Fundamental Challenges

Prof. A. M. Calvagna

Outline

- Principles and practices for dealing with fundamental challenges:
  - –Requirements can change: apply the right **Process**!
  - –Software is intrinsically complex: **Design** for easy maintenance!
  - –Defects are inevitable: **Test** the software!
- These challenges have implications for what to build and how to build it

Faults versus Failures

- Faults are in the static source code or in documents
  - –Defect, bug, and anomaly are synonyms for faults
- Failures are triggered at run time (dynamically)
  - –A failure is unexpected behavior, with crashes being dramatic failures
- Example: When an app crashes some of the time
  - –There must be a fault or faults in the app's code
  - –But, failures occur only some of the time; e.g., when there is a crash

Scriviamo ed eseguiamo test

- Per controllare la *correttezza* del software realizzato

- Il confronto è con *le specifiche* da cui siamo partiti

Correttezza

- Eseguire test permette di controllare se **il codice soddisfa le specifiche del cliente: i test si possono scrivere a partire dalla specifica dei requisiti, dal codice o da entrambi**
- Testare in modo ampio permette di ottenere una valutazione sulla **qualità** del sistema
  - I test devono essere **indipendenti** fra loro ed **auto valutarsi**
- **Responsabilità**: i compiti sono suddivisi uno per metodo, o per classe, questo permette di ottenere **coesione** del codice e che i test di correttezza di una data funzionalità si possano concentrare su un singolo metodo o classe
  - **Principio di Singola Responsabilità (SRP)**. La singola responsabilità è cruciale per la buona comprensione, il riuso, l'ereditarietà (impedisce ambiguità e sovrapposizioni tra gli ADT)


```java
public class Pagamenti { // Pagamenti vers 1.2 (senza command and query)
    private List<Float> importi = new ArrayList<>();
    public void leggiFile(String c, String n) throws IOException {
        LineNumberReader f = new LineNumberReader(new FileReader(new File(c, n)));
        String riga;
        while (true) {
            riga = f.readLine();
            if (riga == null) break;
            inserisci(Float.parseFloat(riga));
        }
        f.close();
    }
    public void inserisci(float x) {
        if (!importi.contains(x)) importi.add(x);
    }
    public float calcolaSomma() {
        float risultato = 0;
        for (float v : importi)  risultato += v;
        return risultato;
    }
    public float calcolaMassimo() {
        float risultato = 0;
        for (float v : importi)  if (risultato < v)  risultato = v;
        return risultato;
    }
    public void svuota() {
        importi = new ArrayList<>();   }
}
```

Pagamenti

– importi: List<Float>

+ leggiFile(c: String, n: String)
+ inserisci(x: float)
+ calcolaSomma(): float
+ calcolaMassimo(): float
+ svuota()


```java
public class TestPagamenti { // per classe Pagamenti vers 1.2
    private Pagamenti pgm = new Pagamenti();
    private void initLista() {
        pgm.svuota();
        pgm.inserisci(321.01f);
        pgm.inserisci(531.7f);
        pgm.inserisci(1234.5f);
    }
    public void testSommaValori() {
        initLista();
        if (pgm.calcolaSomma() == 2087.21f) System.out.println("OK test somma val");
        else System.out.println("FAILED test somma val");
    }

    public void testListaVuota() {
        pgm.svuota();
        if (pgm.calcolaSomma() == 0) System.out.println("OK test somma val empty");
        else System.out.println("FAILED test somma val empty");
        if (pgm.calcolaMassimo() == 0) System.out.println("OK test massimo val empty");
        else System.out.println("FAILED test massimo val empty");
    }
   // continua
```



```java
   // continua
   public void testLeggiFile() {
        try {
        pgm.leggiFile("csvfiles","Importi.csv");
        System.out.println("OK test leggi file");
        } catch (IOException e) {
        System.out.println("FAILED test leggi file");
      }
    }

   public static void main(String[] args) {
        TestPagamenti tl = new TestPagamenti();
        tl.testLeggiFile();
        tl.testSommaValori();
        tl.testMaxValore();
    }
}
```

Output dell'esecuzione di TestPagamenti quando il file Importi.csv è presente nella cartella csv (dentro la cartella con gli eseguibili)

```
OK test leggi file
OK test somma val
OK test massimo val
```

Esecuzione quando il file non viene trovato

```
FAILED test leggi file
OK test somma val
OK test massimo val
```

Testing Terminology

- **Unit testing** is named after units of implementation (single function, single class in OOP)
- Regression testing runs all **known tests** to verify that nothing broke
- A **test suite** is a set of test inputs (—> test case instances/items)
- A **System Verification Group** is a group dedicated to testing

Testing Terminology

–**Black-box** tests are based on inputs and outputs; no need for the code

–**White-box** tests are based on knowledge of the code

–**Validazione** funzionale(= testing) : "Am I building the right product?"

