Unit Testing con JUnit 4 & Mockito

Andrea Fornaia, Ph.D.

Department of Mathematics and Computer Science

University of Catania

Viale A.Doria, 6 - 95125 Catania Italy

fornaia@dmi.unict.it

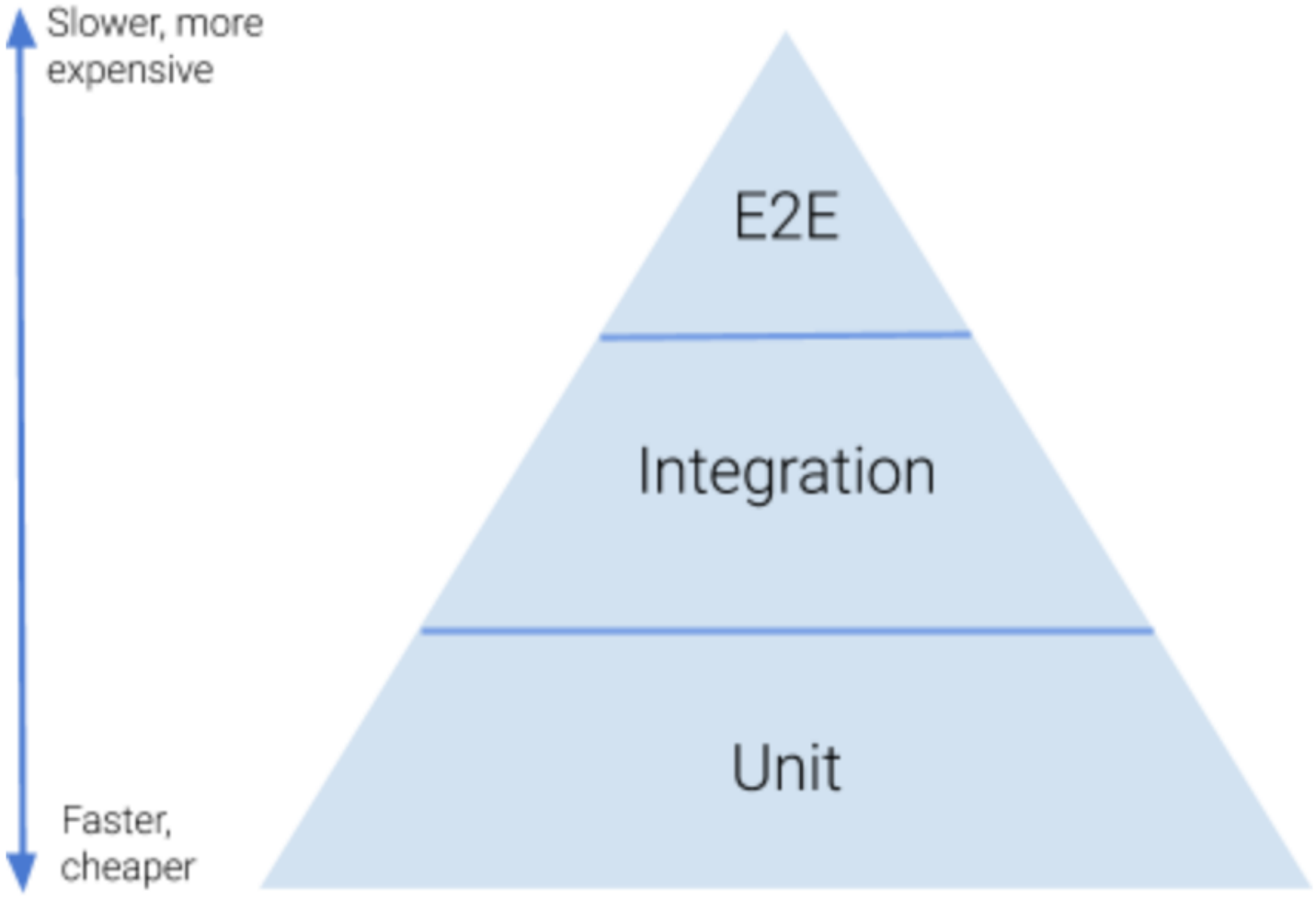
https://www.dmi.unict.it/fornaia/

A.A. 2020/2021

Unit Test

- Uno **unit test** esamina il comportamento di una unità di lavoro. Spesso questa unità di lavoro è identificata come un singolo metodo
  - Es. se forniamo il valore x al metodo m(), restituirà y?
- Uno unit test **verifica** che un metodo segua i termini del contratto definiti dalla sua API, ovvero l'accordo fatto con la signature del metodo
- Uno unit test permette inoltre di fornire una **documentazione eseguibile** relativa alle modalità di utilizzo di un determinato metodo
- Altre tipologie di test:
  - Un **integration test** verifica invece il corretto funzionamento di più unità che interagiscono tra loro
  - Un **end-to-end test** verifica il comportamento dell'intero sistema, ad esempio dal punto di vista dell'utente finale tramite interazioni con l'interfaccia grafica o le API esposte

La piramide dei test



JUnit

- JUnit è un framework per facilitare l'implementazione di programmi di test in Java
- Vedremo JUnit 4
- Princìpi del framework JUnit
  - Ciascuno unit test è un metodo
  - Ogni metodo di test fornito è trovato ed eseguito dal framework usando la riflessione computazionale
  - Si usano istanze di classi di test separate e class loader separati, per evitare effetti collaterali, quindi per avere esecuzioni indipendenti
  - Si hanno una varietà di **asserzioni** per automatizzare il controllo dei risultati dei test
  - Integrazione con tool e IDE diffusi (Eclipse, IntelliJ, VS Code)

Esempio

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

public class TestCalc { // classe di test
   @Test
