

# Unit Testing

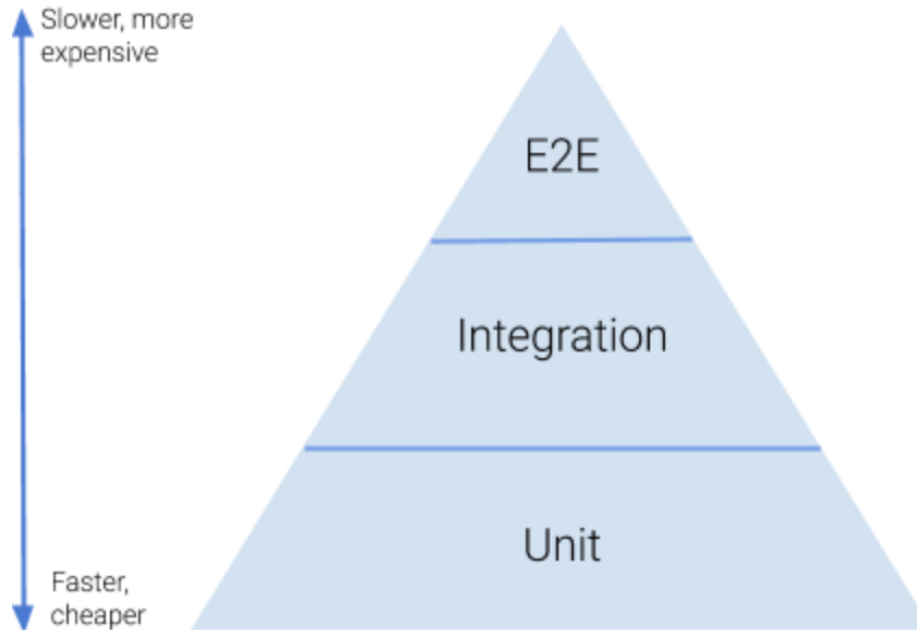
Ingegneria del Software  
A.A. 2020/2021

Prof. Andrea Fornaia

# Unit Test

- Obiettivo dell'attività di test è verificare che il comportamento del sistema sia conforme alle sue specifiche
- Uno **unit test** esamina il comportamento di una unità di lavoro. Spesso questa unità di lavoro è identificata come un singolo metodo
- 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
- Ogni test è caratterizzato da dati di input per il sistema e output stimati per tali input nel caso in cui il sistema operi secondo le sue specifiche
- Gli input sono non solo parametri da inviare ad una funzione, ma anche eventuali file, eccezioni, e stato del sistema, ovvero le condizioni di esecuzione richieste per poter eseguire il test
  - se diamo in input base=10 e exp=2 il metodo *pow()* restituirà 100?
  - se dopo aver chiamato il metodo *pianoTerra()* su un oggetto di tipo Ascensore chiamo il metodo *scendi()*, il metodo *getPiano()* restituirà 0?

# La piramide dei test



- 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

# JUnit

- JUnit è un framework per facilitare l'implementazione di programmi di test in Java
- Vedremo JUnit 4
- Principi 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 @Test

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

public class TestCalculator { // 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
    }
}
```

# Isolamento dei test

- 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() { ... }  
}
```

# Verifica automatica

- 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

# Asserzioni

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

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



# @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 Test con @Before

```
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);  
    }  
}
```

# 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 { }
```

# 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 attributi 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));
    }
}
```

# Assertj

```
<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

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

# Qualità di una test suite

- L'obiettivo di una test suite è quello di esercitare il comportamento di un'applicazione, distinguendo un comportamento anomalo (almeno uno dei test fallisce - **fail**) dal comportamento atteso (tutti i test superati - **pass**)
- Quanti e quali test dovremmo aggiungere per avere una test suite efficace?
- In teoria, il miglior modo per misurare la **qualità** di una test suite sarebbe di valutarne la capacità di individuare **difetti reali** (bug) nell'applicazione
- *“Il test di un programma può essere usato per mostrare la presenza di bug, ma mai per mostrare la loro assenza”* – Dijkstra (1970)
- Una delle metriche maggiormente usate per misurare la **qualità** di una test suite è la **copertura del codice** durante l'esecuzione dei test
- Non è l'unica metrica di copertura, ma è certamente una delle più immediate ed usate, anche grazie alla presenza di numerosi tool integrati con gli IDE
  - Cobertura
  - JaCoCo
  - Codcov
  - Icov

# JaCoCo: code coverage per Java