  –Controllo se le funzionalità ottenute corrispondono a quelle attese (alla specifica dei requisiti)

  –È il punto di vista dell'utente/utilizzatore: sta funzionando?

–**Verifica** della correttezza : "Am I building the product right?"

  –É il punto di vista dall'interno, dello sviluppatore

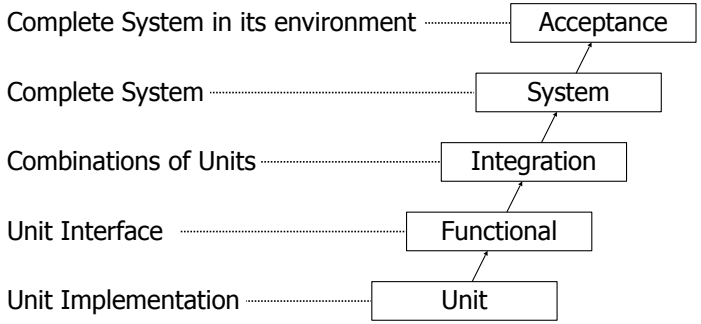
  –I meccanismi interni sono corretti?

  –Es.: testo il metodo int somma(a,b){ return a*b-1; } con a=3 e b=2

  –Il test è superato perché ritorna 5, ma è solo un caso…

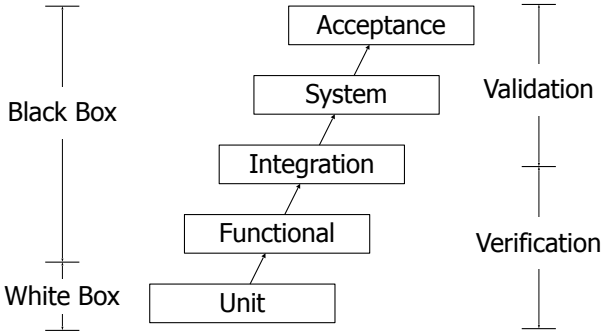
Levels of Testing

Software Under Test and Levels of Testing



Primary Focus of the Levels of Testing

- Il WBT è possibile solo sulle Unità (di codice), per definizione
- Ma le Unit ammettono anche il BBT
- Fino al livello precedente l'integrazione, ha senso parlare di verifica formale di correttezza dei meccanismi interni
- Nella pratica, è più comune che sia testing a tutti i livelli



Testing

- Validation tests tests tend to be black-box
- Verification tests tend to be white-box, but **black-box tests can also be used**
- With agile methods, developers test as they code
- Continuous deployment relies heavily on **automated testing**
- Manual steps add delay and potentially errors

Testing is a Form of Sampling

- We have two typically infinite sets, called populations or domains
  - –Inputs: integer inputs form an infinite set
  - –Executions: with for(i=1; i<N; i++){…}, N can be any integer
- Choose tests so they faithfully represent the population
  - –Black-box tests are chosen to **sample the input domain**, as we'll see
  - –White-box tests are chosen to **sample executions**, as we'll see

Testing Cannot Prove the Absence of Bugs

- Example: There had been 300 successful test runs of the guidance system
  - –Shortly after launch, the Atlas rocket wobbled off course
  - –293 seconds after launch, it was destroyed by the safety officer
- What happened?
  - –Signal contact with the ground was lost
  - –The engineers had anticipated this possibility, but the code had a bug
- Moral of the Story: 300 test runs did not uncover the bug

Formal Verification

–I meccanismi interni sono corretti ?

