UNIVERSITÀ
DEGLI STUDI DI BARI
ALDO MORO

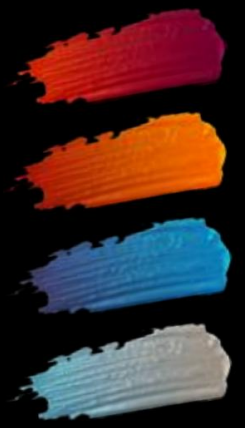
SERLAB
Software Engineering Research

Integrazione e Test di Sistemi Software

Test basati sulle specifiche

Azzurra Ragone

Dipartimento di Informatica - Università degli Studi di Bari
Via Orabona, 4 - 70125 - Bari
Tel: +39.080.5443270 | Fax: +39.080.5442536
serlab.di.uniba.it



Test basati sulle specifiche: riepilogo

7 passaggi per creare la suite di test:

1. Comprensione dei requisiti (cosa deve fare il programma, input e output)

2. Esplorare cosa fa il programma per vari input 3. Esplorare input, output e identificare le partizioni 4. Identificare i casi limite (ovvero i casi limite)
5. Ideare casi di test 6. Automatizzare i casi di test 7. Arricchire la suite di test con creatività ed esperienza

1) Comprendere i requisiti

Leggere attentamente i requisiti:

- cosa dovrebbe fare o NON fare il programma?
- quali sono gli input e gli output?
- i tipi di variabili coinvolte (interi, stringhe, ecc.)
- dominio di input (es. 5<num<10)
- alcuni potrebbero essere impliciti (suscitali!)
- scrivilo!

2) Esplora cosa fa il programma per vari input

Particolarmente importante se non sei tu a scrivere il codice

Costruisci un modello mentale del programma

Gioca con il programma: testalo per diversi input

3) Esplorare input, output e identificare le partizioni

Identificare le partizioni:

- Esaminare ogni input singolarmente:
  - identificare il **tipo** di input (int, stringa, ecc.)
  - **intervallo** di valori (positivo, negativo, tra due valori, ecc.)
  - valori **speciali** (es. Null)
- Osserva le dipendenze tra le variabili: come interagiscono tra loro
- Esplora i possibili tipi di output

4) Identificare i casi limite (detti anche casi limite)

Gli insetti amano i confini!

Identificare i confini di tutte le partizioni.

5) Ideare casi di test

Testare **tutte le combinazioni** di input potrebbe essere costoso (e talvolta non è possibile).

Ridurre il numero di combinazioni.

Testare **i comportamenti eccezionali solo una volta** (non combinarli).

6) Automatizzare i casi di test

Scrivi il test in **JUnit**

Identificare **input** concreti e sapere cosa aspettarsi come **output**

Scrivere test equivale a scrivere codice: i test devono essere facili da leggere e da capire

Dovrebbe essere facile capire quale test non è riuscito e perché

7) Arricchire la suite di test con creatività ed esperienza

Eseguire alcuni **controlli finali**

Rivedi tutti i test che hai creato per vedere se ti mancano alcuni casi

Esempio: sommare due numeri

Implementa il metodo add(): riceve due numeri, sinistro e destro (ciascuno rappresentato come un elenco di cifre), li somma e restituisce il risultato come un elenco di cifre.

Esempi:

- [4,3] + [2,1] = [6,4]
- [2,5] + [1,8] = [4,3]

Requisiti:

- Ogni elemento dovrebbe essere un numero compreso tra [0 e 9]
- Viene generata un'eccezione IllegalArgumentException se questa precondizione non è soddisfatta non tenere

