APPUNTI JUNIT MOCKITO

Indice generale

JUNIT FRAMEWORK

IMPORTANZA DEGLI UNIT TEST

Partiamo da seguente metodo:

```
public class PrimoUnitTest {
    /* AACD => CD  CDAA => CDAA  ACDB => CDB */
    public String troncoLetteraANellePrimeDuePosizioni(String str){
        if (str.length() <= 2){
            return str.replaceAll("A", " ");
        }

        String primi2Caratteri = str.substring(0, 2);
        String stringaSenzaIPrimiDueCaratteri = str.substring(2);

        return primi2Caratteri.replaceAll("A","") + stringaSenzaIPrimiDueCaratteri;
    }
}
```

Quello che vediamo è un semplice metodo che tronca le prime 2 posizioni di una stringa, se esse contengono entrambe il carattere 'A'. Il problema sorge nel momento in cui passo al metodo una stringa del tipo "ACDB", dove abbiamo il carattere 'A' in una sola delle prime due posizioni. Tale metodo esegue una troncatura, anche se abbiamo un solo carattere 'A' nelle prime due posizioni, e ciò non è un comportamento atteso.

Poniamo il fatto di non sapere che questo metodo restituisce output inattesi, allora lo vogliamo testare. Il metodo tradizionale sarebbe quello di dichiarare un metodo main(), dove richiamare tale metodo e vedere che output restituisce. L'altro metodo è quello di utilizzare gli unit test.

Gli unit test permettono di testare unità specifiche di codice, dove un'unità può corrispondere ad un singolo metodo, un gruppo di metodi o un insieme di classi. In genere, nella maggior parte degli scenari, quando si parla di unit test, si parla di testare un metodo o un gruppo specifico di metodi.

Il vantaggio dello unit test è che, una volta scritto per un particolare metodo, saremo in grado di eseguirlo automaticamente in pochi millisecondi e di verificare se tutte le condizioni per quel metodo, che decreteranno il successo o il fallimento di tale test, corrispondono o meno.

SCRITTURA DEL PRIMO UNIT TEST

La prima cosa importante sugli unit test è che non devono essere mescolati con il codice sorgente dove è presente il metodo da testare. Inoltre, quando vado a creare la classe che contiene uno o più unit test, la vado ad inserire in una directory diversa da quella in cui sono presenti tutte le altre classi che verranno eseguite.

Quindi, per convenzione, tutti gli unit test vanno inseriti in una directory a loro dedicata, rinominata come "test".

Altre 2 convenzioni da seguire sono le seguenti:

- il nome della classe creata per gli unit test deve avere lo stesso nome della classe da cui essa deriva, seguito dal suffisso "Test";

- all'interno della directory test, la classe che contiene gli unit test si deve trovare nello stesso percorso di directory della classe da cui deriva nella cartella src.

Vediamo un esempio:

```
import org.junit.jupiter.api.Test;

import static org.junit.jupiter.api.Assertions.*;

class PrimoUnitTestTest{

    @Test
    public void testTroncoLetteraANellePrimeDuePosizioni() {

    }
}
```

Questo è il risultato che otteniamo nel momento in cui generiamo una classe per i test.

Se proviamo ad eseguirlo, otteniamo fin da subito che tale test ha avuto successo, ma in realtà non sta testando nulla. Questo avviene perché, se non ci sono fallimenti da segnalare, allora il test ha automaticamente successo. Per fallimento si intende una qualsiasi aspettativa che non viene soddisfatta e, nel caso dell'esempio, abbiamo uno unit test senza alcuna aspettativa, per cui ha successo.

La prima cosa che si può notare nell'esempio è l'annotazione @Test. Tale annotazione è definita nel pacchetto org.junit.jupiter.api.Test. Ogni annotazione @Test presente nella classe sta ad indicare uno unit test, in cui viene definito sempre un metodo su cui dichiarare i test da eseguire.

