ITSS-2024 HOMEWORK

Corso: Integrazione e Test di Sistemi Software

Autori:

Nome: Sante Dormio

E-mail: s.dormio3@studenti.uniba.it

Matricola: 745566

Nome: Giacinto Lucarelli

E-mail: g.lucarelli16@studenti.uniba.it

Matricola: 697872

Specification Based-Testing (Homework 1)

Codice Testato

```java
package es1_Anagramma_NumeroArmstrong;
import java.util.Arrays;


public class Anagramma_NArmstrong {

	public boolean sonoAnagrammi(String str1, String str2) {
		if (str1 == null || str2 == null) {
			throw new IllegalArgumentException("Le stringhe non possono essere nulle");
		}
		String cleanedStr1 = str1.replaceAll("\\s", "").toLowerCase();
		String cleanedStr2 = str2.replaceAll("\\s", "").toLowerCase();
		if (cleanedStr1.length() != cleanedStr2.length()) {
			return false;
		}
		char[] chars1 = cleanedStr1.toCharArray();
		char[] chars2 = cleanedStr2.toCharArray();
		Arrays.sort(chars1);
		Arrays.sort(chars2);
		return Arrays.equals(chars1, chars2);
	}
	public boolean isNumeroArmstrong(Integer numero) {

		String numeroStr = Integer.toString(numero);
		int lunghezza = numeroStr.length();

		int somma = 0;
		int n = numero;
		while (n > 0) {
			int cifra = n % 10;
			somma += Math.pow(cifra, lunghezza);
			n /= 10;
		}

		return somma == numero;
	}
}

```

Figura 1: Classe Anagramma_NArmstrong contente i metodi testati sonoAnagrammi() e isNumeroArmstrong()

Step 1. Comprensione dei requisiti

L'obiettivo della funzione **sonoAnagrammi()** è verificare se due stringhe sono anagrammi, ovvero se possono essere ottenute l'una dall'altra, riarrangiando i loro caratteri.

La sintassi è: **public boolean sonoAnagrammi**(String str1, String str2)

- **Parametri**:
  - **String str1**, stringa da controllare mettendola a confronto con la str2;
  - **String str2**, stringa che dovrebbe essere anagrammata della str1, che verrà controllata;
- **Return**:
  - Ritorna un **boolean** "vero" se i caratteri delle due stringhe sono uguali e quindi l'anagramma sarà corretto, altrimenti ritorna "false";
- **Eccezioni**:
  - **IllegalArgumentException**: se una o entrambe le stringhe sono nulle;

L'obiettivo della funzione **isNumeroArmstrong()** è verificare se il numero passato come parametro è un numero di Armstrong ovvero se la somma delle potenze delle cifre di un numero è uguale al numero originale.

La sintassi è: **public boolean isNumeroArmstrong**(Integer numero)

- **Parametri**:
  - **Integer numero**, numero da dover controllare calcolandone la somma delle potenze delle cifre che lo compongono;
- **Return**:
  - Ritorna un **boolean** somma che conterrà "vero" se sarà uguale al numero passato come parametro, falso altrimenti;
- **Eccezioni**:
  - **IllegalArgumentException**, verrà lanciata se si inserisce un numero negativo;

```java
@Order(1)
@ParameterizedTest
@DisplayName("Explore what the function does with casual input for string to Anagrammi")
@MethodSource("exploreStrings")
void shouldReturnCorrectResultExploreStrings(String str1, String str2, boolean expected) {
    assertThat(anagrammaNumeroArmstrong.sonoAnagrammi(str2, str2), is(equalTo(expected)));
}

static Stream<Arguments> exploreStrings(){
    return Stream.of(
            Arguments.of("listen", "silent", true),
            Arguments.of("LiSten", "sIlEnT", true),
            Arguments.of(null, null, false),
            Arguments.of("123456".length(), "2345678".length(), true)
            );
}
```

Figura 2: metodo shouldReturnCorrectResultExploreStrings() della classe TestAnagramma_NArmstrong();

```java
@Order(2)
@ParameterizedTest
@DisplayName("Explore what the function does with casual input of number for Armstrong")
@MethodSource("exploreNumbers")
void shouldReturnCorrectResultExploreNumbers(Integer numero, boolean expected) {
    assertThat(anagrammaNumeroArmstrong.isNumeroArmstrong(numero), is(equalTo(expected)));
}

static Stream<Arguments> exploreNumbers(){
    return Stream.of(
            Arguments.of(153, true),
            Arguments.of(null, false)
            );
}
```

Figura 3: shouldReturnCorrectResultExploreNumbers() della classe TestAnagramma_NArmstrong();