Esempio: sommare due numeri

```
public List<Integer> add(List<Integer> left, List<Integer> right) {
  if (left == null || right == null)
    return null;

  Collections.reverse(left);
  Collections.reverse(right);

  LinkedList<Integer> result = new LinkedList<>();

  int carry = 0;

  for (int i = 0; i < max(left.size(), right.size()); i++) {

    int leftDigit = left.size() > i ? left.get(i) : 0;
    int rightDigit = right.size() > i ? right.get(i) : 0;

    if (leftDigit < 0 || leftDigit > 9 ||
     rightDigit < 0 || rightDigit > 9)
      throw new IllegalArgumentException();

    int sum = leftDigit + rightDigit + carry;

    result.addFirst(sum % 10);

    carry = sum / 10;
  }

  return result;
}
```

Returns null if left or right is null

Reverses the numbers so the least significant digit is on the left

Diamo un'occhiata al codice per 5 minuti e proviamo a individuare i bug

While there is a digit, keeps summing, taking carries into consideration

Throws an exception if the pre-condition does not hold

Sums the left digit with the right digit with the possible carry

The digit should be a number between 0 and 9. We calculate it by taking the rest of the division (the % operator) of the sum by 10.

If the sum is greater than 10, carries the rest of the division to the next digit

Input individuali

parametro sinistro :

1 - Vuoto
2 - Nullo
3 - Cifra singola
4 - Cifre multiple
5 - Zeri a sinistra

parametro giusto :

1 - Vuoto
2 - Nullo
3 - Cifra singola
4 - Cifre multiple
5 - Zeri a sinistra

Combinazioni di input

(sinistra, destra) parametri:

1 - lunghezza (elenco di sinistra) > lunghezza (elenco di destra)

2 - lunghezza (elenco di sinistra) < lunghezza (elenco di destra)

3 - lunghezza (elenco di sinistra) = lunghezza (elenco di destra)

Ci stiamo perdendo qualcosa?

Anche se non esplicitamente indicato nella documentazione: dovremmo testare i casi con **"carry"**

[2,5] + [1,8] = [4,3]

Non basta analizzare i parametri (e le loro combinazioni): è necessario avere anche una conoscenza approfondita del dominio.

Caso speciale di prova: riporto

– Somma senza riporto
– Somma con riporto: un riporto all'inizio –
Somma con riporto: un riporto al centro –
Somma con riporto: molti riporti –
Somma con riporto: molti riporti, non consecutivi
– Somma con riporto: riporto propagato a una nuova cifra (più significativa) (es. 99 + 1 = 100) [caso limite]

Progettare casi di test

Decidere pragmaticamente quali partizioni devono essere combinate con altre e quali no

Testare i casi eccezionali solo una volta e non combinarli (ad esempio null, vuoto, una sola cifra):

T1: la sinistra è nulla

T2: la sinistra è vuota

T3: il diritto è nullo

T4: la destra è vuota

T5: cifra singola, nessun riporto

T6: cifra singola, con riporto

Progettare casi di test

Combinazioni di input

Cifre multiple,

lunghezza (elenco a sinistra) = lunghezza (elenco a destra)

T7: nessun trasporto (22 + 33)

T8: riportare la cifra meno significativa (29 + 23)

T9: riporto al centro (293 + 183)

T10: molti carry (179 + 268)

T11: molti carry, non di fila (19171 + 18161)

T12: riporto propagato a una nuova cifra (ora la più significativa) (998 + 172)

Progettare casi di test

Combinazioni di input

Cifre multiple,

lunghezza (elenco a sinistra) > lunghezza (elenco a destra)

OPPURE lunghezza (elenco di sinistra) < lunghezza (elenco di destra)

T13: divieto di trasporto

T14: riportare la cifra meno significativa

T15: portare al centro

T16: molti trasporti

T17: molti carry, non di fila

T18: riporto propagato a una nuova cifra (ora la più significativa)

Progettare casi di test

Casi speciali

Zeri a sinistra (due casi sono sufficienti):

T19: divieto di porto

T20: portare

Confini:

T21: riporto di uno alla nuova cifra più significativa (es. 99 +1).