Nel caso specifico dell'esempio, vogliamo verificare che la stringa che inseriamo nello unit test (valore atteso) sia uguale alla stringa che ci viene restituita dal metodo che stiamo testando (valore attuale). Per fare ciò richiamiamo il metodo assertEquals(), che confronta l'output atteso con l'output attuale, nel seguente modo:

```
import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.*;
class PrimoUnitTestTest{

    @Test
    public void testTroncoLetteraANellePrimeDuePosizioni() {

        PrimoUnitTest primoUnitTest = new PrimoUnitTest();

        /*assertEquals( valore atteso, valore attuale )*/
assertEquals("CD",primoUnitTest.troncoLetteraANellePrimeDuePosizioni("AACD"));
assertEquals("CDAA",primoUnitTest.troncoLetteraANellePrimeDuePosizioni("CDAA"));
assertEquals("ACDB",primoUnitTest.troncoLetteraANellePrimeDuePosizioni("ACDB"));
    }
}
```

Come si può notare dall'esempio, abbiamo provato ad inserire più output diversi per il metodo da testare. Il risultato è che l'ultimo dei 3 test è fallito, perché l'output che ci aspettiamo (ACDB) è diverso dall'output che ci restituisce il metodo testato (CDB). Questo ci è stato di aiuto per capire che andranno fatte delle modifiche sul codice del metodo testato e, una volta eseguite, potremo comunque continuare a testare tale metodo, utilizzando lo stesso unit test scritto in precedenza. Quando lo unit test avrà successo per tutti i casi di output, lì avremo la conferma che il metodo sarà finalmente scritto correttamente a livello di logica.

REFACTORING DELLO UNIT TEST

Partiamo dall'unit test scritto in precedenza:

```
import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.*;
class PrimoUnitTestTest{

    @Test
    public void testTroncoLetteraANellePrimeDuePosizioni() {

        PrimoUnitTest primoUnitTest = new PrimoUnitTest();

        /*assertEquals( valore atteso, valore attuale )*/
assertEquals("CD",primoUnitTest.troncoLetteraANellePrimeDuePosizioni("AACD"));
assertEquals("CDAA",primoUnitTest.troncoLetteraANellePrimeDuePosizioni("CDAA"));
assertEquals("ACDB",primoUnitTest.troncoLetteraANellePrimeDuePosizioni("ACDB"));
    }
}
```

Dobbiamo fare in modo che il codice dello unit test sia della qualità migliore possibile, senza alterarne il comportamento, mediante il refactoring.

Partendo dal nome del metodo che definisce lo unit test, è sempre bene dare un nome che chiarisca cosa si sta testando, con l'aggiunta della condizione che lo unit test deve rispettare:

```
@Test
    public void testTroncoLetteraANellePrimeDuePosizioni_Condizione() {
    …
    }
```

In più, tutti i metodi che definiscono gli unit test devono essere public void e devono essere sempre preceduti dall'annotazione @Test.

Un'altra cosa fatta nell'esempio, ma che è convenzionalmente sbagliata, è che stiamo eseguendo 3 test diversi nello stesso unit test. Idealmente, è necessario creare uno unit test per ogni test:

```
import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.*;

class PrimoUnitTestTest{

    @Test
    public void testTroncoLetteraANellePrimeDuePosizioni_TroncaturaDaEseguireStringaAACD() {

        PrimoUnitTest primoUnitTest = new PrimoUnitTest();

        /*assertEquals( valore atteso, valore attuale )*/
        assertEquals("CD",primoUnitTest.troncoLetteraANellePrimeDuePosizioni("AACD"));

    }

    @Test
    public void testTroncoLetteraANellePrimeDuePosizioni_TroncaturaDaNonEseguireStringaCDAA() {

        PrimoUnitTest primoUnitTest = new PrimoUnitTest();
```

```
assertEquals("CDAA",primoUnitTest.troncoLetteraANellePrimeDuePosizioni("CDAA"));

    }

    @Test
    public void testTroncoLetteraANellePrimeDuePosizioni_TroncaturaDaNonEseguireStringaACDB() {

        PrimoUnitTest primoUnitTest = new PrimoUnitTest();


assertEquals("ACDB",primoUnitTest.troncoLetteraANellePrimeDuePosizioni("ACDB"));

    }
}
```

Il vantaggio nel creare più unit test sta nel fatto che, quando uno di questi fallisce, sappiamo esattamente qual è nello specifico.

