClass`

- Le abbiamo usate negli esempi finora!

Eseguire i test

- JUnit è un framework Java e può essere eseguito in vari modi
  - stand-alone, con risultati su console
  - stand-alone, con interfaccia grafica (integrata nel pacchetto JUnit)
  - strumenti di deploy e esecuzione automatica (ANT, Maven, etc.)
  - direttamente nei più comuni IDE, tramite plugin (in molti casi JUNit è già incluso, es. in Eclipse)

Demo in Eclipse

Qualche linea guida…

- I test dovrebbero essere scritti prima del codice (Test-Driven-Development)
- Scrivere un test per tutto ciò che può generare errori
- I test devono essere indipendenti e semplici (così come classi e metodi da testare)
- Eseguire i test spesso
- Aggiornare ed estendere i test

…e una massima

“Whenever you are tempted to type something into a print statement or a debugger expression, write it as a test instead.”

[Martin Fowler]

xUnit tools

- Junit, http://junit.org/
- HTTPUnit, http://httpunit.sourceforge.net/
- DBUnit, http://www.dbunit.org/
- XMLUnit, http://xmlunit.sourceforge.net/

- PHPUnit, http://www.phpunit.de/
- PyUnit, http://pyunit.sourceforge.net/