

# Introduzione al Testing

Dott. Giovanni Francesco Lucchese  
giovannifrancesco.lucchese@euris.it

Gruppo Euris S.p.A.

22/02/2023



# Programma del corso

- ▶ Teoria del testing
- ▶ Software Testing
- ▶ Test di componente
- ▶ Junit
  - ▶ Best practices
  - ▶ Dati di test
- ▶ Test Doubles
  - ▶ Mockito
  - ▶ Best Practices
- ▶ Cenni di teoria
  - ▶ test di Integrazione
  - ▶ test di Sistema
  - ▶ Cenni sui test di Accettazione

# Teoria del Testing



# Terminologia

- ▶ **Errore** (error, mistake): L'azione sbagliata dello sviluppatore causata da una svista o dalla comprensione errata dei requisiti.
- ▶ **Difetto** (defect, fault or bug): Il risultato dell'errore. Il difetto può essere introdotto sia all'interno del codice che dei requisiti.
- ▶ **Guasto** (failure): L'incapacità del software o un suo componente di eseguire le funzionalità richieste dalle specifiche.

# Prevenzione degli errori

La prevenzione degli errori viene effettuata prima che il sistema venga rilasciato o messo in un funzione.

I principali metodi di prevenzione si basano sull'utilizzare una buona metodologia di programmazione per ridurre la **complessità del codice** (direttive di stile, clean code, versionamento etc...) e, nel caso dei requisiti, utilizzare metodologie di **pianificazione del lavoro** (metodi agili, stesura di story, BDD).

# Individuazione dei difetti

L'individuazione è un tipo di gestione che viene effettuata durante l'esecuzione del sistema e mira a scoprire i guasti.

I metodi principali sono la creazione mirata di un guasto tramite il **testing**, la ricerca di un guasto con **debugging** e il **monitoraggio** dello stato del sistema, che viene effettuato tramite il recupero di specifiche metriche.

# “Recupero” dei guasti

Le azioni di recupero (o recovery) servono a irrobustire il sistema creando delle **reti di sicurezza** che arginano gli effetti causati da un difetto.

Grazie alle azioni di recupero, nonostante il difetto venga attivato dal flusso di esecuzione, il guasto non si verifica e viene gestito.

Uno strumento di recupero ampiamente usato sono le istruzioni «try... catch».

# Software Testing





# Software testing

Il software testing è una pratica che mira **riprodurre dei guasti** in un contesto controllato al fine di valutare la qualità del software e ridurre il rischio che il software riscontri dei guasti durante il suo funzionamento.

# Testing vs Debugging

- ▶ Testare serve a identificare un difetto causato da un guasto nel software.
- ▶ L'azione di debug è un'attività svolta al fine di trovare, analizzare e sistemare un difetto causato da un guasto identificato.

# I 7 principi del testing

- ▶ Il testing mostra la presenza di difetti, non la loro assenza.
- ▶ Il testing esaustivo è impossibile.
- ▶ Il testing anticipato salva tempo e denaro.
- ▶ I difetti si raggruppano.
- ▶ Il paradosso del pesticida.
- ▶ Il testing è dipendente dal contesto.
- ▶ L'assenza di difetti è una fallacia.

# Terminologia

- ▶ **Caso di test:** Un insieme di precondizioni, input, azioni e postcondizioni atte a valutare il sistema sotto test.
- ▶ **Sistema sotto test (System Under Test o SUT):** Il sistema che viene validato dal testing.
- ▶ **Precondizione:** Lo stato in cui si deve trovare il SUT e l'ambiente di test prima dell'esecuzione del test.
- ▶ **Postcondizione:** Lo stato atteso in cui il SUT e l'ambiente di test si troveranno alla fine dell'esecuzione.

# Esercizio 1a

Nel metodo in figura, identificare:

- ▶ precondizioni
- ▶ postcondizioni
- ▶ SUT

```
5  public int add(int a, int b) {  
6      return a + b;  
7  }  
8
```

# Livelli di test

- ▶ Test di componente
- ▶ Test di integrazione
- ▶ Test di sistema
- ▶ Test di accettazione

# Test di Componente



# Test di componente

Il test di componente è solitamente effettuato in **isolamento** dal resto del sistema e dipendentemente dall'architettura software del sistema sotto test l'isolamento è ottenuto tramite l'utilizzo di test doubles.