```
<build>
  <plugins>
    <plugin>
      <groupId>org.jacoco</groupId>
      <artifactId>jacoco-maven-plugin</artifactId>
      <version>0.8.7</version>
      <executions>
        <execution>
          <goals>
            <goal>prepare-agent</goal>
          </goals>
        </execution>
        <execution>
          <id>report</id>
          <phase>test</phase>
          <goals>
            <goal>report</goal>
          </goals>
        </execution>
      </executions>
    </plugin>
  </plugins>
</build>
```

Plugin da aggiungere al pom.xml (maven)

Il goal **prepare-agent** modifica automaticamente il codice delle classi a livello bytecode per permettere di registrare le linee di codice coperte durante l'esecuzione dei test

Il goal **report** crea un report (anche web) sul risultato della copertura del codice, evidenziando ad esempio quali linee del codice sono state coperte e quali no

Associamo la creazione del report alla fase di test: il report verrà quindi generato quando eseguiamo

**\$ mvn test**

## Coverage Gutters ryanluker.vscode-coverage-gutters

ryanluker | 102,404 | ★★★★★ | Repository | License | v2.7.3

Display test coverage generated by lcov or xml - works with many languages

Disable

Uninstall

⚙️ This extension is enabled globally.

Estensione di VS Code per evidenziare le linee di codice coperte

È necessario che sia stato generato il report di copertura (con mvn test)




# Code Coverage

- **Class coverage:** percentuale di *classi* coperte
- **Method coverage:** percentuale di *metodi* coperti
- **Line coverage:** percentuale di *linee* di codice coperte
- **Statement coverage:** percentuale di *istruzioni* coperte
  - su una linea posso avere più di una istruzione
  - più preciso
  - **nota:** JaCoCo conta le istruzioni coperte a livello di bytecode
- **Branch coverage:** percentuale di rami di esecuzione coperti
  - Es. if (a && b) ha 2 rami da coprire:
    - (a && b) è vera
    - (a && b) è falsa
- **Condition coverage:** percentuale di condizioni coperte
  - Es. if (a && b) ha 4 condizioni da coprire:
    - (a) è vera
    - (a) è falsa
    - (b) è vera
    - (b) è falsa
  - JaCoCo conta i branch coperti a livello di bytecode, è più quindi una condition coverage



# Esempio code coverage con JaCoCo

## Str

Element	Missed Instructions	Cov.	Missed Branches	Cov.
• <a href="#">foo(boolean, boolean)</a>		100%		50%
• <a href="#">Str()</a>		100%		n/a
Total	0 of 13	100%	2 of 4	50%

## Str.java




```
1. public class Str {
2.
3.     public String foo(boolean a, boolean b) {
4.         String result = "KO";
5.         if (a && b) {
6.             result = "OK";
7.         }
8.         return result;
9.     }
10.    /**
11.     Bytecode
12.     0: ldc         #16 // String KO
13.     2: astore_3
14.     3: iload_1
15.     4: ifeq         14
16.     7: iload_2
17.     8: ifeq         14
18.    11: ldc         #18 // String OK
19.    13: astore_3
20.    14: aload_3
21.    15: areturn
22.    LineNumberTable:
23.     line 4: 0
24.     line 5: 3
25.     line 6: 11
26.     line 8: 14
27.    */
28. }
```

```
@Test
public void shouldGiveOkWhenBothTrue() {
    assertEquals("OK", foo(true,true));
}
```

- Le istruzioni di *foo()* sono 10 (le altre 3 sono del costruttore)
- La copertura degli statement è 10/10
- La copertura dei branch (a livello bytecode) è 2/4
- La copertura delle linee è 4/4 (nel sorgente)
- La copertura dei metodi è 2/2 (per la classe corrente)
- Dobbiamo aggiungere dei test per aumentare la branch coverage

# Esempio code coverage con JaCoCo

## Str

Element	Missed Instructions	Cov.	Missed Branches	Cov.
<a href="#">foo(boolean, boolean)</a>		100%		100%
<a href="#">Str()</a>		100%		n/a
Total	0 of 13	100%	0 of 4	100%

## Str.java

```
1. public class Str {
2.
3.     public String foo(boolean a, boolean b) {
4.         String result = "KO";
5.         if (a && b) {
6.             result = "OK";
7.         }
8.         return result;
9.     }
10.    /**
11.     * Bytecode
12.     * 0: ldc         #16 // String KO
13.     * 2: astore_3
14.     * 3: iload_1
15.     * 4: ifeq        14
16.     * 7: iload_2
17.     * 8: ifeq        14
18.     * 11: ldc        #18 // String OK
19.     * 13: astore_3
20.     * 14: aload_3
21.     * 15: areturn
22.     * LineNumberTable:
23.     *   line 4: 0
24.     *   line 5: 3
25.     *   line 6: 11
26.     *   line 8: 14
27.     */
28. }
```