Step 2. Esplorare cosa fa la funzione per vari input

Si nota in particolare nel metodo **sonoAnagrammi()** che:

- **Str1** e **Str2** possono essere inserite con caratteri maiuscoli e minuscoli, ma verranno sempre sostituiti con caratteri minuscoli nelle variabili **CleanedStr1** e **CleanedStr2**;
- **Str1** e **Str2** non possono essere nulle come stringhe;
- **CleanedStr1** e **CleanedStr2** devono avere la stessa lunghezza;

Si nota in particolare nel metodo **isNumeroArmstrong()** che:

- L'Integer **numero** viene convertito in una stringa in **numeroStr**;
- L'Integer **numero** non può essere null;

Step 3. Input, output ed identificazione delle partizioni

- Classi di Input (Individuali):
    - **Str1**:
        - Null;
        - Stringhe con spazi e senza spazi;
        - Stringhe con Caratteri maiuscoli e minuscoli;
        - Lunghezza = 1;
        - Lunghezza > 1;
        - Lunghezza = 0 (Stringa vuota);

    - **Str2**:
        - Null;
        - Stringhe con spazi e senza spazi;
        - Stringhe con Caratteri maiuscoli e minuscoli;
        - Lunghezza = 1;
        - Lunghezza > 1;
        - Lunghezza = 0 (Stringa vuota);
        - Lunghezza =! Da Str1;
        - Lunghezza = a Str1;
    - **Numero**:
        - Numero intero in variabile Integer;
        - Numero < 0;
        - Numero > = 0;
        - Numero = null;
- Combinazioni di Input:
    - **Str2** contiene le stesse lettere di **Str1** in ordine diverso**;**
    - **Str2** non contiene le stesse lettere di **Str1** in qualsiasi ordine**;**
    - **Str1 e Str2** contengono spazi e caratteri maiuscoli e minuscoli e caratteri speciali;
    - **Numero** è un numero intero in variabile Integer;
- Classi di Output Attesi:
    - Variabile booleana “Vera”;
    - Variabile booleana “Falsa”;

Step 4. Identificare i Boundary Case

Per la funzione sonoAnagrammi():

1. **Stringhe vuote:**
    - str1 e/o str2 sono stringhe vuote (""). Questo potrebbe testare se la funzione gestisce correttamente le stringhe vuote.
2. **Stringhe con un solo carattere:**
    - str1 e/o str2 contengono solo un carattere. Verificare se il programma gestisce correttamente casi di input con stringhe molto corte.
3. **Stringhe identiche:**
    - str1 e str2 sono identiche. Questo testa il comportamento della funzione quando entrambe le stringhe sono le stesse.

Per la funzione isNumeroArmstrong():

1. **Numero singola cifra:**
    - Un numero compreso tra 0 e 9. Verifica se il programma gestisce correttamente numeri a singola cifra.
2. **Numeri di Armstrong:**
    - Numeri di Armstrong noti (ad esempio, 0, 1, 153, 370, 407, etc.). Ci assicuriamo che la funzione restituisca “**true**” per numeri noti di Armstrong.
3. **Numeri non di Armstrong:**
    - Numeri che non sono numeri di Armstrong (ad esempio, 15, 100, 200, etc.). Ci assicuriamo che la funzione restituisca “**false**” per numeri che non sono numeri di Armstrong.

4. **Limiti del numero:**
    - Il numero massimo e minimo supportato dal tipo di dato (ad esempio, per Integer in Java, i valori sono Integer.MAX_VALUE e Integer.MIN_VALUE). Verifica come gestisce la funzione numeri ai limiti consentiti.
5. **Numero negativo:**
    - Testa la gestione dei numeri negativi, poiché la definizione di numero di Armstrong richiede numeri non negativi.