Il test di componente può valutare i requisiti funzionali (e.g. correttezza di un calcolo), requisiti non funzionali (e.g. test di performance) e proprietà strutturali (e.g. test di decisione).



# Test di componente

Nei test di componente i tipici difetti e guasti si possono raggruppare in tre categorie:

- ▶ Funzionalità incorrette
- ▶ Problemi di flusso di dati
- ▶ Codice e logiche non corrette

# Test di componente - Test di unità

Questo tipo di test viene effettuato per ogni componente del sistema in un ambiente artificiale in cui il SUT viene eseguito in isolamento.

La necessità di avere un ambiente isolato nasce dall'esigenza di testare una parte di codice senza influenze da dipendenze esterne.

# Esercizio 1b

- ▶ Relativamente a precondizioni, SUT e postcondizioni determinati nell'esercizio 1a, scrivere un test di unità utilizzando il metodo *main()* per eseguire il test.

The background features abstract, overlapping geometric shapes in various shades of blue, ranging from light sky blue to deep navy blue. These shapes are primarily located on the left and right sides of the frame, leaving a large white central area.

# Junit



# Junit

Junit è un framework di test per Java che predispone una collezione di istruzioni e pacchetti per facilitare la stesura dei test e la loro esecuzione.

Questo framework è specializzato nell'eseguire test di componente ma può essere utilizzato anche per test di integrazione.

# Junit – Istruzioni principali

```
1  package it.euris.ires;
2
3  import static org.junit.jupiter.api.Assertions.assertEquals;
4
5  import org.junit.jupiter.api.DisplayName;
6  import org.junit.jupiter.api.Test;
7
8  class CalculatorTests {
9
10     @Test
11     @DisplayName("1 + 1 = 2")
12     void addsTwoNumbers() {
13         Calculator calculator = new Calculator();
14         assertEquals(expected: 2, calculator.add(a: 1, b: 1), message: "1 + 1 should equal 2");
15     }
16
17 }
18
```

# Junit - Classpath dei test

I test vengono inseriti all'interno della cartella  
`src/test/<package>`

In particolare se testiamo la classe `Calculator.class`, il relativo test dovrà trovarsi all'interno dello stesso package della classe e, convenzionalmente, il nome della classe di test sarà  
`<nomeClasseTestata>Test.class`.

In particolare, la classe di test per il file `Calculator.class` si chiamerà `CalculatorTest.class`

# Junit - Asserzioni

Un'asserzione è un predicato che indica il predicato stesso sempre vero. Le asserzioni vengono utilizzate per verificare le post-condizioni di un caso di test.

Junit predispone una libreria di asserzioni che permettono di effettuare differenti valutazioni; le più utilizzate sono:

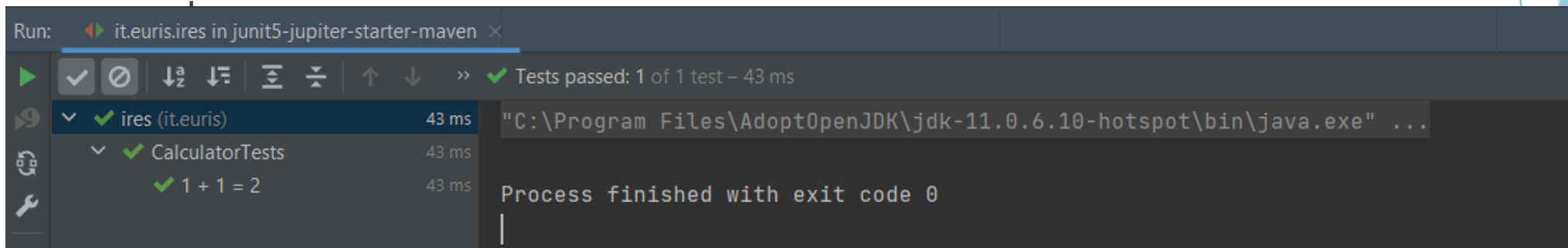
- ▶ `assertEquals(<T>, <T>);`
- ▶ `assertTrue(boolean)`



# Junit - Quando un test ha successo?

Un test ha successo quando tutte le asserzioni sono vere.