   public void testAdd() { // l'annotazione indica il test
       Calculator c = new Calculator(); // crea istanza
       double result = c.add(10, 50); // chiama metodo da testare
       assertEquals(60, result, 0); // controlla il risultato e
   }                                  // genera eccezione se result != 60
}
```

Fondamenti

- JUnit crea una **nuova istanza** della classe di test prima di invocare ciascun metodo annotato con `@Test`
  - Per evitare effetti indesiderati
  - Non si possono quindi riusare variabili fra un metodo ed un altro
  - Ogni test è indipendente dagli altri

```
public class TestCalc {
  @Test
  public void testAdd1() { ... }

  @Test
  public void testAdd2() { ... }
}
```

Fondamenti

- Per verificare se il codice si comporta come ci si aspetta si usa una *assertion*, ovvero una chiamata al metodo assert, che verifica se il risultato ottenuto (*actual*) coincide con il risultato atteso (*expected*)
- La classe che fornisce i metodi usati per valutare le esecuzioni è la classe `Assert`
- `assertTrue(boolean condition)` valuta se `condition` è `true`, se non è così il test ha trovato un errore
- `assertEquals(int a, int b)` verifica se due int sono uguali
- I metodi assert registrano fallimenti o errori e li riportano
- 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

Fondamenti

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

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

Test Suite

- Per eseguire tante classi di test si usa un oggetto chiamato `Suite`

```
public class TestCaseA { // costruisco i casi di test
   @Test public void testA1() { ... }
}
public class TestCaseB {
   @Test public void testB1() { ... }
}

@RunWith(Suite.class) // Runner: classe che esegue i test
@SuiteClasses({TestCaseA.class}) // indico i casi di test della suite
public class TestSuiteA { }