Test parametrizzato

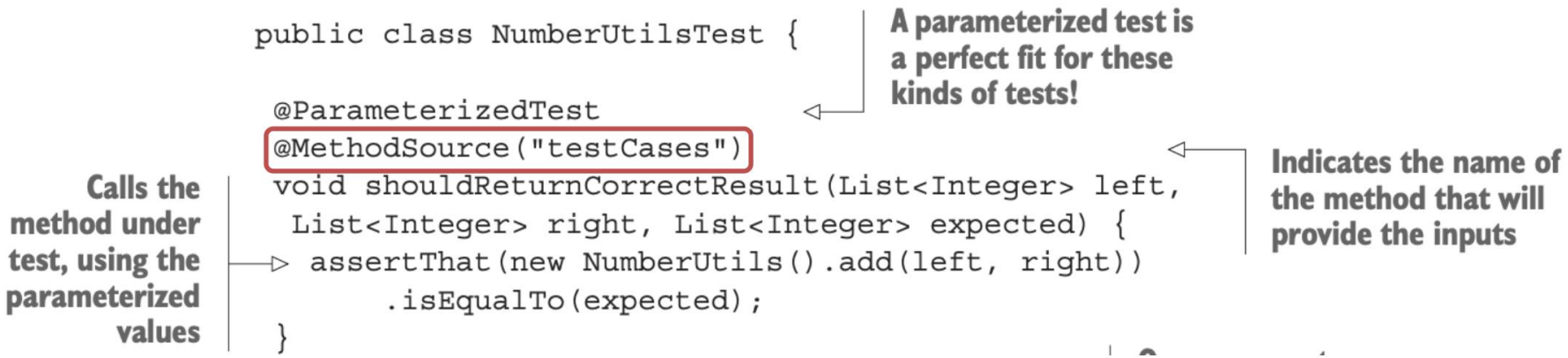
Utilizzeremo la funzionalità ParameterizedTest di JUnit:

- Scrivere un metodo di test shouldReturnCorrectResult() che funzioni come uno scheletro, con variabili invece di valori codificati

- Scrivere un metodo testCases() che fornisca input a shouldReturnCorrectResult()

- Il collegamento tra i due metodi avviene tramite l'annotazione @MethodSource

Test parametrizzato



Test parametrizzato

```
static Stream<Arguments> testCases() {

  return Stream.of(
    of(null, numbers(7,2), null), // T1
    of(numbers(), numbers(7,2), numbers(7,2)), // T2
    of(numbers(9,8), null, null), // T3
    of(numbers(9,8), numbers(), numbers(9,8 )), // T4

    of(numbers(1), numbers(2), numbers(3)), // T5
    of(numbers(9), numbers(2), numbers(1,1)), // T6
```

One argument per test case

Tests with nulls and empties

Tests with single digits

T1: la sinistra è nulla

T2: la sinistra è vuota

T3: il diritto è nullo

T4: la destra è vuota

T5: cifra singola, nessun riporto

T6: cifra singola, con riporto

Test parametrizzato

```
static Stream<Arguments> testCases() {

  return Stream.of(
    of(null, numbers(7,2), null), // T1
    of(numbers(), numbers(7,2), numbers(7,2)), // T2
    of(numbers(9,8), null, null), // T3
    of(numbers(9,8), numbers(), numbers(9,8 )), // T4

    of(numbers(1), numbers(2), numbers(3)), // T5
    of(numbers(9), numbers(2), numbers(1,1)), // T6

    of(numbers(2,2), numbers(3,3), numbers(5,5)), // T7
    of(numbers(2,9), numbers(2,3), numbers(5,2)), // T8
    of(numbers(2,9,3), numbers(1,8,3), numbers(4,7,6)), // T9
    of(numbers(1,7,9), numbers(2,6,8), numbers(4,4,7)), // T10
    of(numbers(1,9,1,7,1), numbers(1,8,1,6,1),
      numbers(3,7,3,3,2)), // T11
    of(numbers(9,9,8), numbers(1,7,2), numbers(1,1,7,0)), // T12
```

One argument per test case

Tests with nulls and empties

Tests with single digits

Tests with multiple digits

lunghezza (elenco di sinistra) = lunghezza (elenco di destra)