–Il testing non da la certezza che funzioneranno sempre

–Riduco la complessità eccessiva passando dal codice ad un suo "modello formale"

–Provo con certezza matematica la sussistenza o meno di proprietà di correttezza sul modello (ad. Es. assenza di deadlock, di loop infiniti, etc…)

1. Con "Proof Assistant" tools (PVS, Coq, HOL, Isabelle, etc…)
2. Con "Model Checker" tools (SPIN, NuSMV, PathFinder, etc…) che cercano nel modello contro-esempi che falsificano la proprietà

Una astrazione (modello) sbagliata anche di pochissimo può alterare dettagli importanti del codice, e le proprietà dimostrate non sono più trasferibili dal modello (al software, e quindi) alla realtà.

Case Study: Large Project, Strong Focus on Testing

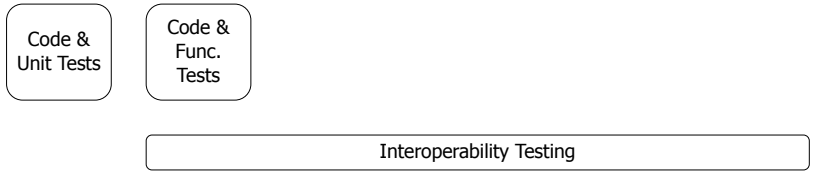
- Planning phase
  - –Extensive customer involvement
  - –Highly skilled team with deep experience with similar projects
- Sprints measured in weeks
  - –Feature sets developed in parallel by subteams
  - –Nightly builds and regression tests
  - –Intense focus on early detection of correctness and performance issues

Develop Feature Sets in Parallel

Deliver code and unit tests and perform sanity functional tests.

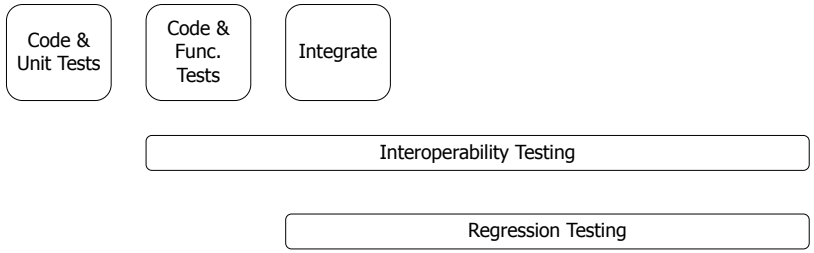
Deliver Code and Automated Functional Tests

Also, start Interoperability Testing with third party systems



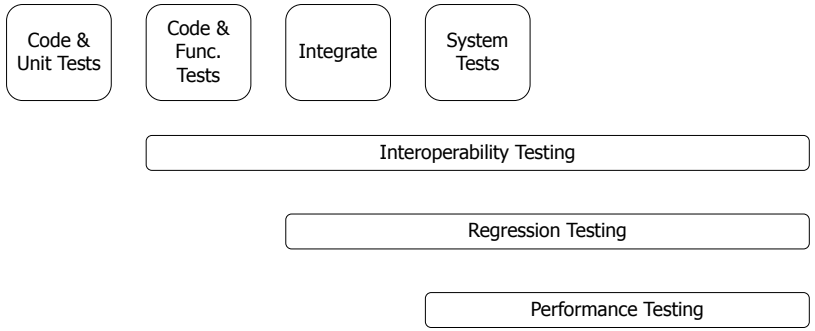
Integration: Approve Features for SV

Start Regression Testing (SV stands for System Verification)



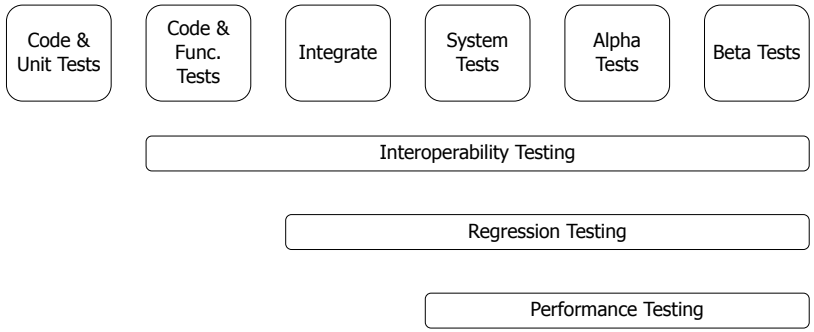
SystemTest: Test Features for Failure Scenarios