@RunWith(Suite.class)
@SuiteClasses({ TestSuiteA.class, TestSuiteB.class })
public class MasterTestSuite { }
```

@Before e @After

- Un metodo annotato con **`@Before`** (setUp) viene eseguito prima dell'esecuzione di ciascun metodo `@Test`
  - Serve ad inizializzare lo stato prima del test
- Analogamente, un metodo **`@After`** (tearDown) viene eseguito dopo l'esecuzione di ciascun metodo `@Test`
  - Comunque vada l'esecuzione
  - Serve a portare il sistema ad uno stato opportuno (clean)
- **`@BeforeClass`** annota un metodo **statico** che verrà chiamato **solo una volta** prima dell'esecuzione di tutti i metodi `@Test`
  - Utile per eseguire operazioni costose, es. apertura connessione con un database
  - Essendo un metodo statico, può modificare solo attributi statici della classe
  - Può rendere i test meno indipendenti tra loro
- Analogamente **`@AfterClass`**

Esempio

```
public class TestCalc2 {
  private Calculator calc;
  @Before public void setUp() {
    calc = new Calculator();
  }
  @Test public void testAdd() {
    double result = calc.add(10, 50);
    assertEquals(60, result, 0);
  }
  @Test public void testSub() {
    double result = calc.sub(30, 20);
    assertEquals(20, result, 0);
  }
}
```

Convenzioni di Naming

Metodo da testare: int sum(int a, int b) throws NegativeNumbersException

Scegliere una convenzione e usarla coerentemente per tutti i test del progetto

- test[nome_metodo]
  - *testSum (ci limita ad un solo test case!)*
- test[feature_da_testare]
  - *testSumBiggerThenZeroNumbers*
  - *testSumWithNegativeNumber*
- when[stato_da_testare]_expect[comportamento_atteso]
  - *whenBiggerThenZeroNumbers_expectReturnSum*
  - *when_negativeNumber_expect_throwException*
- should[comportamento_atteso]_when[stato_da_testare]
  - *shouldReturnSum_whenBiggerThenZeroNumbers*
  - *should_returnSum_when_biggerThenZeroNumbers*
- given[precondizioni]_when[stato_da_testare]_then[comp_atteso] **(BDD)**
  - *given_initializedCalculator_when_negativeNumber_expect_throwException*
  - *given_initializedCalculator_when_biggerThenZeroNumbers_expect_returnSum*

Assertj

```xml
<dependency>
   <groupId>org.assertj</groupId>
   <artifactId>assertj-core</artifactId>
   <version>3.16.1</version>
   <scope>test</scope>
</dependency>
```

- AssertJ può essere usata assieme a Junit per definire asserzioni complesse con uno stile *fluent*
- Permette di scrivere asserzioni tramite concatenazione di più metodi, migliorando la leggibilità del codice di test
- Molto estesa e completa

```java
// JUnit Assertions
assertTrue(hobbit.startsWith("Fro"));
assertTrue(hobbit.endsWith("do"));
assertTrue(hobbit.equalsIgnoreCase("frodo"));

// AssertJ
assertThat(hobbit)
   .startsWith("Fro")
   .endsWith("do")
   .isEqualToIgnoringCase("frodo")
```

Mock e Dependence Injection

- Uno unit test, a differenza di un integration test, deve testare una singola unità (es. metodo) in maniera indipendente dalle classi con cui **collabora**
- Necessario simulare il comportamento delle componenti da cui il metodo testato dipende tramite dei Mock (fantocci)
- Un Mock è un'istanza di una classe il cui comportamento di alcuni dei suoi metodi viene definito direttamente nel test
- Deve essere possibile passare il mock al metodo come dipendenza (es. tramite costruttore della classe, parametro, o metodo setter): questa pratica viene detta **dependence injection**
- É buona norma fare in modo che la classe si leghi ad un'interfaccia invece che ad una classe, disaccoppiandola quindi da un'implementazione specifica: il mock verrà creato come implementazione di questa interfaccia
- Questo permette di testare un metodo anche quando una classe da cui dipende non è stata ancora implementata o quando il comportamento della dipendenza varia ad ogni invocazione (es. generatore di valori casuali)

Esempio di Mock senza uso di librerie

- Supponiamo di voler testare il metodo sum di una classe *Consumer* che invoca al suo interno il metodo *sum* definito nell'interfaccia *Sommatore*
- Un'istanza compatibile con l'interfaccia sommatore viene passata in questo caso come dipendenza al costruttore della classe da testare
- Nel test possiamo definire un'implementazione dell'interfaccia come *classe anonima*, implementando tutti i metodi dell'interfaccia (in questo caso solo uno) in modo che si limiti a restituire al metodo da testare un risultato predefinito