T7: nessun trasporto (22 + 33)

T8: riportare la cifra meno significativa (29 + 23)

T9: riporto al centro (293 + 183)

T10: molti carry (179 + 268)

T11: molti carry, non di fila (19171 + 18161)

T12: riporto propagato a una nuova cifra (ora la più significativa) (998 + 172)

Test parametrizzato

Cifre multiple,

lunghezza (elenco a sinistra) > lunghezza (elenco a destra)

OPPURE lunghezza (elenco di sinistra) < lunghezza (elenco di destra)

T13: divieto di trasporto

T14: riportare la cifra meno significativa

T15: portare al centro

Tests with multiple digits, different length, with and without carry (from both sides)

```
of(numbers(2,2), numbers(3), numbers(2,5)), // T13.1
of(numbers(3), numbers(2,2), numbers(2,5)), // T13.2
of(numbers(2,2), numbers(9), numbers(3,1)), // T14.1
of(numbers(9), numbers(2,2), numbers(3,1)), // T14.2
of(numbers(1,7,3), numbers(9,2), numbers(2,6,5)), // T15.1
of(numbers(9,2), numbers(1,7,3), numbers(2,6,5)), // T15.2
```

Test parametrizzato

Cifre multiple,

lunghezza (elenco a sinistra) > lunghezza (elenco a destra)

OPPURE lunghezza (elenco di sinistra) < lunghezza (elenco di destra)

T16: molti trasporti

T17: molti carry, non di fila

T18: riporto propagato a una nuova cifra (ora la più significativa)

Tests with multiple digits, different length, with and without carry (from both sides)

```
of(numbers(3,1,7,9), numbers(2,6,8), numbers(3,4,4,7)), // T16.1
of(numbers(2,6,8), numbers(3,1,7,9), numbers(3,4,4,7)), // T16.2
of(numbers(1,9,1,7,1), numbers(2,1,8,1,6,1),
  numbers(2,3,7,3,3,2)), // T17.1
of(numbers(2,1,8,1,6,1), numbers(1,9,1,7,1),
  numbers(2,3,7,3,3,2)), // T17.2
of(numbers(9,9,8), numbers(9,1,7,2), numbers(1,0,1,7,0)), // T18.1
of(numbers(9,1,7,2), numbers(9,9,8), numbers(1,0,1,7,0)), // T18.2
```

Test parametrizzato

Zeri a sinistra:

T19: divieto di porto

T20: portare

Confini:

T21: riporto di uno alla nuova cifra più significativa (es. 99 +1).

Tests with zeroes on the left

```
        of(numbers(0,0,0,1,2), numbers(0,2,3), numbers(3,5)), // T19
        of(numbers(0,0,0,1,2), numbers(0,2,9), numbers(4,1)), // T20

        of(numbers(9,9), numbers(1), numbers(1,0,0)) // T21
    );
}
```