Start Performance Testing



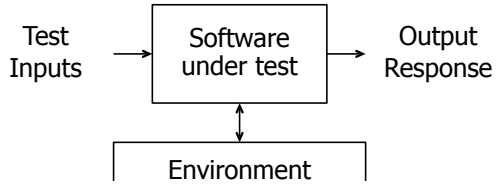
Internal Alpha and External Beta Tests

Alpha tests are trials within the company, beta tests are customer trials



Testing Overview

- 1. Decide on the Software Under Test and white box or black box
- 2. Set up its Environment so the code can be run repeatably
- 3. Select a set of test inputs based on a criterion for good tests
- 4. Evaluate the outputs with the intent of finding bugs



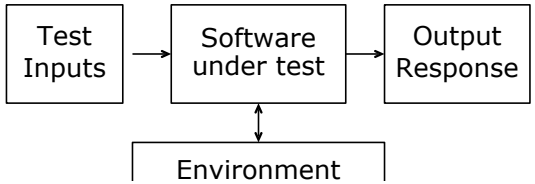
Defining a Test Strategy

- What is the software under test?
  - **White-box** tests are based on knowledge of the code
  - **Black-box** tests are based only on inputs and outputs
- What does the software need from its **environment** to before it can run?
  - The software is typically not self-standing. It needs **values or functions**.
- How will test inputs be selected? What is a good test set?
- How will the output be evaluated? That is, did a test pass or fail?

Select Tests for Coverage

- How will test inputs be selected? What is a good test set?
- How much tests are enough?
- **Coverage** is the fraction of constructs touched by a set of tests
- A **coverage metric** is the criterion to quantitatively measure the coverage
- Select white-box tests for **code coverage**
  - Statement coverage is a typical code-coverage criterion
  - Decision or Branch coverage is a stronger criterion
- Select black-box tests for **input-domain coverage**
  - Select test inputs so they are a representative sample of all inputs
  - Test all Combinatorial Interactions of a degree

Testing Context : what is the environment?



The Role of the Environment

- The environment provides the software under test what it needs to run
  - Often, the software under test is **not self-standing**
- Example: `day()` converts an object of class `Date` to a day of the year
  - `int day(Date d) { … if( isleap(d.year) { … } … }`
- For the code for r `day()` to run, the environment must supply
  - An input `d`, a declaration of `Date`, and a function `isleap()`

Stabilize the Environment

- What does it mean to stabilize the environment?
  - –Provide consistent values to make tests repeatable
  - –Provide a controlled setting for testing expected and unexpected behavior
- How?
  - –Simulate the interfaces and the messages across them
  - –Interfaces include software APIs, file systems, communications
  - –Human inputs can be a challenge; e.g., gestures that cannot be recorded

Drivers and Stubs

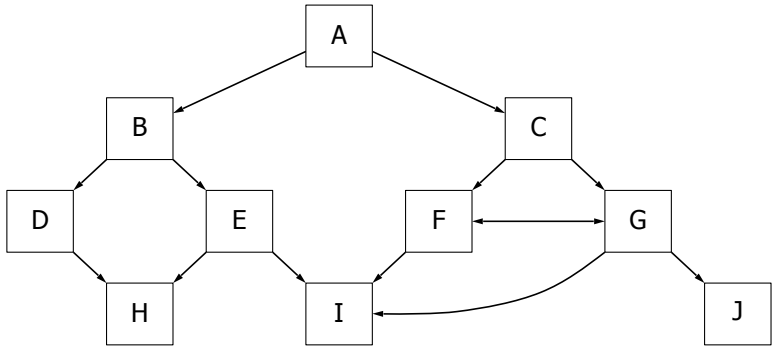
- A driver for module N
  - –Simulates an environment for N
  - –May provide stubs for modules or functions used by N
- A stub for module M
  - –Has the same interface a as M
  - –Simulates the functionality provided by M
  - –May return dummy or pre-programmed values
  - –Usually called "mock objects"

System Integration and Testing

- Start with unit-tested modules
  - –Then, integration errors are likely to be due to module interactions
- Integrate the whole system all at once, if possible
  - –Then, the environment is only for external services
- Incremental integration builds up the system in stages
  - –Top-down integration requires stubs for used modules
  - –Bottom-up integration requires drivers

What order would you integrate and test these modules?