- ▶ Le asserzioni possono catturare le eccezioni e verificarne il loro

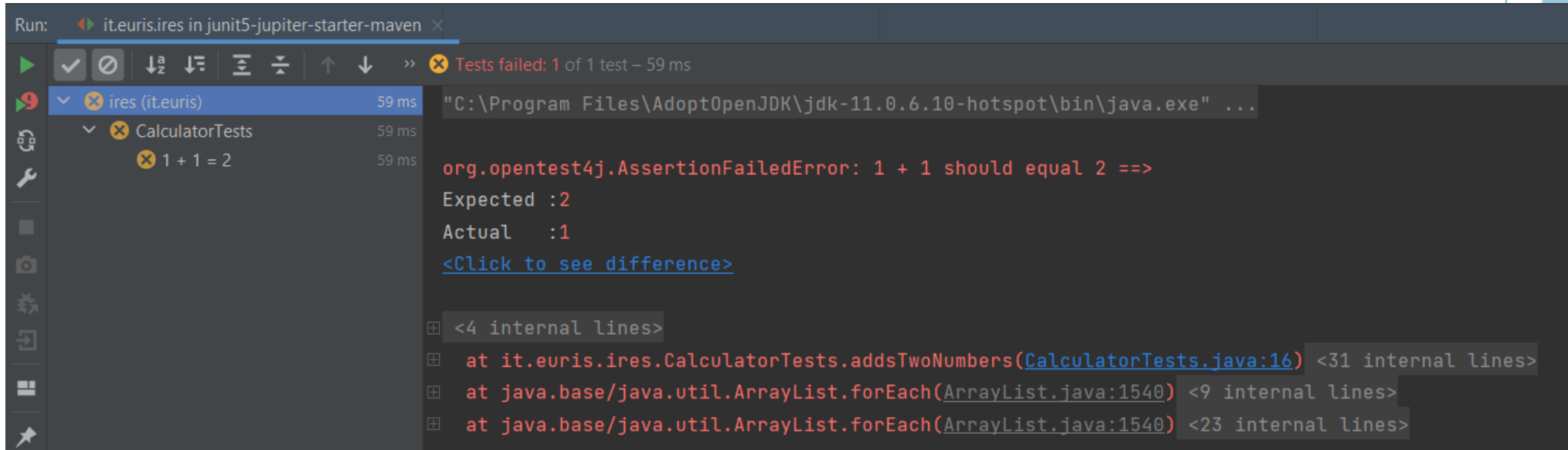


The screenshot shows the 'Run' console of an IDE. At the top, it says 'Run: it.euris.ires in junit5-jupiter-starter-maven'. Below this is a toolbar with icons for running, debugging, and other actions. The main area shows the test results: 'Tests passed: 1 of 1 test - 43 ms'. A tree view on the left shows the test hierarchy: 'ires (it.euris)' (43 ms), 'CalculatorTests' (43 ms), and '1 + 1 = 2' (43 ms). The right pane shows the command used to run the test: '"C:\Program Files\AdoptOpenJDK\jdk-11.0.6.10-hotspot\bin\java.exe" ...' and the message 'Process finished with exit code 0'.

```
Run: it.euris.ires in junit5-jupiter-starter-maven
Tests passed: 1 of 1 test - 43 ms
ires (it.euris) 43 ms
  CalculatorTests 43 ms
    1 + 1 = 2 43 ms
"C:\Program Files\AdoptOpenJDK\jdk-11.0.6.10-hotspot\bin\java.exe" ...
Process finished with exit code 0
```

# Junit - Quando un test fallisce?

- Un test fallisce quando un'asserzione è falsa o quando viene sollevata un'eccezione inattesa.



```
Run: it.euris.ires in junit5-jupiter-starter-maven X
Tests failed: 1 of 1 test – 59 ms

v ires (it.euris) 59 ms
  v CalculatorTests 59 ms
    x 1 + 1 = 2 59 ms

"C:\Program Files\AdoptOpenJDK\jdk-11.0.6.10-hotspot\bin\java.exe" ...

org.opentest4j.AssertionFailedError: 1 + 1 should equal 2 ==>
Expected :2
Actual   :1
<Click to see difference>

<4 internal lines>
at it.euris.ires.CalculatorTests.addsTwoNumbers(CalculatorTests.java:16) <31 internal lines>
at java.base/java.util.ArrayList.forEach(ArrayList.java:1540) <9 internal lines>
at java.base/java.util.ArrayList.forEach(ArrayList.java:1540) <23 internal lines>
```