L'ultima cosa che si può notare è la duplicazione del codice riguardante la dichiarazione dell'istanza di PrimoUnitTest. Per evitare ciò, basta rendere la variabile della dichiarazione di classe:

```
import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.*;

class PrimoUnitTestTest{

    PrimoUnitTest primoUnitTest = new PrimoUnitTest();

    @Test
    public void testTroncoLetteraANellePrimeDuePosizioni_TroncaturaDaEseguireStringaAACD() {

        /*assertEquals( valore atteso, valore attuale )*/
        assertEquals("CD",primoUnitTest.troncoLetteraANellePrimeDuePosizioni("AACD"));

    }

    @Test
    public void testTroncoLetteraANellePrimeDuePosizioni_TroncaturaDaNonEseguireStringaCDAA() {

assertEquals("CDAA",primoUnitTest.troncoLetteraANellePrimeDuePosizioni("CDAA"));

    }

    @Test
    public void testTroncoLetteraANellePrimeDuePosizioni_TroncaturaDaNonEseguireStringaACDB() {

assertEquals("ACDB",primoUnitTest.troncoLetteraANellePrimeDuePosizioni("ACDB"));

    }
}
```

METODI assertTrue() E assertFalse()

Partiamo da questo metodo:

```
public class SecondoUnitClass {

    /*ABCD => false
    * ABAB => true
    * AB => true, perchè avendo la stringa solo due caratteri,
    * essi sono considerati sia come primi 2 che ultimi 2,
    * per cui è come se i primi 2 caratteri fossero uguali agli
    * ultimi 2
    * A => false, perchè la stringa ha meno di 2 caratteri*/

    public boolean verificoCheIPrimiEGliUltimiDueCaratteriSonoGliStessi(String str){

        if (str.length() <= 1) { return false; }
        if (str.length() == 2) { return true; }

        String primi2Caratteri = str.substring(0, 2);

        String ultimi2Caratteri = str.substring(str.length() - 2);

        return primi2Caratteri.equals(ultimi2Caratteri);
    }
}
```

Da come si può capire dal nome del metodo, il risultato sarà true se, data una stringa, i primi due caratteri sono uguali agli ultimi 2, altrimenti false.

Se devo testare metodi che restituiscono un valore booleano come risultato, Junit mi mette a disposizione i due metodi assertFalse() e assertTrue() che, rispettivamente, verificano se il metodo da testare restituisce false o true come output:

```
import org.junit.jupiter.api.Test;

import static org.junit.jupiter.api.Assertions.*;

class SecondoUnitClassTest {

    SecondoUnitClass secondoUnitClass = new SecondoUnitClass();

    @Test
    public void
testVerificoCheIPrimiEGliUltimiDueCaratteriSonoGliStessi_Primi2CaratteriEUltimi2Caratte
riSonoDiversi() {

assertFalse(secondoUnitClass.verificoCheIPrimiEGliUltimiDueCaratteriSonoGliStessi("ABCD
"));
    }

    @Test
    public void
testVerificoCheIPrimiEGliUltimiDueCaratteriSonoGliStessi_Primi2CaratteriEUltimi2Caratte
riSonoUguali() {

assertTrue(secondoUnitClass.verificoCheIPrimiEGliUltimiDueCaratteriSonoGliStessi("ABAB"
));
```

```
    }

    @Test
    public void
testVerificoCheIPrimiEGliUltimiDueCaratteriSonoGliStessi_Primi2CaratteriSonoAncheGliUlt
imi2() {

assertTrue(secondoUnitClass.verificoCheIPrimiEGliUltimiDueCaratteriSonoGliStessi("AB"))
;
    }

    @Test
    public void
testverificoCheIPrimiEGliUltimiDueCaratteriSonoGliStessi_UnSoloCarattere() {

assertFalse(secondoUnitClass.verificoCheIPrimiEGliUltimiDueCaratteriSonoGliStessi("A"))
;
    }
}
```

@Before E @After

Con l'annotazione @Before stiamo indicando che il metodo, il quale è assegnato, deve essere eseguito prima di ogni altro unit test, ripetendo la sua esecuzione tante volte per quanti sono i test presenti nella classe.

