Programmazione Orientata agli Oggetti

Code4Fun: Unit Testing.. i Mock

A cura di Roger Voyat

Sommario

- Un metodo difficile per lo Unit Test
- Unit Test classico: step
- i Mock
- il framework Mockito con i suoi costrutti
- Riepilogo Quando usare i Mock?

Un metodo difficile per lo Unit Test

```
public class DiaDia2 {
    private Partita partita;
    private IO io;
    private FabbricaDiComandi factory;

    /**
     * Processa una istruzione
     */
    public boolean processaIstruzione(String istruzione) {

        AbstractComando comandoDaEseguire = this.factory.costruisciComando(istruzione,getIo());
        boolean partitaFinita = comandoDaEseguire.esegui(getPartita());
        return partitaFinita;
    }
```

Come lo testo?

Classico

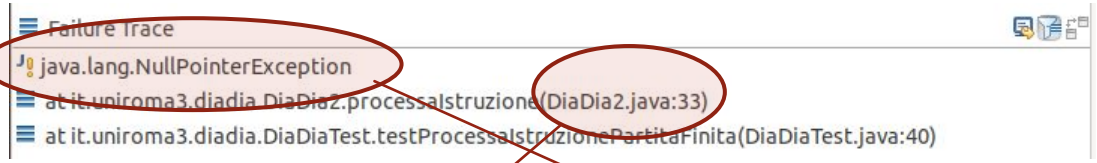
Step 1- Test vuoto

```java
/**
 * @author rvoyat
 *
 */
public class DiaDiaTest {

    private static final String ISTRUZIONE_FINE_PARTITA = "vai fine";
    private DiaDia2 diadia2;


    @Before
    public void setUp() {
        diadia2 = new DiaDia2();
    }

    @Test
    public void testProcessaIstruzionePartitaFinita() {

        /**/

        assertTrue(diadia2.processaIstruzione(ISTRUZIONE_FINE_PARTITA));
    }
```

Scriviamo il test vuoto con l'aspettativa e l'assert...

Il test fallisce.. ce lo aspettavamo già..

Step 1- Test vuoto



Scopriamo perché **fallisce**...

Sorgente

```
31    public boolean processaIstruzione(String istruzione) {
32
33        AbstractComando comandoDaEseguire = this.factory.costruisciComando(istruzione,getIo());
34        boolean partitaFinita = comandoDaEseguire.esegui(getPartita());
35        return partitaFinita;
36    }
```

Manca l'oggetto FabbricaDiComandi (null)...

Step 2 - Test refactor

```
public class DiaDiaTest {

    private static final String ISTRUZIONE_FINE_PARTITA = "vai fine";
    private DiaDia2 diadia2;
    private FabbricaDiComandiRiflessiva fabbricaDiComandi;


    @Before
    public void setUp() {
        diadia2 = new DiaDia2();
        fabbricaDiComandi = new FabbricaDiComandiRiflessiva();
    }

    @Test
    public void testProcessaIstruzionePartitaFinita() {

        diadia2.setFactory(fabbricaDiComandi);

        /**/

        assertTrue(diadia2.processaIstruzione(ISTRUZIONE_FINE_PARTITA));
    }
```

1. Istanziamo nel Test l'oggetto *FabbricaDiComandi*
2. Rilanciamo il Test….

Il test fallisce....

Step 2 - Test refactor

Scopriamo perché fallisce ancora...

```
31    public boolean processaIstruzione(String istruzione) {
32
33        AbstractComando comandoDaEseguire = this.factory.costruisciComando(istruzione,getIo());
34        boolean partitaFinita = comandoDaEseguire.esegui(getPartita());
35        return partitaFinita;
36    }
```

Mancano Partita e al suo interno Labirinto...

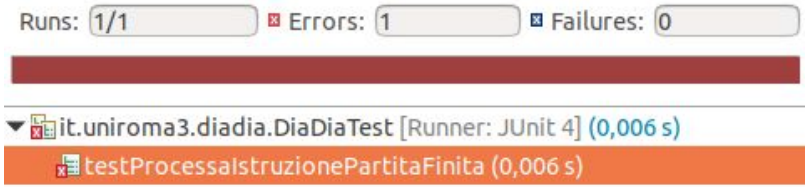
La faccenda inizia a complicarsi….