# Esercizio 1c

- ▶ Implementare il metodo per la sottrazione all'interno della classe Calculator.
- ▶ Creare un test case per tale metodo all'interno della classe CalculatorTest.java

# Junit - Best practices



# Arrange-act-assert pattern

Questo pattern è un ottimo modo per strutturare dei casi di test: il test viene suddiviso in tre sezioni:

- ▶ **arrange**: serve a impostare le precondizioni, impostare le strutture dati e i mock. Infine il SUT viene inizializzato ed è pronto per essere utilizzato.
- ▶ **act**: il SUT viene eseguito e si ottiene il risultato da valutare.
- ▶ **assert**: il risultato viene valutato tramite asserzioni.

Questo pattern è talvolta definito GIVEN-WHEN-THEN dalla metodologia di sviluppo chiamata Behavior-Driven-Development (BDD).

# Arrange-act-assert pattern

```
10  @Test
11  @DisplayName("1 + 1 = 2")
12  void addsTwoNumbers() {
13      // arrange
14      int numberA = 1;
15      int numberB = 1;
16      int expectedResult = 2;
17      Calculator calculator = new Calculator();
18      // act
19      int result = calculator.add(numberA, numberB);
20      // assert
21      assertEquals(expectedResult, result,
22          String.format("%s + %s should equal %s", numberA, numberB, expectedResult));
23  }
24
```

# Junit 5 – @BeforeEach e @AfterEach

@BeforeEach definisce che il metodo annotato sia eseguito prima di ogni annotazione @Test, @ParameterizedTest, @RepeatedTest and @TestFactory.

Serve a inizializzare le precondizioni comuni ad ogni test.

@AfterEach definisce che il metodo annotato sia eseguito successivamente ad ogni annotazione @Test, @ParameterizedTest, @RepeatedTest and @TestFactory.

Viene utilizzato per pulire le strutture dati di test e rendere ogni test isolato l'uno dagli altri.

# Come chiamare i test case

I test case devono avere dei nomi specifici in modo che il loro scopo sia facilmente individuabile. Vi sono due modi per dare un nome descrittivo ad un test in junit5:

- ▶ Utilizzando l'annotazione `@DisplayName(String)`.
- ▶ Sfruttando il costrutto GIVEN-WHEN-THEN della metodologia BDD.



# GIVEN-WHEN-THEN

Il costrutto given-when-then serve a descrivere il comportamento del software attraverso un linguaggio semi-formale. La sequenza di azioni definite dal paradigma viene definito **scenario**.

- ▶ **Given / And:** il contesto che descrive lo stato iniziale del SUT
- ▶ **When / And:** le azioni che scatenano lo scenario
- ▶ **Then:** il risultato atteso

# GIVEN-WHEN-THEN

Un esempio di scenario relativo al metodo add() della classe Calculator.

**Scenario:** la somma di 1 e 1 ha come risultato 2

GIVEN: dato il metodo di add() della classe Calculator

WHEN: i dati in input sono 1 e 1

THEN: il risultato dovrebbe essere 2

# GIVEN-WHEN-THEN

```
10  @Test
11  void givenAddWhenInputsBoth1ThenReturn2() {
12      // arrange
13      int numberA = 1;
14      int numberB = 1;
15      int expectedResult = 2;
16      Calculator calculator = new Calculator();
17      // act
18      int result = calculator.add(numberA, numberB);
19      // assert
20      assertEquals(expectedResult, result,
21          String.format("%s + %s should equal %s", numberA, numberB, expectedResult));
22  }
```

# GIVEN-WHEN-THEN

```
24  @Test
25  @DisplayName("GIVEN subtract method WHEN first operator is 1 AND second operator is 1 THEN should return 0")
26  void subtractTwoNumbers() {
```

In questo caso il nome del caso del test viene descritto nell'annotazione, che permette di utilizzare un linguaggio semi-formale senza dover ricorrere al camelCase.

Utilizzare il costrutto GIVEN-WHEN-THEN permette allo sviluppatore/tester di ingegnerizzare il caso di test prima di scrivere il codice.