An edge from M to N means that module M uses module N

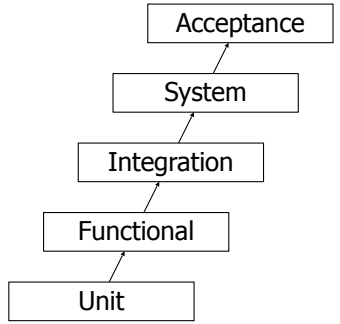


Incremental Integration

- Bottom-up integration
  - –Any order in which used modules appear before using modules
- Top-down integration
  - –Using modules appear before used modules
- Modules that are in a cycle must be integrated and tested together

Test Automation

- Continuous deployment relies heavily on automated testing
- Manual steps add delay and potentially errors
- Example test automation tools:
  - JUnit
  - TestNG
- Example test generation tools
  - Randoop, EvoSuite, Ttuples



JUnit

- Integrazione con tool e IDE diffusi (Eclipse, IntelliJ, VS Code)
- Scrivo classi per il test, in cui inserisco metodi annotati con **@Test**
- Ogni metodo di test è trovato ed eseguito automaticamente dal framework (usando la riflessione computazionale)
- Ogni metodo viene eseguito in una nuova istanza della classe di test
  - per avere ripetizioni indipendenti ed evitare effetti collaterali
- Si usano **asserzioni** per automatizzare la valutazione dell'esito del test

Assertions

- La classe Assert fornisce i metodi usati per valutare il test
- assertTrue(boolean condition) valuta se condition è true
- assertEquals(a, b) verifica se due oggetti sono uguali
- Altri metodi assert

```
assertNull(Object object)
assertSame(expected, actual)
assertTrue(boolean condition)
assertFalse(boolean condition)
fail(String message)  //termina il test, utile per testare le routines di eccezione
```

- I metodi assert registrano fallimenti o successi e li riportano
- Quando si registra un fallimento, l'esecuzione del metodo di test corrente viene interrotta, ma si prosegue con gli altri

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

public class TestCalc { // 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
}
```

Esempio calculator su vscode…

Test suites

- Una selezione di classi di test può confluire in una test suite
- Posso raggruppare allo stesso modo alcune test suites in una test suites di livello superiore
- altrimenti posso eseguire singoli metodi, singole classi o tutto.

```java
@RunWith(Suite.class) // Runner: questa è una test suite
@Suite.SuiteClasses({ConsumerTest.class, SommatoreTest.class})
public class TestSuiteA { }

@RunWith(Suite.class) // Runner: questa è una test suite
@Suite.SuiteClasses({CalculatorTest.class, SommatoreTest.class})
public class TestSuiteB { }


@RunWith(Suite.class) // Runner: questa è una test suite di suites
@Suite.SuiteClasses({ TestSuiteA.class, TestSuiteB.class })
public class MasterTestSuite { }
```


@Before e @After

- Un metodo annotato con @Before (setUp)
  - Viene chiamato prima di ogni @Test
  - Lo uso per inizializzare lo stato dell'istanza della classe di test
    - In modo che i tutti i test partano dalle stesse condizioni
  - Un metodo @After (tearDown)
  - Per operazioni di chiusura o finali (clean)
  - Viene eseguito comunque sia andato il test

- Vedi esempio Calcolator2…

@BeforeClass e @AfterClass

- @BeforeClass annota un metodo statico
  - Eseguito prima di iniziare il lancio dei @Test della classe
  - Utile per eseguire operazioni costose, una tantum
    - es. apertura connessione remota e query a un database
  - Essendo un metodo statico, può modificare solo attributi statici della classe

- Analogamente @AfterClass

- Esempio: Singleton su VSCODE

AssertJ

- AssertJ può essere usata assieme a Junit per definire asserzioni complesse con uno stile *fluent*
- Permette di scrivere asserzioni concatenate, migliorando la leggibilità

```
String hobbit = "frodo";

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

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


Mock Objects (oggetti finti)

ConsumerIntegrationTest
SommatoreTest
Consumer.TestSumTwice_NoMockito

- Il test di unità deve testare la correttezza di **una singola classe** (o sarebbe un integration test…)
  - **simulo** il comportamento delle classi con cui collabora, con degli oggetti **finti**.
  - devo **isolare** la classe sotto test: lo sviluppo delle altre classi può essere incompleto (c'è solo l'interfaccia), non ancora testato o comunque errato
- creo **un'istanza** di una classe che implementa per finta **l'interfaccia** del vero collaboratore, oppure direttamente un'istanza di sottoclasse anonima
- Definisco il comportamento dei suoi metodi direttamente nel test, limitato a restituire un risultato predefinito per input predefiniti (comportamento da **stub**)