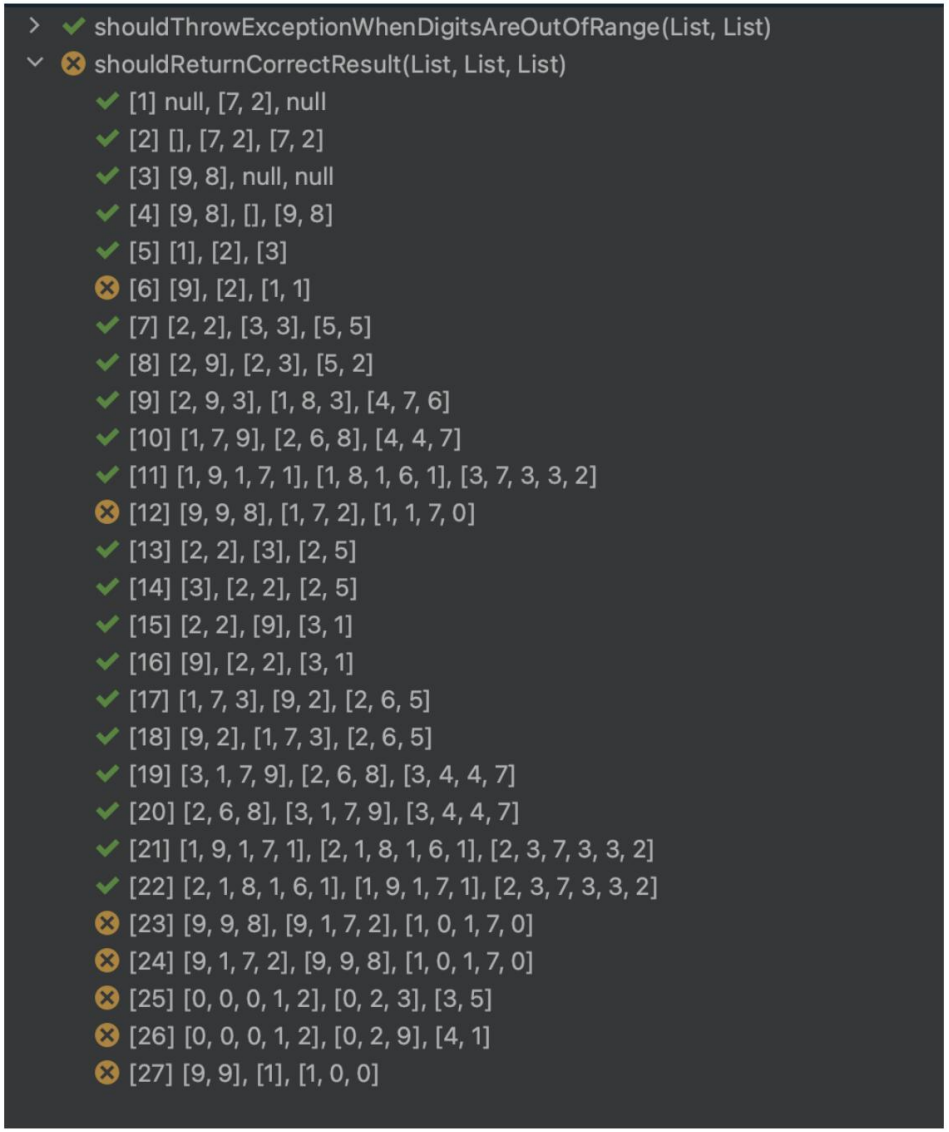
The boundary test

Risultati dei test

Test che falliscono:

- Tutti i test che riguardano un riporto che diventa una nuova cifra più a sinistra falliscono (es. 9 + 2 = 11 restituisce 1 o 998 + 172 = 1170 restituisce 170)

- Il test con zeri a sinistra (risultato previsto [3,5], risultato restituito [0,0,0,3,5]) fallisce



Correzione dei bug

- Semplice correzione per il bug del carry: aggiungere il carry alla fine

```
int carry = 0;
for (int i = 0; i < Math.max(left.size(), right.size()); i++) {

    int leftDigit = left.size() > i ? left.get(i) : 0;
    int rightDigit = right.size() > i ? right.get(i) : 0;

    if (leftDigit < 0 || leftDigit > 9 || rightDigit < 0 || rightDigit > 9)
        throw new IllegalArgumentException();

    int sum = leftDigit + rightDigit + carry;

    result.addFirst( e: sum % 10);
    carry = sum / 10;
}

// if there's some leftover carry, add it to the final number
if (carry > 0)
    result.addFirst(carry);
```

- Semplice correzione per il bug degli zeri a sinistra: rimuovere gli zeri a sinistra prima di restituire il risultato

- Es. sinistra = [0,0,0,1,2]
  destra = [0,2,3]
  risultato atteso [3,5]
  risultato restituito [0,0,0,3,5].