# Calculator – 1d

- ▶ Implementare le funzioni moltiplicazione e divisione
- ▶ Testare i nuovi metodi utilizzando
  - ▶ Assegnare al test un nome esplicativo del suo scopo, creando per ogni caso di test uno scenario.
  - ▶ il pattern arrange-act-assert per suddividere il caso di test.

# Junit - Dati di test



# Come scegliere i dati di test

I dati del test dipendono dalle precondizioni, ma alcune di esse sono implicitamente descritte nel tipo di dato.

In base a quale tipo di dato viene inserito si possono effettuare alcune analisi sui dati per decidere quali scegliere e generalmente due metodi vengono utilizzati, ovvero la partizione dell'input in classi di equivalenza e l'analisi dei valori limite.

# Partizione in classi di equivalenza

È una tecnica che può essere applicata a qualsiasi livello di test. In questa tecnica gli input vengono suddivisi in partizioni che raggruppano valori con una caratteristica simile e per ogni partizione viene selezionato un valore nominale che identifica la partizione.

Semplificando possiamo definire la caratteristica simile come la classe di equivalenza e i gruppi di valori come le partizioni. Ad esempio una classe di equivalenza può essere il segno negativo o positivo di una serie di numeri. Le partizioni sono i numeri positivi e i numeri negativi.



# Boundary Value Analysis

È una tecnica che basa il testing su input identificati alle estremità di una partizione di input. L'idea base di questo metodo è di utilizzare valori di input che si trovano:

- ▶ Al loro minimo
- ▶ Appena sopra il minimo
- ▶ Un valore nominale
- ▶ Appena sotto il massimo
- ▶ Al loro massimo

Questa tecnica viene applicata in combinazione con la tecnica di partizione in classi di equivalenza

# Scelta degli input - Esempio

Nel caso della addizione possiamo identificare due partizioni derivate dalla classe di equivalenza determinata dal segno:

- ▶  $[\text{Integer.MIN\_VALUE}, -1], [0, \text{Integer.MAX\_VALUE}]$

Mentre dalla BVA delle due partizioni otteniamo i seguenti valori

- ▶  $\{\text{Integer.MIN\_VALUE}, \text{Integer.MIN\_VALUE}+1, -50, -2, -1\}$
- ▶  $\{0, 1, 50, \text{Integer.MAX\_VALUE}-1, \text{Integer.MAX\_VALUE}\}$

# Scelta degli input – Caso di test

```
41 @ParameterizedTest(name = "GivenFirstArgument {0} AndSecondArgument {1} WhenAddThenShouldReturn {2}")
42 @CsvSource({
43     "0, 1, 1",
44     "0, -1, -1",
45     "-50, 50, 0",
46     Integer.MAX_VALUE + ", " + Integer.MAX_VALUE + ", -2"
47 })
48 void addWithBvaValues(int first, int second, int expectedResult) {
49     Calculator calculator = new Calculator();
50     assertEquals(expectedResult, calculator.add(first, second),
51         () -> first + " + " + second + " should equal " + expectedResult);
52 }
53
54 }
55
```

# Junit 5 – parametrized test

L'annotazione `@ParameterizedTest` rendere possibile eseguire un test più volte con differenti argomenti di input. Questa annotazione sostituisce la classica annotazione `@Test`.

In aggiunta all'annotazione bisogna assegnare un sorgente di valori al caso di test utilizzando le notazioni "Source": nell'esempio precedente è stata utilizzata l'annotazione `@CsvSource` che simula un documento csv come fonte di valori.

Per maggiori informazioni: <https://junit.org/junit5/docs/current/user-guide/#writing-tests-parameterized-tests-sources>

# Calculator – Esercizio 1e

- ▶ Identificate per i metodi relativi alla sottrazione, moltiplicazione, e divisione gli input di test utilizzando la partizione in classi di equivalenza e l'analisi dei valori limite.
- ▶ Implementate dei test che supportino tali valori utilizzando l'annotazione `@ParameterizedTest`

# Test doubles



# Isolamento e dipendenze

Non è improbabile di trovarsi a dover maneggiare un SUT che ha delle dipendenze da altre classi. Queste dipendenze non permettono di creare di effettuare i test di componente in totale isolamento e controllo in quanto il flusso di esecuzione si sposta al di fuori del nostro SUT.

Per risolvere questo problema si utilizza delle strutture chiamate «test doubles», ovvero dei componenti “falsi” che sostituiscono le vere dipendenze.