Step 3 - Test refactor (bis)

```java
public class DiaDiaTest {

    private static final String ISTRUZIONE_FINE_PARTITA = "vai fine";
    private DiaDia2 diadia2;
    private FabbricaDiComandiRiflessiva fabbricaDiComandi;
    private Partita partita;


    @Before
    public void setUp() {
        diadia2 = new DiaDia2();
        fabbricaDiComandi = new FabbricaDiComandiRiflessiva();
        partita = new Partita();
    }
```

```java
public class DiaDiaTest {

    private static final String ISTRUZIONE_FINE_PARTITA = "vai fine";
    private DiaDia2 diadia2;
    private FabbricaDiComandiRiflessiva fabbricaDiComandi;
    private Partita partita;


    @Before
    public void setUp() {
        diadia2 = new DiaDia2();
        fabbricaDiComandi = new FabbricaDiComandiRiflessiva();
        partita = new Partita(new Labirinto());
    }

    @Test
    public void testProcessaIstruzionePartitaFinita() {

        diadia2.setFactory(fabbricaDiComandi);
        diadia2.setPartita(partita);

        /**/

        assertTrue(diadia2.processaIstruzione(ISTRUZIONE_FINE_PARTITA));
    }
```

1. Istanziamo nel Test l'oggetto Partita che vuole un Labirinto …
2. Iniziamo a capire che ci sono troppe variabili da impostare che mancano ancora..
3. Proviamo a rilanciare il Test….



Il test fallisce....

Runs: 1/1 Errors: 1 Failures: 0

it.uniroma3.diadia.DiaDiaTest [Runner: JUnit 4] (6,513 s)

testProcessaIstruzionePartitaFinita (6,513 s)

Failure Trace

java.lang.NullPointerException

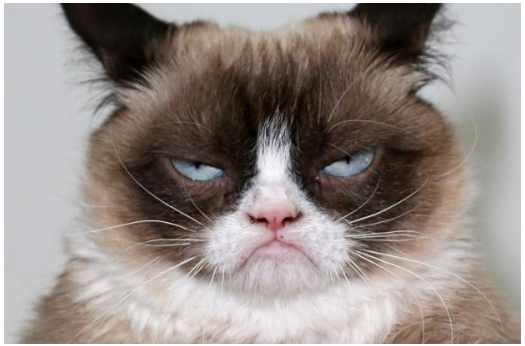
at it.uniroma3.diadia.comandi.ComandoVai.esegui(ComandoVai.java:36)

at it.uniroma3.diadia.DiaDia2.processaIstruzione(DiaDia2.java:34)

at it.uniroma3.diadia.DiaDiaTest.testProcessaIstruzionePartitaFinita(DiaDiaTest.java:50)

Sorgente

```
12 public class ComandoVai extends AbstractComando {
13
14     private final static String NOME = "vai";
15
16     public ComandoVai() {
17         super.setNome(NOME);
18     }
19
20     /**
21      * esecuzione del comando
22      */
23     @Override
24     public boolean esegui(Partita partita) {
25         Stanza stanzaCorrente = partita.getStanzaCorrente();
26         Stanza prossimaStanza = null;
27         if (super.getParametro() == null) {
28             super.getIO().mostraMessaggio("Dove vuoi andare? Devi specificare una direzione");
29             return partita.isFinita();
30         }
31         Direzione direzione;
32         try {
33             direzione = Direzione.valueOf(super.getParametro().toUpperCase());
34         } catch (IllegalArgumentException e) {
35             //caso in cui viene specificata una direzione non contemplata dall'enum Direzione
36             super.getIO().mostraMessaggio("Direzione inesistente");
37             return partita.isFinita();
38         }
```



Grrrrrrrrrr

Non so voi ma io ho gia perso la pazienza e la voglia di scrivere il test!

Mock

Perché abbiamo bisogno dei Mock.