Con l'annotazione @After stiamo indicando che il metodo, il quale è assegnato, deve essere eseguito dopo ogni altro unit test, ripetendo la sua esecuzione tante volte per quanti sono i test presenti nella classe.

Vediamo un semplice esempio con @Before:

```
import org.junit.*;

public class BeforeAfterTest {

    @Before
    public void testBefore(){
        System.out.println("Before Test");
    }

    @Test
    public void test1(){
        System.out.println("Test 1 eseguito");
    }

    @Test
    public void test2(){
        System.out.println("Test 2 eseguito");
    }

    @After
    public void testAfter(){
        System.out.println("After Test");
    }
}
```

L'output di questo esempio è il seguente:

Before Test
Test 1 eseguito
After Test
Before Test
Test 2 eseguito
After Test

@BeforeClass E @AfterClass

Abbiamo esaminato le annotazioni @Before e @After, che rispettivamente eseguono il metodo, i quali sono assegnati, prima e dopo ogni test presente nella classe, ripetendo la sua esecuzione tante volte per quanti sono i test presenti nella classe.

Se, però, volessi eseguire tale metodo prima o dopo tutti i test presenti nella classe, eseguendolo una sola volta, allora posso assegnargli rispettivamente le annotazioni @BeforeClass o @AfterClass.

Il metodo con annotazione @BeforeClass o @AfterClass, oltre che essere public void, deve essere sempre statico.

Vediamo un esempio:

```
import org.junit.*;

public class BeforeAfterTest {

    @BeforeClass
    public static void testBeforeClass(){
        System.out.println("Before Class Test");
    }

    @Before
    public void testBefore(){
        System.out.println("Before Test");
    }

    @Test
    public void test1(){
        System.out.println("Test 1 eseguito");
    }

    @Test
    public void test2(){
        System.out.println("Test 2 eseguito");
    }

    @After
    public void testAfter(){
        System.out.println("After Test");
    }

    @AfterClass
    public static void testAfterClass(){
        System.out.println("After Class Test");
```

```
    }
}
```

L'output di questo codice è il seguente:

Before Class Test
Before Test
Test 1 eseguito
After Test
Before Test
Test 2 eseguito
After Test
After Class Test

COMPARARE ARRAY IN JUNIT

Per comparare due array in uno unit test, si deve utilizzare un altro metodo della classe Assert denominato assertArrayEquals().

Vediamo un esempio in cui testiamo l'ordinamento di un array:

```
import org.junit.Test;
import java.util.Arrays;
import static org.junit.jupiter.api.Assertions.*;

public class ArrayTest {

    @Test
    public void testOrdinamentoArray_arrayRandomInIngresso(){

        int[] numbers = {12, 3, 4, 1};
        int[] aspettativa = {1, 3, 4, 12};

        Arrays.sort(numbers);

        assertArrayEquals(aspettativa, numbers);
    }
}
```

TESTARE ECCEZIONI IN JUNIT

Quando si ha a che fare con le eccezioni negli unit test, non sono elementi da considerare necessariamente come condizione di errore. Quindi, possiamo in qualche modo far considerare l'eccezione allo unit test come conseguenza prevista.