Tipologie di Test Doubles

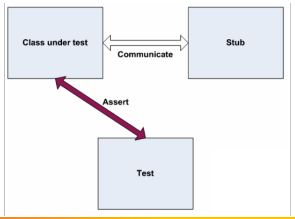
- **Dummy** objects are passed around but never actually used. Usually they are just used to fill parameter lists.
- **Fake** objects actually have working implementations, but usually take some shortcut which makes them not suitable for production (an in memory database is a good example).
- **Stubs** provide canned answers to calls made during the test, usually not responding at all to anything outside what's programmed in for the test.
- **Spies** are stubs that also record some information based on how they were called. One form of this might be an email service that records how many messages it was sent.
- **Mock objects** can be programmed to know how many times and in what order functions should be called during testing.

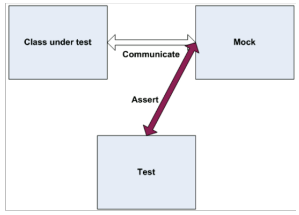
Mock vs Stub

Gli stub servono a fornire dati di ritorno alle chiamate della classe testata:
Decido l'esito del test con asserzioni che esaminano lo stato della classe testata
Con mockito possono fare anche questo.

Mock sono oggetti finti creati per controllare il comportamento della classe sotto test: **le sue interazioni con il mock**
Tiene conto del **numero**, dell'**ordine** e dei valori nei **parametri** delle chiamate ricevute durante il test
Le asserzioni, e l'esito del test, esaminano lo stato del mock, non della classe testata.

BDD: Behaviour Driven Development (complementare al Test Driven Development)





Behaviour Driven Development (BDD)

Style of writing tests that uses //given //when //then comments as fundamental parts of your test methods.

The acceptance criteria validating the feature under test is a scenario, with the following structure:

- **Given**: the initial context at the beginning of the scenario, in one or more clauses; che specificano cosa va restituito dai metodi mock quando invocati
- **When**: the event that triggers the scenario; La classe testata viene usata e usa in cascata anche il mock object
- **Then**: the expected outcome, in one or more clauses: asserzioni sullo stato e/o sulle interazioni

Mockito per Java

Consumer.TestSumTwiceMockito su vscode

Libreria che semplifica la scrittura di test coi Mock objects

Non è più necessario implementare nei test classi finte

Vanno solo dichiarate e inizializzate col un comando: **initMock()**

```
@Mock
Collaborator mockObj;

//sintassi alternativa, senza annotazione
Collaborator mockObj = mock(Collaborator.class);
```

Mockito permette:

Di indicare direttamente durante l'esecuzione del test, quale valore restituire quando un metodo del mock viene invocato (**when/given**)

Di verificare se e con quali input un determinato metodo del mock sia stato chiamato durante l'esecuzione del test (**verify/then**): vero uso da Mock

MockitoBDD per Java

Il test valuta se, quante volte e con quali input i metodi dei mock sono stati chiamati durante l'esecuzione del test

```
Person person = mock(Person.class);
Seller police = mock(Police.class);
Bike bike = new bike();

//when
person.ride(bike);
person.ride(bike);

//then
then(person).should(times(2)).ride(bike);
then(person).shouldHaveNoMoreInteractions();
then(police).shouldHaveZeroInteractions();
```

Consumer.TestSumTwiceBDD

Test parametrici

Il metodo (o i metodi) della classe di test vengono ripetuti per ogni tupla di assegnamenti indicata in un array statico annotato @Parameters

```
@Parameters
    public static Collection<Integer[]> getParam() {
        return Arrays.asList(new Integer[][] {
           // a, b, expected
            { 1, 1, 2 },
            { 3, 2, 5 },
            { 4, 3, 7 },
        });
    }
```

Le tuple sono passate ad ogni nuova istanza del costruttore della classe di test

```
public MyTestClass(int a, int b, int expected) {…}
```

Vediamo un esempio: Consumer.TestSumTwiceBDD