- quando abbiamo applicazioni che dipendono da uno specifico ambiente di runtime

- quando nei metodi sotto test vi sono chiamate a oggetti e metodi *‘collaboratori’* :
  - per comportamento non deterministico
  - per difficoltà nella riproduzione o implementazione
  - se il comportamento del collaboratore può cambiare
  - per introdurre attributi e metodi solo ai fini del test

Principali Frameworks per il Mock in Java

1. EasyMock 
2. jMock
3. mockito 

Passo 1 (Classe): Identificare oggetti e metodi da ‘mockare’

```
public class DiaDia2 {
    private Partita partita;
    private IO io;
    private FabbricaDiComandi factory;

    /**
     * Processa una istruzione
     */
    public boolean processaIstruzione(String istruzione) {

        AbstractComando comandoDaEseguire = this.factory.costruisciComando(istruzione,getIo());
        boolean partitaFinita = comandoDaEseguire.esegui(getPartita());
        return partitaFinita;
    }
```

- Prima cosa per capire se ho bisogno dei Mock: verificando se ci sono classi/metodi **‘collaboratori’** richiamati all’interno del metodo da testare...

Caratteristiche principali dei Mock

- **Fake**: Con i Mock si può provare una parte dell'applicazione anche quando tutto il resto non è pronto

- **Expectation**: Verificare che la classe esterna che chiama questo mock abbia il comportamento corretto.

- **Developping pattern**:
  1. inizializza mock
  2. imposta aspettative
  3. esegui test
  4. verifica asserzioni.

Passo 2 (Test)

```java
public class DiaDia2Test {

    private static final String ISTRUZIONE_FINE_PARTITA = "vai fine";
    @Mock
    private FabbricaDiComandiRiflessiva mockFabbricaDiComandi;
    @Mock
    private Partita mockPartita;
    @Mock
    private IOSimulator io;
    @Mock
    private ComandoFine mockComandoFine;

    private DiaDia2 diadia2;


    @Before
    public void setUp() {
        diadia2 = new DiaDia2();
        MockitoAnnotations.initMocks(this);
    }

    @Test
    public void testProcessaIstruzionePartitaFinita() {

        Mockito.lenient().when(mockFabbricaDiComandi.costruisciComando(ISTRUZIONE_FINE_PARTITA, io))
            .thenReturn(mockComandoFine);
        Mockito.lenient().when(mockComandoFine.esegui(mockPartita))
            .thenReturn(true);

        diadia2.setFactory(mockFabbricaDiComandi);
        diadia2.setPartita(mockPartita);
        diadia2.setIo(io);

        assertTrue(diadia2.processaIstruzione(ISTRUZIONE_FINE_PARTITA));
    }
```

Dev. Pattern #1
Inizializza Mock

Dev. Pattern #2
Aspettative

Dev. Pattern #3
Esegui test

Dev. Pattern #4
Verifica Assert

I costrutti di ‘Mockito’

```
public class DiaDia2Test {

    private static final String ISTRUZIONE_FINE_PARTITA = "vai fine";
    @Mock
    private FabbricaDiComandiRiflessiva mockFabbricaDiComandi;
    @Mock
    private Partita mockPartita;
    @Mock
    private IOSimulator io;
    @Mock
    private ComandoFine mockComandoFine;

    private DiaDia2 diadia2;


    @Before
    public void setUp() {
        diadia2 = new DiaDia2();
        MockitoAnnotations.initMocks(this);
    }

    @Test
    public void testProcessaIstruzionePartitaFinita() {

        Mockito.lenient().when(mockFabbricaDiComandi.costruisciComando(IS
            .thenReturn(mockComandoFine);
        Mockito.lenient().when(mockComandoFine.esegui(mockPartita))
            .thenReturn(true);

        diadia2.setFactory(mockFabbricaDiComandi);
        diadia2.setPartita(mockPartita);
        diadia2.setIo(io);

        assertTrue(diadia2.processaIstruzione(ISTRUZIONE_FINE_PARTITA));
    }
```

mock() or **@Mock**: permettono la definizione di un oggetto mock a cui è poi possibile associare un comportamento ad esempio utilizzando dei metodi **when**();

initMocks (solo per Junit 4)