Per fare quanto detto, non bisogna fare altro che catturare l'eccezione all'interno dello unit test, come nell'esempio:

```
@Test
public void testOrdinamentoArray_arrayInizializzatoANull(){

    int[] numbers = null;
    try {
        Arrays.sort(numbers);
    } catch (NullPointerException e) {
        //Success
    }
}
```

Junit ci mette comunque a disposizione una maniera più pulita di scrivere l'esempio precedente, ma che funziona allo stesso modo:

```
@Test(expected = NullPointerException.class)
public void testOrdinamentoArray_arrayInizializzatoANull(){

    int[] numbers = null;
    Arrays.sort(numbers);
}
```

Da notare bene il fatto che, se dichiaro in uno unit test il verificarsi di un'eccezione ed essa non si verifica, il test fallisce. Allo stesso modo, se non dichiaro alcuna eccezione in uno unit test ed essa si verifica, il test fallisce.

Tornando all'esempio precedente, lasciando il codice così come sta, il test ha successo. Ma se dovessi togliere dalla dichiarazione dell'array numbers l'inizializzazione a null, allora il test fallisce.

TESTARE LE PERFORMANCE IN JUNIT

Junit è uno strumento che è anche in grado di testare le prestazioni dei metodi. Per fare ciò è possibile impostare un timeout per specificare in quanti millisecondi deve essere eseguita un'operazione. Se il tempo di esecuzione di tale operazione supera il timeout impostato, il test fallisce, altrimenti ha successo.

Questo è un ottimo test da avere quando si desidera imporre un requisito di prestazione molto rigido su determinate operazioni e quando si desidera essere avvisati sui tempi di prestazione di modifiche effettuate da altre persone.

Vediamo un esempio in cui tentiamo di ordinare un array per un milione di volte prima dello scadere del timeout:

```
@Test(timeout = 100)
public void testPrestazioniOrdinamento_arrayRandomInIngresso(){

    int array[] = {12, 23, 4};
    for(int i = 1; i <= 1000000; i++) {
        array[0] = i;
        Arrays.sort(array);
    }
}
```

TEST PARAMETRIZZATI

Riprendiamo l'esempio della classe PrimoUnitTestTest:

```
import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.*;

class PrimoUnitTestTest{

    PrimoUnitTest primoUnitTest = new PrimoUnitTest();

    @Test
    public void testTroncoLetteraANellePrimeDuePosizioni_TroncaturaDaEseguireStringaAACD() {

        /*assertEquals( valore atteso, valore attuale )*/
        assertEquals("CD",primoUnitTest.troncoLetteraANellePrimeDuePosizioni("AACD"));
    }

    @Test
    public void testTroncoLetteraANellePrimeDuePosizioni_TroncaturaDaNonEseguireStringaCDAA() {

assertEquals("CDAA",primoUnitTest.troncoLetteraANellePrimeDuePosizioni("CDAA"));
    }

    @Test
    public void testTroncoLetteraANellePrimeDuePosizioni_TroncaturaDaNonEseguireStringaACDB() {

assertEquals("ACDB",primoUnitTest.troncoLetteraANellePrimeDuePosizioni("ACDB"));

    }
}
```

Tutti gli unit test nell'esempio hanno lo stesso codice, tranne per il fatto che sto inserendo per ognuno di essi un input diverso e mi aspetto un output specifico. Per evitare questa situazione, entrano in gioco i test parametrizzati. I test parametrizzati permettono di rendere gli input che passiamo agli unit test e gli output che ci aspettiamo dei parametri. Tali parametri vengono inseriti in un array, che verrà passato allo unit test e che verrà eseguito più volte, senza dover essere riscritto.

La classe in cui avremo il test parametrizzato deve avere prima della sua definizione la seguente istruzione:

```
@RunWith(Parameterized.class)
```

dopodichè si deve dichiarare un metodo public static che restituisce una Collection.