Il termine test doubles è stato coniato dalla analogia delle star cinematografiche con i relativi «stunt doubles».

I test doubles servono a rendere l'esecuzione del test deterministica.

# Test Doubles - Dummy

- ▶ È il test doubles più semplice la cui funzione è quella di soddisfare la dipendenza passandolo ad un costruttore.
- ▶ I metodi di una classe Dummy non devono avere implementazione.

I test dummies vengono utilizzati quando una dipendenza non apporta nessuna logica funzionale al comportamento del SUT.



# Test Doubles - Stub

- ▶ Sono leggermente più complessi dei Dummy. Forniscono delle risposte predefinite.
- ▶ I metodi di una classe Stub non prevedono logica e non sollevano eccezioni.

Gli stub vengono utilizzati quando si vuole ottenere una esecuzione deterministica di un caso di test, in modo che tale caso di test sia ripetibile con lo stesso risultato nonostante una componente di dipendenza cambi la sua implementazione.

# Test Doubles - Mock

- ▶ I mock sono oggetti che si aspettano di ricevere un determinato dato.
- ▶ Contengono dei metodi che permettono di verificare le aspettative impostate.

I mock servono a verificare la corretta interazione tra il SUT e le sue dipendenze.

# Test Doubles - Spy

- ▶ Gli Spy sono oggetti che registrano i dati che ricevono ma, a differenza dei mock, non si aspettano un determinato dato a priori.
- ▶ Contengono dei metodi che permettono di verificare se l'oggetto spy ha chiamato il dato desiderato.

Anche gli oggetti Spy servono a verificare la corretta interazione tra il SUT e le sue dipendenze. Si utilizzano quando non si ha la certezza di quale dato viene utilizzato dal SUT per interagire con le dipendenze.

# Test Doubles - Fake

- ▶ I Fake sono differenti dal resto dei Test Doubles.
- ▶ Non implementano risposte predefinite o degli strumenti di registrazione e verifica, ma ricreano la classe con una logica di business semplificata.
- ▶ Contengono dei metodi che permettono di verificare se l'oggetto spy ha chiamato il dato desiderato.

Vengono utilizzati per simulare il comportamento reale di un componente, senza tuttavia scomodare sotto dipendenze che potrebbero rendere il test non controllabile (e.g. interazioni con database, chiamate HTTP, etc.)

# Test doubles - Frameworks

Molti frameworks di test doubles presentano la sindrome della papera: possono fare molteplici azioni come nuotare, volare e camminare. Questi frameworks permettono con un solo costrutto di ottenere le funzionalità di dummies, mocks, spies e stubs.

Il framework Java che utilizzeremo in questo corso si chiama Mockito.

<https://site.mockito.org/>

# Test doubles - Mockito



# Mockito e Junit5

Mockito si integra con Junit5 tramite l'annotazione `@ExtendWith()`. L'argomento dell'annotazione è la classe `MockitoExtension.class` che permette a Junit di gestire i costrutti test double di Mockito durante l'esecuzione runtime del codice.

```
12
13     @ExtendWith(MockitoExtension.class)
14     public class MockitoExampleTest {
15
```

# Mockito - Mock

```
@Test
public void whenNotUseMockAnnotation_thenCorrect() {
    final List<String> mockList = Mockito.mock(List.class);
    mockList.add("one");
    Mockito.verify(mockList).add("one");
    assertEquals( expected: 0, mockList.size());

    Mockito.when(mockList.size()).thenReturn(100);
    assertEquals( expected: 100, mockList.size());
}
```



# Mockito – Mock tramite injection

```
26
27  @Mock
28  private List<String> mockedList;
29
30  @Test
31  public void whenUseMockAnnotation_thenMockIsInjected() {
32      mockedList.add("one");
33      Mockito.verify(mockedList).add("one");
34      assertEquals( expected: 0, mockedList.size());
35
36      Mockito.when(mockedList.size()).thenReturn(100);
37      assertEquals( expected: 100, mockedList.size());
38  }
39
```

# Mockito – Spy

```
43  @Test
44  public void whenNotUseSpyAnnotation_thenCorrect() {
45      final List<String> spyList = Mockito.spy(new ArrayList<String>());
46      spyList.add("one");
47      spyList.add("two");
48
49      Mockito.verify(spyList).add("one");
50      Mockito.verify(spyList).add("two");
51
52      assertEquals( expected: 2, spyList.size());
53
54      Mockito.doReturn( toBeReturned: 100).when(spyList).size();
55      assertEquals( expected: 100, spyList.size());
56  }
```