Mockito.lenient():(facoltativo) Pulisce i test e aiuta nel debug

when().thenReturn() : usato per definire l’output di un metodo quando viene chiamato

when().thenThrow()

Usato per testare se lanciata un eccezione

```
@Test(expected = NullPointerException.class)
public void testProcessaIstruzioneNullPointer() {
    Mockito.lenient(). when(mockFabbricaDiComandi.costruisciComando(ISTRUZIONE_FINE_PARTITA, io))
      .thenThrow(NullPointerException.class);

    diadia2.setFactory(mockFabbricaDiComandi);
    diadia2.setPartita(mockPartita);
    diadia2.setIo(io);

    diadia2.processaIstruzione(ISTRUZIONE_FINE_PARTITA);
}
```

Verify

Permette di verificare (testare) la corretta invocazione dei metodi.

```
@Test
void test() {
    List<String> mockList =  Mockito.mock(List.class);
    mockList.add("Pankaj");
    mockList.size();

    Mockito.verify(mockList).add("Pankaj");
    Mockito.verify(mockList,  Mockito.times(1)).size();
}
```

verify(mockList, times(1)).size(); //uguale al normale metodo di verifica
verify(mockList, atLeastOnce()).size(); // deve essere chiamato almeno una volta
verify(mockList, atMost(2)).size(); // deve essere chiamato al massimo 2 volte
verify(mockList, atLeast(3)).size(); // deve essere chiamato almeno 3 volte
verify(mockList, never()).clear(); // non deve mai essere chiamato

Spy

Usato per creare dei Wrapper di oggetti Java
Ogni chiamata, se non diversamente specificato, è delegata all'oggetto.

```
@Test
public void whenSpyingOnList_thenCorrect() {
    List<String> list = new LinkedList<String>();
    List<String> spyList = Mockito.spy(list);

    spyList.add("one");
    spyList.add("two");

    Mockito.verify(spyList).add("one");
    Mockito.verify(spyList).add("two");

    assertEquals(2, spyList.size());
}
```

il metodo ***add*** viene chiamato REALMENTE!

```
@Test
public void whenStubASpy_thenStubbed() {
    List<String> list = new LinkedList<String>();
    List<String> spyList = Mockito.spy(list);

    assertEquals(0, spyList.size());

    Mockito.doReturn(100).when(spyList).size();
    assertEquals(100, spyList.size());
}
```

per il metodo ***size()*** viene fatto uno Stub (fake)!

Mock vs Spy


```java
@Test
public void whenCreateMock_thenCreated() {
    List mockedList = Mockito.mock(ArrayList.class);

    mockedList.add("one");
    Mockito.verify(mockedList).add("one");

    assertEquals(0, mockedList.size());
}

@Test
public void whenCreateSpy_thenCreate() {
    List spyList = Mockito.spy(new ArrayList());

    spyList.add("one");
    Mockito.verify(spyList).add("one");

    assertEquals(1, spyList.size());
}
```

Con Mock nella realtà non viene effettivamente chiamato il metodo 'add'

Con Spy invece si...verrà chiamato effettivamente il metodo add e l'elemento verrà aggiunto all'elenco

Riepilogo:

Quando usare i mock

- per sostituire le classi/metodi con cui collaborano i metodi sotto test -> **Fine-grained isolation**
- per un comportamento ‘pilotato’ di tutti questi metodi/classi *fake*.

Altre caratteristiche dei Mock

- non implementano alcuna logica
- test mirati per un singolo metodo senza effetti collaterali
- risparmio di tempo ed energia nello scrivere il test.
- piccoli test sono facili da capire
- piccoli test sono facili da mantenere

Semplifica il refactoring dei test quando vi è una regressione !

Riferimenti Mockito :

Mockito: https://site.mockito.org/

Mockito Central Repository: https://search.maven.org/artifact/org.mockito/mockito-core

Mockito v. 3.3.3 usata pegli esempi: https://search.maven.org/artifact/org.mockito/mockito-core/3.3.3/jar

Seguire la guida ‘Come importare una libreria in Eclipse vista nella lezione di Lombok’