Vediamo un esempio di test parametrizzato:

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

import org.junit.Test;
import org.junit.runner.RunWith;
import org.junit.runners.Parameterized;

import java.util.Arrays;
import java.util.Collection;

/*Istruzione che indica che questa classe ha al suo interno un test parametrizzato*/
@RunWith(Parameterized.class)
public class TestParametrizzati {

    /*Variabili da passare al costruttore della classe TestParametrizzati
    che dobbiamo generare*/
    private String inputTestato;
    private String outputAtteso;

    /*Definiamo il costruttore della classe TestParametrizzati*/
    public TestParametrizzati(String outputAttuale, String outputAtteso) {
        this.inputTestato = outputAttuale;
        this.outputAtteso = outputAtteso;
    }

    /*Definiamo la raccolta di parametri*/
    @Parameterized.Parameters
    public static Collection<Object[]> condizioniTest() {
        return Arrays.asList(new Object[][] {
            /* { input testato, output atteso } */
                {"AACD", "CD"},
                {"CDAA", "CDAA"},
                {"ACDB", "ACDB"}
        });
    }

    @Test
    public void testTroncoLetteraANellePrimeDuePosizioni_Parametri() {
        PrimoUnitTest primoUnitTest = new PrimoUnitTest();

        System.out.println("Eseguo il test con il numero " + inputTestato);

        assertEquals(outputAtteso,
primoUnitTest.troncoLetteraANellePrimeDuePosizioni(inputTestato));
    }}
```

ORGANIZZARE GLI UNIT TEST IN TEST SUITE

Junit ci permette anche di inserire classi di test in gruppi specifici, chiamati test suite.

Quando definiamo un test suite, possiamo eseguire tutti gli unit test, definiti all'interno delle classi di test raggruppate nel test suite, in maniera simultanea.

Vediamo un esempio di creazione di test suite:

```
import org.junit.runner.RunWith;
import org.junit.runners.Suite;


@RunWith(Suite.class)
@Suite.SuiteClasses({
        ArrayTest.class,
        BeforeAfterTest.class
})

public class TestSuite {
}
```

PERCHÈ UTILIZZARE MOCKITO

IMPOSTAZIONE DEL SET (SYSTEM UNDER TEST)

La prima cosa da fare per capire il perché abbiamo bisogno di Mockito è quella di iniziare a creare un semplice esempio. L'esempio consiste nel creare una parte specifica di una applicazione manageriale.

Questo passaggio non è ancora correlato direttamente a Mockito, ma è importante per comprenderne l'utilizzo:

```
import org.junit.Test;

import static org.junit.Assert.*;

//ServizioTatuaggio.java
//public List<String> recuperoTatuaggi(String utente);

//TatuaggioBusinessImpl.java
//public List<String> recuperoTatuaggiRelativiAllaPrimavera(String utente)

public class PrimoMockitoTest {

    @Test
    public void test() {
        assertTrue( true);
    }
}
```

ServizioTatuaggio.java è un'interfaccia che serve per la nostra applicazione manageriale e definisce un metodo che restituirà una lista di tatuaggi che possiede ogni utente. La cosa importante è che tale interfaccia rappresenta un servizio esterno per la nostra applicazione, ma che non si porta dietro nessuna implementazione.

L'implementazione la dobbiamo creare noi e servirà per creare una sorta di filtro su questi tatuaggi. In particolare, dobbiamo riuscire a vedere quali tatuaggi di quell'utente sono legati alla primavera. Quindi supponiamo, per esempio, che un utente abbia cinque tatuaggi, ma solo due di essi siano relativi alla primavera. Quello che vogliamo fare è dare una funzionalità all'utente che gli permetta di filtrare i suoi tatuaggi in base ad un argomento specifico.

Il metodo recuperoTatuaggiRelativiAllaPrimavera() chiamerà al suo interno il metodo recuperoTatuaggi() dell'interfaccia, in modo da avere tutta la lista di tatuaggi, e filtrerà nello specifico i tatuaggi che sono legati alla primavera.

Vediamo la loro implementazione:

ServizioTatuaggio.java

```
import java.util.List;

public interface ServizioTatuaggio {

    public List<String> recuperoTatuaggi(String utente);
}
```

TatuaggioBusinessImpl.java

```
import java.util.ArrayList;
import java.util.List;

public class TatuaggioBusinessImpl {

    private ServizioTatuaggio servizioTatuaggio;