# Mockito – Spy with injection

```
@Spy
private final List<String> spiedList = new ArrayList<>();

@Test
public void whenUseSpyAnnotation_thenSpyIsInjectedCorrectly() {
    spiedList.add("one");
    spiedList.add("two");

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

    assertEquals( expected: 2, spiedList.size());

    Mockito.doReturn( toBeReturned: 100).when(spiedList).size();
    assertEquals( expected: 100, spiedList.size());
}
```

# Mockito - ArgumentCaptor

```
72  @Test
73  public void whenNotUseCaptorAnnotation_thenCorrect() {
74      final List<String> mockList = Mockito.mock(List.class);
75      final ArgumentCaptor<String> arg = ArgumentCaptor.forClass(String.class);
76      mockList.add("one");
77      Mockito.verify(mockList).add(arg.capture());
78
79      assertEquals("expected: 'one'", arg.getValue());
80  }
81
82  @Captor
83  private
84  ArgumentCaptor<String> argCaptor;
85
86  @Test
87  public void whenUseCaptorAnnotation_thenTheSam() {
88      mockedList.add("one");
89      Mockito.verify(mockedList).add(argCaptor.capture());
90
91      assertEquals("expected: 'one'", argCaptor.getValue());
92  }
93
94  }
```

# Esercizio

- ▶ Sostituire le implementazioni manuali dei mock con la controparte relativa di mockito.

# Test doubles – Best practices



# Best Practices

- ▶ Paradigma Query and Command Separation.
- ▶ Utilizza mock/stub solo su classi appartenenti al progetto.
  - ▶ Sei obbligato a utilizzare un test double su una classe esterna?  
Wrap it up!
- ▶ Non utilizzare i mock sulle strutture dati.
- ▶ Non aggiungere logica/comportamenti nei test doubles!
- ▶ Utilizza i mock/stub solo nelle dipendenze del SUT.
- ▶ Se il SUT ha bisogno di troppi test doubles rifattorizza il SUT.

# Command Query Separation

## ▶ Commands

- ▶ Non ritornano valori.
- ▶ Solitamente modificano i dati all'interno della classe.
- ▶ Utilizza il `verify()` di Mockito.

## ▶ Queries

- ▶ Ritornano dati.
- ▶ Non creano side-effects.
- ▶ Utilizza il `given()` di Mockito.



# Esercizio – Calculator 2

- ▶ Testare il pacchetto calculator con le nuove classi fornite.

# Cenni di teoria



# White-box testing

- ▶ Il sistema sotto test è trasparente: si conosce l'implementazione.
- ▶ Valuta principalmente la struttura del codice, ovvero la logica dell'implementazione.
- ▶ Le precondizioni e postcondizioni sono basate sulle specifiche e sull'implementazione.

# Coverage

- ▶ Statement Coverage: la copertura delle istruzioni.
- ▶ Branch Coverage: la copertura dei flussi di decisione.
- ▶ Condition Coverage: la copertura delle espressioni booleani che influiscono sui flussi di decisione.

# Black-box testing

- ▶ Il sistema sotto test è considerato una scatola nera: non si conosce l'implementazione.
- ▶ Si valuta solamente la relazione tra input e output, ovvero il comportamento del software.
- ▶ Le precondizioni e postcondizioni sono basate sulle specifiche.

# Test di integrazione

I test di integrazione vengono effettuati prima, durante e dopo l'integrazione di un nuovo modulo all'interno del software.

Lo scopo di questo livello di test è di identificare i difetti che possono nascondersi tra le interazioni di diversi moduli

# Test di Sistema

Questo livello di test mira a validare il prodotto software nel suo intero quando questo è completamente integrato e sviluppato.

Lo scopo principale è di valutare le specifiche end-to-end sia funzionali che non funzionali.

# Test di accettazione

Questo livello di test, generalmente definito come User-Acceptance Testing (UAT) ha lo scopo di validare il flusso business end-to-end del prodotto.

Questo tipo di test è tendenzialmente eseguito manualmente e viene effettuato in un ambiente simile all'ambiente di produzione.



# Testing Pyramid