Correzione dei bug

- Semplice correzione per il bug degli zeri a sinistra: rimuovere gli zeri a sinistra prima di restituire il risultato

```
for (int i = 0; i < Math.max(left.size(), right.size()); i++) {

    int leftDigit = left.size() > i ? left.get(i) : 0;
    int rightDigit = right.size() > i ? right.get(i) : 0;

    if (leftDigit < 0 || leftDigit > 9 || rightDigit < 0 || rightDigit > 9)
        throw new IllegalArgumentException();

    int sum = leftDigit + rightDigit + carry;

    result.addFirst( e: sum % 10);
    carry = sum / 10;
}

// if there's some leftover carry, add it to the final number
if (carry > 0)
    result.addFirst(carry);

// remove leading zeroes from the result
while (result.size() > 1 && result.get(0) == 0)
    result.remove( index: 0);

return result;
```

Ex.:
[0,0,0,3,5]

Ci stiamo perdendo qualcosa?

Requisiti:

- Ogni elemento dovrebbe essere un numero compreso tra [0 e 9]
- Viene generata un'eccezione IllegalArgumentException se questa precondizione non è soddisfatta

```
@ParameterizedTest                                  <-- A parameterized test also fits well here.
@MethodSource("digitsOutOfRange")
void shouldThrowExceptionWhenDigitsAreOutOfRange(List<Integer> left,
  ➥ List<Integer> right) {
  assertThatThrownBy(() -> new NumberUtils().add(left, right))
      .isInstanceOf(IllegalArgumentException.class);      <-- Asserts that an exception happens

}

static Stream<Arguments> digitsOutOfRange() {           <-- Passes invalid arguments
  return Stream.of(
      of(numbers(1,-1,1), numbers(1)),
      of(numbers(1), numbers(1,-1,1)),
      of(numbers(1,10,1), numbers(1)),
      of(numbers(1), numbers(1,11,1))
  );
}
```


Lezioni apprese

Il bug riscontrato in questo esempio non è dovuto a un codice errato, ma alla *mancanza di codice.*

Si tratta del tipo di bug scoperti tramite test basati sulle specifiche.

In questo caso, imporre la “copertura del codice” sarebbe stato inutile.

Lezioni da asporto

ÿ *I requisiti* sono molto utili per generare test

ÿ È importante conoscere lo *spazio del dominio* e come interagiscono le variabili

ÿ Scegli l'input più semplice (ad esempio, scegli un piccolo valore int o una stringa breve)

ÿ Segui l' *approccio in 7 passaggi*

ÿ Ricorda che gli insetti amano *i confini*

ÿ Se il numero di casi di test è troppo grande, dovresti decidere cosa è

vale la pena testare e cosa no (quale sarebbe il costo di un fallimento?)

ÿ Utilizzare *test parametrizzati* quando i test hanno lo stesso scheletro

Casi limite: Q1

Condizione: valore > 100 Quali

sono i punti di accensione e spegnimento?

Tempo: 1 minuto

Diapositiva per gentile concessione di Mauricio Aniche

Casi limite: Q1

Condizione: valore > 100 Quali

sono i punti di accensione e spegnimento?

Sul punto = 100 (sempre il valore nella condizione)

Il punto on rende la condizione falsa ((100 > 100) == false), quindi il punto off dovrebbe rendere la condizione vera.

Fuori punto = 101

Diapositiva per gentile concessione di Mauricio Aniche

Casi limite: Q2

Condizione: valore >= 101 Quali

sono i punti di accensione e spegnimento?

Tempo: 1 minuto

Diapositiva per gentile concessione di Mauricio Aniche

Casi limite: Q2

Condizione: valore >= 101 Quali

sono i punti di accensione e spegnimento?

Sul punto = 101 (sempre il valore nella condizione)

Il punto on rende vera la condizione ((101 >= 101) == true), quindi il punto off dovrebbe rendere falsa la condizione.

Fuori punto = 100

Diapositiva per gentile concessione di Mauricio Aniche

Casi limite: Q3

Condizione: valore == 100 Quali

sono i punti di accensione e spegnimento?

