# TESTING E VERIFICA DEL SOFTWARE

# TUTORIAL MOCKITO

Ing. Piazzi Simona

## Esempio 1 – Lista della Spesa

Si supponga di avere una classe che rappresenta la lista della spesa con un metodo in grado di determinare il tempo medio necessario per fare la spesa.

```
 5 public class ListaSpesa {
 6     private List<String> articoli;
 7
 8     public int calcoloTempo() {
 9         int tempo;
10
11         if (articoli.size() > 30) {
12             tempo = 2;
13         } else {
14             tempo = 1;
15         }
16
17         return tempo;
18     }
19
20 }
```

**Test 1** Verificare che, per una lista di 35 articoli, il tempo necessario per fare la spesa è 2 ore.

Per soddisfare questa richiesta, la prima idea che viene in mente è quella di creare una lista di 35 articoli casuali che verrà utilizzata dalla funzione calcoloTempo() per verificare che questa restituisca un valore pari a 2.

Tuttavia, la creazione di questa lista ha un costo in termini di risorse e ha come unico valore quello di essere utilizzata all'interno della funzione.

Il focus del test deve invece essere spostato sulla verifica dell'output della funzione calcoloTempo() dove gli elementi utilizzati al suo interno devono essere visti come ausilio per mostrare il comportamento della funzione.

Lo scopo di Mockito è quindi quello di fornire la possibilità di mascherare il comportamento di un oggetto al fine di conferirgli il comportamento desiderato.

```
16    @Mock
17    List<String> articoli;
18
19    @InjectMocks
20    ListaSpesa mockListaSpesa;
21
22    @BeforeEach
23    public void setup() {
24        MockitoAnnotations.openMocks(this);
25    }
26
27    @Test
28    void test1() {
29        when(articoli.size()).thenReturn(35);
30        assertEquals(35, articoli.size());
31
32        assertEquals(2, mockListaSpesa.calcoloTempo());
33    }
```

L'annotazione @Mock ha lo scopo di creare un'istanza finta della variabile *articoli* che potrà successivamente essere manipolata per farle assumere il comportamento desiderato.

L'annotazione @InjectMocks viene utilizzata per inserire automaticamente la variabile *articoli* all'interno dell'oggetto da testare, ovvero *mockListaSpesa*. In linea generale, questa annotazione "inietta" automaticamente all'interno dell'oggetto da testare tutte le variabili che sono precedute dall'annotazione @Mock.

Prima di poter procedere con il test relativo alla funzione calcoloTempo(), è necessario abilitare l'uso delle notazioni di Mockito all'interno dei test come mostrato nella linea 24. Dato che questa operazione deve essere eseguita prima di ogni test, è stata aggiunta la notazione @BeforeEach.

Terminate queste operazioni preliminari, è possibile passare all'implementazione del test come mostrato nelle linee 29-32. La prima istruzione ha il compito di manipolare il comportamento della variabile *articoli* in modo che quando (when) viene chiamato il metodo size(), questo restituisca (thenReturn) un valore pari a 35.

In questo modo, ogni volta che il metodo size() verrà chiamato restituirà un valore pari a 35. Per verificare che il metodo calcoloTempo() restituisca il valore atteso di 2 ore, sarà sufficiente fare un confronto come mostrato nella linea 32.

**Test 2** Verificare che, per una lista di 5 articoli, il tempo necessario per fare la spesa è 1 ora.

Prova ad applicare quanto visto nel test 1: la soluzione è nel codice fornito.

## Esempio 2 – Distributore di caramelle

Si supponga di avere una classe che rappresenta un distributore di caramelle. Esso ha una capacità massima di 10 caramelle e può essere riempito con caramelle di gusti diversi. Allo stesso tempo è possibile prelevare una caramella per volta se il gusto selezionato è disponibile.

```
 5 public class DistributoreCaramelle {
 6     private List<String> caramelle;
 7     private static final int MAX_CARAMELLE = 10;
 8
 9     public void fill(int numeroCaramelle, String gusto) {
10         if (caramelle.size() + numeroCaramelle <= MAX_CARAMELLE) {
11             for (int i = 0; i < numeroCaramelle; i++) {
12                 caramelle.add(gusto);
13             }
14         }
15     }
16
17     public void getCandy(String gusto) {
18         if (caramelle.size() > 0 && caramelle.contains(gusto)) {
19             caramelle.remove(gusto);
20         }
21     }
22
23 }
```

**Test 1** Riempire il distributore con 2 caramelle al Limone e 3 alla Fragola e prelevarne una alla Fragola. Verificare che il distributore contiene 4 caramelle.

Come illustrato nell'esempio della lista della spesa, prima di eseguire il test è necessario impostare le notazioni di Mockito ed attivarle.

```
20     @Spy
21     List<String> caramelle = new ArrayList<>();
22
23     @InjectMocks
24     DistributoreCaramelle distributore;
25
26     @BeforeEach
27     public void setup() {
28         MockitoAnnotations.openMocks(this);
29     }
```

La notazione @Spy permette di attivare un meccanismo di tracciamento su un oggetto reale. In questo caso, si provvederà a tracciare quali metodi vengono chiamati dalla variabile *caramelle* all'interno della classe *DistributoreCaramelle*.

```
51    @Test
52    public void testGetCandy1() {
53        distributore.fill(2, "Limone");
54        distributore.fill(3, "Fragola");
55
56        verify(caramelle, times(5)).add(anyString());
57        verify(caramelle, times(3)).add("Fragola");
58
59        distributore.getCandy("Fragola");
60
61        verify(caramelle, times(1)).remove("Fragola");
62        verify(caramelle, times(0)).remove("Limone");
63        assertEquals(4, caramelle.size());
64    }
```

Dopo l'inserimento di 2 caramelle al limone e 3 alla fragola (linee 53-54), si procede a verificare (verify) se un determinato metodo è stato chiamato il numero di volte (times) atteso.

Ad esempio, la linea 56 verifica che il metodo add() con argomento una qualsiasi stringa (anyString()) è stato chiamato 5 volte. Mentre la linea 57 verifica che nello specifico siano state inserire 3 caramelle alla fragola.

Provando a rimuovere una caramella alla fragola il comportamento atteso sarà che venga rimossa una caramella alla fragola (linea 61) e nessuna caramella al limone (linea 62).

Al termine è possibile verificare che il distributore contiene esattamente 4 caramelle.

**Test 2** Verificare che il distributore non viene riempito se ha raggiunto la capacità massima.

```
41    @Test
42    public void testFill2() {
43        when(caramelle.size()).thenReturn(10);
44
45        distributore.fill(2, "Limone");
46
47        verify(caramelle, times(1)).size();
48        verify(caramelle, times(0)).add(anyString());
49    }
```

Nonostante la notazione @Spy, il comportamento della variabile *caramelle* può comunque essere manipolato. Come mostra la linea 43, è possibile impostare il valore restituito dal metodo size() con la capacità massima del distributore.

Provando un inserimento di 2 caramelle al limone, viene verificato che nessuna caramella viene aggiunta (linea 48).

Le soluzioni ai seguenti tre test aggiuntivi sono presenti nel codice:

**Test 3** Aggiungere due caramelle alla fragola e verificare che il primo elemento della variabile *caramelle* è una caramella alla fragola.

**Test 4** Aggiungere due caramelle al limone e verificare che la richiesta di prelievo di una caramella alla fragola non viene eseguita.

**Test 5** Verificare che non è possibile prelevare una caramella alla fragola se il distributore è vuoto.