```
@Test
public void testSumTwice() {
    int a = 2;
    int b = 3;
    int expected = 10;

    Consumer consumer = new Consumer(new Sommatore() { // dependence injection
        @Override
        public int sum(int a, int b){
            return 5;
        }
    });

    int res = consumer.sumTwice(a,b); // al suo interno: return 2 * sommatore.sum(a,b);
    assertThat(res).isEqualTo(expected);
}
```

Mockito

```xml
<dependency>
   <groupId>org.mockito</groupId>
   <artifactId>mockito-core</artifactId>
   <version>3.3.3</version>
   <scope>test</scope>
</dependency>
```

- Mockito è una libreria per semplificare la creazione dei Mock
- Durante l'esecuzione di un test, mockito permette:
  - di definire il valore da restituire quando un metodo del mock viene chiamato durante l'esecuzione del test (when/given)
  - di verificare se e con quali parametri un determinato metodo del mock sia stato chiamato durante l'esecuzione del test (verify/then)
- A differenza dell'approccio tramite classi astratte, non siamo obbligati a definire il comportamento di tutti i metodi dell'interfaccia

Esempio di Mock con Mockito

```java
public class ConsumerTest {
    @Mock
    Sommatore sommatore;

    @Before
    public void setUp() {
        initMocks(this);
    }

    @Test
    public void testSumTwice_withMockito() {
        int a = 2;
        int b = 3;
        int expected = 10;

        // given
        when(sommatore.sum(a,b)).thenReturn(5);

        // when
        Consumer consumer = new Consumer(sommatore);
        int res = consumer.sumTwice(a,b);

        // then
        verify(sommatore, times(1)).sum(a,b); // posso verficare se è stato chiamato
                                              // una sola volta con i valori di “a” e “b”
        assertThat(res).isEqualTo(expected);
    }
}
```

Test Double

- **Dummy** objects are passed around but never actually used. Usually they are just used to fill parameter lists.
- **Fake** objects actually have working implementations, but usually take some shortcut which makes them not suitable for production (an in memory database is a good example).
- **Stubs** provide canned answers to calls made during the test, usually not responding at all to anything outside what's programmed in for the test.
- **Spies** are stubs that also record some information based on how they were called. One form of this might be an email service that records how many messages it was sent.
- **Mocks** are objects pre-programmed with expectations which form a specification of the calls they are expected to receive.

I termini **mock** e **stub** sono i più usati, spesso riferendosi alla stessa cosa

[https://martinfowler.com/articles/mocksArentStubs.html]

State VS Behavior Verification

- Supponiamo di avere una classe da testare *PhoneBookService* che collabora con *PhoneBookRegistry*
- Il metodo da testare *PhoneBookService.register()* chiama i metodi di *PhoneBookRegistry* per aggiornarne lo stato (aggiunta nuovo contatto)
- Un approccio orientato alla **verifica dello stato** creerebbe un'**asserzione sull'output** del metodo chiamato o **sul cambiamento di stato dell'oggetto**
- In questo caso però il metodo da testare non ha un valore di ritorno e il cambiamento di stato avviene su un collaboratore (il registro)
- Un approccio orientato alla **verifica del comportamento**, invece, **verifica la correttezza della sequenza di chiamate** verso i metodi dei collaboratori (con l'uso di *verify() sul mock)*
- In questo caso, invece di verificare che il registro abbia un nuovo contatto, ci limitiamo a verificare che il metodo di inserimento sia stato invocato
- Il secondo approccio, reso possibile dai mock, ci permette di testare *PhoneBookService* indipendentemente da un'implementazione di *PhoneBookRegistry*

Verifica del comportamento