Tempo: 1 minuto

Diapositiva per gentile concessione di Mauricio Aniche

Casi limite: Q3

Condizione: valore == 100 Quali

sono i punti di accensione e spegnimento?

Sul punto = 100 (sempre il valore nella condizione)

Il punto preciso rende vera la condizione.

Ci sono due punti fuori strada!

Fuori punto = 101 e 99

Diapositiva per gentile concessione di Mauricio Aniche

Casi limite: Q4

Condizione: valore > n + 1 Quali

sono i punti di accensione e spegnimento?

Tempo: 2 minuti

Diapositiva per gentile concessione di Mauricio Aniche

Casi limite: Q4

Condizione: valore > n + 1 Quali

sono i punti di accensione e spegnimento?

Sul punto = n + 1 (sempre il valore nella condizione)

Il punto di contatto rende la condizione falsa (n + 1 > n + 1 == falso).

Il punto di partenza dovrebbe rendere vera la condizione

Punto di partenza = (n + 1)

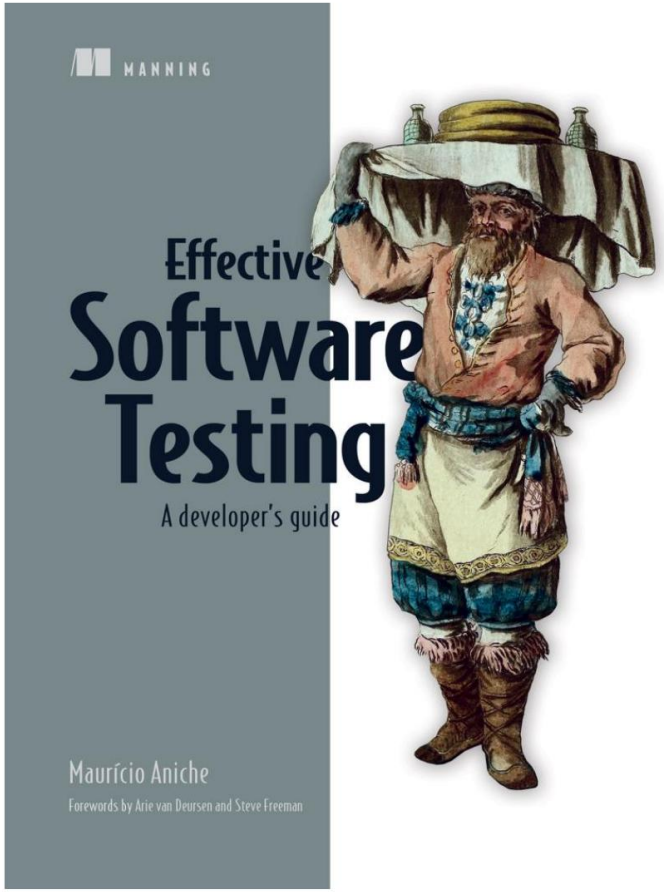
+ 1 Se n è un parametro di input, allora dovresti scegliere qualsiasi "n" e poi "n+1".

Diapositiva per gentile concessione di Mauricio Aniche

Libro di riferimento:

Test software efficaci. Guida per sviluppatori. Mauricio Aniche. Ed. Manning. **(Capitolo** 2)

Utilizza il codice sconto "au35ani" per uno sconto del 35% sul prezzo.



Riferimenti:

- AssertJ - libreria Java per asserzioni fluide: https://assertj.github.io/doc/
- Javadoc di base di Assertj: https://www.javadoc.io/doc/org.assertj/assertj-core/latest/index.html
- Assertj core javadoc: Asserzioni: https://www.javadoc.io/doc/org.assertj/assertj-core/latest/org/assertj/core/api/Assertions.html
- Introduzione ad AssertJ: https://www.baeldung.com/introduction-to-assertj

Azzurra Ragone

Dipartimento di Informatica - Piano VI - Stanza 616 Email: azzurra.ragone@uniba.it