Step 5. Definire i Test Case

TestCaseID	Descrizione	Risultato Atteso
T1.	**Str1 =** “listen” e **Str2 =** “silent”	True
T2.	**Str1 =** "eleven plus two" e **Str2 =** "twelve plus one"	True
T3.	**Str1 =** "Race" e **Str2 =** "care"	True
T4.	**Str1 =** "debit card" e **Str2 =** "bad credit"	True
T5.	**Str1 =** "hello" e **Str2 =** "world"	False
T6.	**Str1 =** "astronomer" e **Str2 =** "moon starer"	False
T7.	**Str1 =** "hello" e **Str2 =** null	False
T8.	**Str1 =** null e **Str2 =** null	IllegalArgumentException
T9.	**Str1** = “hello” e **Str2** = null	IllegalArgumentException
T10.	**Str1** = null e **Str2** = “hello”	IllegalArgumentException
T11.	**Boundary Case: Str1 =** “ ”e **Str2 =** “ ”	True (poiché entrambe le stringhe vuote sono considerate anagrammi)
T12.	**Boundary Case: Str1 =** “a” e **Str2 =** “a”	True (stesso carattere considerato anagramma)
T13.	**Boundary Case: Str1 =** "hello" e **Str2 =** "hello"	True (stringhe identiche considerate anagrammi)
T14.	**Numero =** 153	True
T15.	**Numero =** 370	True
T16.	**Numero =** 15	False
T17.	**Numero =** 100	False
T18.	**Boundary case: Numero =** 5	True (numero singola cifra)
T19.	**Boundary case: Numero =** 0	True (0 è considerato numero di Armstrong)
T20.	**Boundary case: Numero =** -1	IllegalArgumentException(Numero negativo)
T21.	**Numero =** Integer.MAX_VALUE (Limite max del numero)	False(non è numero di Armstrong)
T22.	**Numero =** Integer.MIN_VALUE (Limite minimo del numero)	IllegalArgumentException(Numero negativo)
T23.	**Numero =** Null	IllegalArgumentException (Numero = null)

Step 6. Automatizzazione dei Test Case

```
49●   @Order(3)
50    @ParameterizedTest
51    @DisplayName("Sad Path Test Strings")
52    @MethodSource("sadPathTestCasesStrings")
53    void shouldReturnCorrectResultSadPathStrings(String str1, String str2) {
54        if(str1 == null && str2 == null) {
55            assertFalse(new Anagramma_NArmstrong().sonoAnagrammi(str1, str2));
56        }
57    }
58
59●   static Stream<Arguments> sadPathTestCasesStrings(){
60     return Stream.of(
61             Arguments.of(null, null),//T8
62             Arguments.of("hello", null), //T9
63             Arguments.of(null, "hello") //T10
64             );
65    }
```

Figura 4: shouldReturnCorrectResultSadPathStrings() della classe TestAnagramma_NArmstrong();

Si è deciso di utilizzare dei test **parametrici** poiché la struttura dei test, in termini di dati e di asserzioni, si ripeteva. Inoltre, si noti la suddivisione dei test in due funzioni per ogni metodo (2 per sonoAnagrammi() e 2 per isNumeroArmstrong()), una inerente l'Happy Path ed una inerente il Sad Path, ciò al fine di una migliore organizzazione del Report dei Test.

Il metodo **shouldReturnCorrectResultSadPathStrings()** analizza i casi di test in cui si verificano eccezioni per il metodo **sonoAnagrammi()** e questo è:

- T8. **Str1 & Str2** sono null;
- T9. **Str1 & Str2** sono rispettivamente una "hello" e una null;
- T10. **Str1 & Str2** sono rispettivamente una null e una "hello";

```
67●   @Order(4)
68    @ParameterizedTest
69    @DisplayName("Sad Path Test Numbers")
70    @MethodSource("sadPathTestCasesNumbers")
71    void shouldReturnCorrectResultSadPathNumbers(Integer numero) {
72        if(numero<0 && numero == Integer.MIN_VALUE) {
73            assertFalse(new Anagramma_NArmstrong().isNumeroArmstrong(numero));
74        }
75    }
76
77●   static Stream<Arguments> sadPathTestCasesNumbers(){
78     return Stream.of(
79             Arguments.of(-1),//T20
80             Arguments.of(Integer.MIN_VALUE) //T22
81             );
82    }
```

Figura 5: shouldReturnCorrectResultSadPathNumbers() della classe TestAnagramma_NArmstrong();

Il metodo **shouldReturnCorrectResultSadPathNumbers()** analizza i casi di test in cui si verificano eccezioni per il metodo **isNumeroArmstrong()** e questi sono:

- T20. **Numero** è un numero negativo;
- T22. **Integer.MIN_VALUE** restituisce il minor numero inseribile che non è un numero di Armstrong;

```
81●   @Order(5)
82    @ParameterizedTest
83    @DisplayName("Happy Path Test Strings")
84    @MethodSource("happyPathTestCasesStrings")
85    void shouldReturnCorrectResultHappyPathStrings(String str1, String str2, boolean expected) {
86        assertThat(new Anagramma_NArmstrong().sonoAnagrammi(str1, str2), is(equalTo(expected)));
87    }
88
89●   static Stream<Arguments> happyPathTestCasesStrings(){
90        return Stream.of(
91                Arguments.of("listen", "silent", true), //T1
92                Arguments.of("eleven plus two", "twelve plus one", true), //T2
93                Arguments.of("Race", "care", true), //T3
94                Arguments.of("debit card", "bad credit", true), //T4
95                Arguments.of("astronomer", "moon starer", true), //T6
96                Arguments.of(" ", " ", true), //T9 boundary case
97                Arguments.of("a", "a", true), //T10 boundary case
98                Arguments.of("hello", "hello", true) //T11 boundary case
99                );
100   }
101
```