    /* Costruttore della classe TatuaggioBusinessImpl*/
    public TatuaggioBusinessImpl(ServizioTatuaggio servizioTatuaggio) {
        this.servizioTatuaggio = servizioTatuaggio;
    }

    public List<String> recuperoTatuaggiRelativiAllaPrimavera(String utente){

        List<String> tatuaggiFiltrati = new ArrayList<String>();
        List<String> tatuaggi = servizioTatuaggio.recuperoTatuaggi(utente);

        for(String tatuaggio : tatuaggi){
            if(tatuaggio.contains("Primavera")){
                tatuaggiFiltrati.add(tatuaggio);
            }
        }

        return tatuaggiFiltrati;
    }

}
```

Dopo aver implementato recuperoTatuaggiRelativiAllaPrimavera(), quello che vogliamo fare è scrivere un test per questo metodo specifico. Ma tale metodo dipende dall'interfaccia ServizioTatuaggi che è un servizio esterno sviluppato da qualche altro team. Essendo un servizio esterno, non ne possediamo l'implementazione, quindi sembra apparentemente impossibile poter scrivere un test al momento.

Quello che stiamo facendo finora è porre delle basi per capire il perché abbiamo bisogno di Mockito e che tipo di problemi risolve.

ESEMPIO DI STUBBING – CON GLI SVANTAGGI CHE SI PORTA DIETRO

Quindi quello che stiamo tentando di fare è quello di testare un servizio che dipende da un altro esterno, che è stato implementato da un altro team e di cui non ne abbiamo l'accesso all'implementazione.

Una delle cose che possiamo fare in questi casi è quella ci creare uno stub. Uno stub non è altro che un'implementazione di esempio di questo particolare servizio esterno, da eseguire in modo da poter eseguire dei test. Detto in altre parole, lo stub è una classe che restituisce una sorta di dati fittizi.

Per cui, quello che facciamo ora è creare uno stub per l'interfaccia ServizioTatuaggio e proveremo a utilizzare tale stub per testarlo.

Detto ciò, di seguito abbiamo i 2 passaggi che seguiremo per continuare la creazione della nostra applicazione manageriale:

- creare TodoServiceStub;
- test di TatuaggioBusinessImpl utilizzando TodoServiceStub.

Vediamo il tutto scritto in codice:

ServizioTatuaggioStub.java

```
import java.util.Arrays;
import java.util.List;

public class ServizioTatuaggioStub implements ServizioTatuaggio {

    /* Implementazione di esempio del servizio esterno ServizioTatuaggio */
    @Override
    public List<String> recuperoTatuaggi(String utente) {
        return Arrays.asList("Imparare di primavera in paese",
                             "Maledetta primavera",
                             "Il mare d'estate");
    }
}
```

TatuaggioBusinessImplStubTest.java

```
import org.junit.Test;
import static org.junit.Assert.*;
import java.util.List;

public class TatuaggioBusinessImplStubTest {

    @Test
    public void testRecuperoTatuaggiRelativiAllaPrimavera_utilizzoStub(){

        ServizioTatuaggio servizioTatuaggio = new ServizioTatuaggioStub();
        TatuaggioBusinessImpl tatuaggioBusinessImpl = new
TatuaggioBusinessImpl(servizioTatuaggio);

        List<String> tatuaggiFiltrati =
tatuaggioBusinessImpl.recuperoTatuaggiRelativiAllaPrimavera("Paolo");

        assertEquals(2, tatuaggiFiltrati.size());
    }

}
```

Nell'esempio abbiamo creato uno stub, che restituisce valori fittizi, e usiamo tali valori per verificare che il metodo RecuperoTatuaggiRelativiAllaPrimavera() funzioni o meno.

Ora vediamo però quali sono gli svantaggi nell'utilizzo di uno stub per i test:

- Quando vogliamo che lo stub deve restituire diversi tipi di valori in diversi tipi di scenari, allora diventa un codice molto complesso, molto grande e difficile da manutenere;
- Lo stub deve implementare tutti i metodi del servizio che definisce.

Gli stub, quindi, sono molto utili in scenari molto semplici, ma nella maggior parte dei progetti si utilizza qualcosa di più dinamico, chiamato mock.