```
@Test
public void shouldGiveOkWhenBothTrue() {
    assertEquals("OK", foo(true,true));
}

/* +++++ test aggiunti +++++ */

@Test
public void shouldGiveKoWhenBothFalse() {
    assertEquals("KO", foo(false,false));
}

@Test
public void shouldGiveKoWhenLastFalse() {
    assertEquals("KO", foo(true,false));
}
```

- La branch coverage è ora 4/4

# Limiti della code coverage

- Avere una code coverage elevata è una condizione necessaria ma in certi casi può non essere sufficiente
- Il fatto che abbiamo ottenuto il 100% della copertura del codice non significa che abbiamo coperto tutti i possibili comportamenti dell'applicazione
  - Cicli, ricorsione, ordine di esecuzione dei metodi, codice mancante...
- La copertura ci indica solo che abbiamo eseguito il codice durante i test, non che ne abbiamo testato il comportamento in maniera significativa
- Il caso limite è quello di test senza asserzioni: il codice verrà coperto, ma non sto verificando la correttezza del comportamento
- Anche in presenza di asserzioni, queste devono essere scelte in modo tale che siano in grado di individuare la presenza di possibili comportamenti anomali
- In molto casi, il problema non risiede nei test presenti, ma nei **test mancanti**: è necessario considerare dei test case aggiuntivi (come visto nell'esempio con JaCoCo)

# Test Driven Development

- Tecnica di sviluppo agile che consiste nel dividere lo sviluppo in cicli rapidi in cui si alternano le seguenti fasi:
  - **Red:** aggiungere un piccolo test per una funzionalità da implementare o per uno scenario di utilizzo che dovrebbe essere considerato; in questo momento il test dovrebbe fallire
  - **Green:** scrivere il codice sufficiente per superare il test senza far fallire quelli precedentemente scritti
  - **Refactor:** migliorare la qualità del codice scritto senza far fallire nessuno dei test
- Vantaggi:
  - *Prima scrivi il test, poi il codice:* si è obbligati a ragionare prima su cosa bisogna fare invece che sul come farlo, prendendo delle decisioni di design (nomi classi, metodi) durante la scrittura del test; l'IDE permetterà di automatizzare la creazione delle classi e dei metodi dichiarati nei test
  - *Scrivi solo il codice per superare il test:* si tende a dare priorità a soluzioni semplici, migliorabili in seguito tramite refactoring, se necessario
  - *Feedback rapido:* ci si concentra sulla risoluzione continua di piccoli problemi specifici
  - *Refactoring periodico:* si ragiona fin da subito sul miglioramento della qualità del codice (dal cambiamento di un nome di una variabile all'uso di una struttura dati, un algoritmo, o un design pattern differente) con il vantaggio di avere già dei test in grado di verificare che il comportamento esterno non sia cambiato
- Svantaggi
  - Serve esperienza, sia nella scrittura dei test che nella scrittura del codice e relativo refactoring

# Test Double e Unit Test

- A differenza di un integration test, uno unit test deve testare una singola unità (es. una classe) in maniera indipendente dalle altre classi con cui **collabora**
- Necessario simulare il comportamento delle componenti da cui il metodo testato dipende tramite dei **test double** (controfigure)
- Ci sono diversi tipi di test double, ma i più noti e usati sono gli **stub** e i **mock**
- Lo **stub** è un'istanza di una classe il cui comportamento di alcuni dei suoi metodi viene definito direttamente nel test:
  - 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)
- Il **mock** è uno stub che inoltre permette di verificare **quali e quante volte** i suoi metodi sono stati chiamati durante il test:
  - permette di testare il comportamento di metodi **void** o di metodi che aggiornano uno **stato esterno** alla classe da testare (es. gli attributi del collaboratore)
- Deve essere possibile passare test double come dipendenza alla classe da testare (es. tramite costruttore, parametro, o metodo setter): questa pratica viene detta **dependence injection**

# Esempio di Mock senza uso di librerie

- Supponiamo di voler testare il metodo *accendi()* di una classe *Torcia* che invoca al suo interno il metodo *isScarica()* definito in *Batteria*
- Vogliamo testare il caso in cui la torcia non deve accendersi se la batteria è scarica
- Un'istanza compatibile con il tipo *Batteria* viene passata in questo caso come dipendenza al costruttore della classe da testare, ridefinendo solo il metodo ***isScarica()*** in modo che ci **restituisca sempre *true***
- *Batteria* potrebbe anche essere un'interfaccia o una classe astratta, o comunque una classe priva di un'implementazione completa