```
public class PhoneBookService {
   PhoneBookRegistry registry;              // ci si lega ad un'interfaccia
   public PhoneBookService(PhoneBookRegistry registry) {
       this.registry = registry;
   }
   public void register(String name, String number) {
       if (! registry.contains(name))       // solo se contains() da false
           registry.insert(name, number);   // riesco a testare questa linea
   }
}
```

```
public class PhoneBookServiceTest {
   @Mock private PhoneBookRegistry registry;
   PhoneBookService service;
   @Before
   public void setUp() {
       initMocks(this);
       service = new PhoneBookService(registry);
   }
   @Test
   public void testRegister() {
       String name = "Andrea", number = "123456";
       when(registry.contains(name)).thenReturn(false);
       service.register(name, number);
       verify(registry).insert(name, number); // verifico se insert è stata
                                              // chiamata con name e number
   }
}
```

Behaviour Driven Development

*org.mockito.BDDMockito.**

```
@Test
public void testRegister_withoutBDD() {
   String name = "Andrea", number = "123456";
   when(registry.contains(name)).thenReturn(false);
   service.register(name, number);
   verify(registry).insert(name, number);

}
```

BDD: given/when/then

```
@Test
public void testRegister_withBDD() {
   String name = "Andrea", number = "123456";
   // given
   given(registry.contains(name)).willReturn(false);
   // when
   service.register(name, number);
   // then
   then(registry).should().insert(name, number);
}
```

Esempio con comportamento non deterministico

```
public class SubServiceTest {

   @Mock private RandomGenService randomGenService;

   @Before
   public void setUp() throws Exception {
       initMocks(this);
   }

   @Test public void testCreateRandomMul() {
       // given (mocked random generator will return 555, then 333)
       given(randomGenService.genRandomNumber())
           .willReturn(555, 333);

       // when
       SubService subService = new SubServiceImpl(randomGenService);
       Sub sub = subService.createRandomSub();

       // then
       assertThat(sub.getA()).isEqualTo(555);
       assertThat(sub.getB()).isEqualTo(333);
       assertThat(sub.getResult()).isEqualTo(222);

   }
}
```

Test Parametrici

- @RunWith specifica il Runner, la classe del framework che esegue i test (**Parameterized.class** per i test parametrici)
- @Parameters annota un metodo statico che restituisce i set di parametri da usare per ciascun test case
- Ogni set di parametri viene passato al costruttore della classe di test, creando un'istanza per ciascun test case; i parametri verranno passati ai metodi di @Test tramite gli attribuiti della classe

```
@RunWith(Parameterized.class)
public class SommatoreParamTest {
    private int a;  // variabili usate nel test
    private int b;
    private int expected;

    @Parameters
    public static Collection<Integer[]> getParam() { // fornisce i parametri
        return Arrays.asList(new Integer[][] { // a, b, expected
            { 1, 1, 2 }, { 3, 2, 5 }, { 4, 3, 7 }, });
    }

    public SommatoreParamTest(int a, int b, int expected) { // inizializza parametri
        this.a = a;
        this.b = b;
        this.expected = expected;
    }

    @Test
    public void testSum() {
        Sommatore sommatore = new Sommatore();
        assertEquals(expected, sommatore.sum(a, b));
    }
}
```

Riferimenti

- J. B. Rainsberger: *JUnit Recipes. Practical Methods for Programmer Testing*
- T. Kaczanowski: *Practical Unit Testing with Junit and Mockito*
- https://martinfowler.com/articles/practical-test-pyramid.html
- https://dzone.com/articles/7-popular-unit-test-naming
- https://www.baeldung.com/mockito-series
- https://assertj.github.io/doc/
- https://martinfowler.com/articles/mocksArentStubs.html