```
@Test
public void seBatteriaScarica_quandoAccesa_nonSiAccende() {

    Torcia torcia = new Torcia(new Batteria(123) { // dependence injection
        @Override
        public boolean isScarica() {
            return true;
        }
    });

    torcia.accendi(); // al suo interno chiama isScarica() per determinare se può accendersi
    assertTrue(torcia.isSpenta())
}
```

# Mockito

```
<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...thenReturn)
  - di verificare se e con quali parametri un determinato metodo del mock sia stato chiamato durante l'esecuzione del test (verify)
- A differenza dell'approccio tramite classi anonime, non siamo obbligati a definire il comportamento di tutti i metodi astratti

# Esempio di Mock con Mockito

```
import org.mockito.Mock;
import static org.mockito.Mockito.*; // when, verify, times
import static org.mockito.MockitoAnnotations.initMocks;

public class TorciaTest {
    @Mock
    Batteria batteria;

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

    @Test
    public void seBatteriaScarica_quandoAccesa_nonSiAccende_conMockito() {

        when(batteria.isScarica()).thenReturn(true);

        Torcia torcia = new Torcia(batteria);
        torcia.accendi();

        verify(batteria, times(1)).isScarica(); // posso verificare se è stato chiamato
                                                // una sola volta (conta come asserzione)
        assertTrue(torcia.isSpenta())
    }
}
```

- **@Mock** annota gli attributi a cui associare le istanze di mock
- **initMock(this)** crea il mock (si inserisce tipicamente nel metodo con @Before)
- **when()** e **thenReturn()** servono a definire il valore da restituire quando viene chiamato un determinato metodo del mock
- **verify()** permette di verificare se e quante volte il metodo del mock è stato chiamato



# Verifica dello stato VS Verifica del comportamento

- Supponiamo di avere una classe da testare *Torcia* che collabora con *Batteria*
- Il metodo da testare *Torcia.accendi()* chiama anche il metodo **Batteria.consuma()** per consumare la batteria, aggiornandone lo stato
- 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 *accendi()* non ha un valore di ritorno e il cambiamento di stato avviene su un collaboratore (la batteria) che non è il componente che vogliamo testare: potrebbe non essere stato ancora implementato o potremmo non voler rendere questo stato accessibile dall'esterno, es. con *getLivello()*
- 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 livello della batteria sia diminuito dopo la chiamata al metodo *Torcia.accendi()*, ci limitiamo a verificare che il metodo *Batteria.consuma()* sia stato chiamato
- Il secondo approccio, reso possibile dai mock, ci permette di testare *Torcia* indipendentemente da un'implementazione di *Batteria*

```
public class TorciaTest {
    private Batteria batteria;
    private Torcia torcia;

    @Before
    public void setUp() {
        batteria = new Batteria(1);
        torcia = new Torcia(batteria);
    }

    @Test
    public void quandoAccesa_consumaNienteEnergia() {
        int livelloPrima = batteria.getLivello();
        torcia.accendi();
        int livelloDopo = batteria.getLivello();
        assertTrue(livelloDopo < livelloPrima);
    }
}
```

```
public class TorciaTest {  
    @Mock  
    private Batteria batteria;  
    private Torcia torcia;  
    @Before  
    public void setUp() {  
        initMocks(this);  
        torcia = new Torcia(batteria);  
    }  
    @Test  
    public void quandoAccesa_consumaN Batteria_conMockito() {  
        torcia.accendi();  
        verify(batteria, times(1)).consumo(); // verifico il comportamento, ovvero  
                                                // se consumo è stato chiamato una  
                                                // e una sola volta  
    }  
}
```

# Verifica del comportamento

```
public class TorciaTest {  
    @Mock  
    private Batteria batteria;  
    private Torcia torcia;  
    @Before  
    public void setUp() {  
        initMocks(this);  
        torcia = new Torcia(batteria);  
    }  
    @Test  
    public void quandoAccesa_consumoBatteria_conMockito() {  
        torcia.accendi();  
        verify(batteria, times(1)).consumo(); // verifico il comportamento, ovvero  
                                                // se consumo è stato chiamato una  
                                                // e una sola volta  
    }  
}
```

# Verifica del comportam

# Riferimenti

- J. B. Rainsberger: *JUnit Recipes. Practical Methods for Programmer Testing*
- T. Kaczanowski: *Practical Unit Testing with Junit and Mockito*
- K. Beck: *Test Driven Development by Example*
- <https://martinfowler.com/articles/practical-test-pyramid.html>
- <https://assertj.github.io/doc/>
- <https://martinfowler.com/articles/mocksArentStubs.html>