Figura 6: shouldReturnCorrectResultHappyPathStrings() della classe TestAnagramma_NArmstrong();

Il metodo **shouldReturnCorrectResultHappyPathStrings()** include i casi in cui **Str1 e Str2** sono diversi da null, ovvero tutti i restanti casi di Test.

```
102    @Order(6)
103    @ParameterizedTest
104    @DisplayName("Happy Path Test Numbers")
105    @MethodSource("happyPathTestCasesNumbers")
106    void shouldReturnCorrectResultHappyPathNumbers(Integer numero, boolean expected) {
107        assertThat(new Anagramma_NArmstrong().isNumeroArmstrong(numero), is(equalTo(expected)));
108    }
109
110    static Stream<Arguments> happyPathTestCasesNumbers(){
111        return Stream.of(
112                Arguments.of(153, true), //T12
113                Arguments.of(370, true), //T13
114                Arguments.of(5, true), //T16 boundary case
115                Arguments.of(0, true) //T17 boundary case
116                );
117    }
118
```

Figura 7: shouldReturnCorrectResultHappyPathNumbers() della classe TestAnagramma_NArmstrong();

Il metodo **shouldReturnCorrectResultHappyPathNumbers()** include i casi in cui **numero** è diverso da un numero negativo, ovvero tutti i restanti casi di Test.

Step 7. Ampliamento della suite di Test con creatività ed esperienza

Per estendere la suite di test, si sono testati stringhe e numeri particolari, quali stringhe di caratteri speciali o con introduzioni di esse in altre stringhe, o per quanto riguarda i numeri del secondo metodo, il numero di Armstrong più grande calcolabile dal calcolatore.

TestCaseID	Descrizione	Risultato Atteso
T24.	**Str1** è una stringa contente “hello!@#” e **Str2** “!@#olleh” che contengono diversi caratteri speciali	True (Sono Anagrammi)
T25.	**Str1** è una stringa contente “a&b c” e **Str2** “bca& ” contengono diversi caratteri speciali e divisi da spazi	True (Sono Anagrammi)
T26.	**Str1** è una stringa contente “&%$£ ?/” e **Str2** “?% &£$” contenenti due stringhe di soli caratteri speciali con spazi	True (Sono Anagrammi)
T27.	**Numero** = 548834 contiene il più grande numero di Armstrong (tra gli Integer possibili) che il calcolatore può calcolare	True (è un numero di Armstrong)

I Test sono implementati come segue:

```
119●    @Order(7)
120     @ParameterizedTest
121     @DisplayName("Special Characters in Strings Test")
122     @MethodSource("specialCharactersInStringsTestCases")
123     void shouldReturnCorrectResultSpecialCharactersInStrings(String str1, String str2, boolean expected) {
124         assertThat(new Anagramma_NArmstrong().sonoAnagrammi(str1, str2), is(equalTo(expected)));
125     }
126
127●    static Stream<Arguments> specialCharactersInStringsTestCases(){
128         return Stream.of(
129                 Arguments.of("hello!@#", "!@#olleh", true), //T21
130                 Arguments.of("a&b c", "bca& ", true), //T22
131                 Arguments.of("&%$£ ?/", "?% &£$/", true) //T23
132                 );
133     }
134
135●    @Order(8)
136     @ParameterizedTest
137     @DisplayName("Special Numbers of Armstrong Test")
138     @MethodSource("specialNumbersOfArmstrongTestCases")
139     void shouldReturnCorrectResultSpecialNumbersOfArmstrong(Integer numero, boolean expected) {
140         assertThat(new Anagramma_NArmstrong().isNumeroArmstrong(numero), is(equalTo(expected)));
141     }
142
143●    static Stream<Arguments> specialNumbersOfArmstrongTestCases(){
144         return Stream.of(
145                 Arguments.of(548834, true) //T24 numero int di Armstrong più grande calcolabile
146                 );
147     }
148
149 }
150
```

Figura 7: shouldReturnCorrectResultSpecialCharactersInStringsTestCases() & shouldReturnCorrectResultSpeciaNumbersOfArmstrong() della classe TestAnagramma_NArmstrong();

Esecuzione dei Test e Bug Fixing

Esito

Di seguito il report dei test:

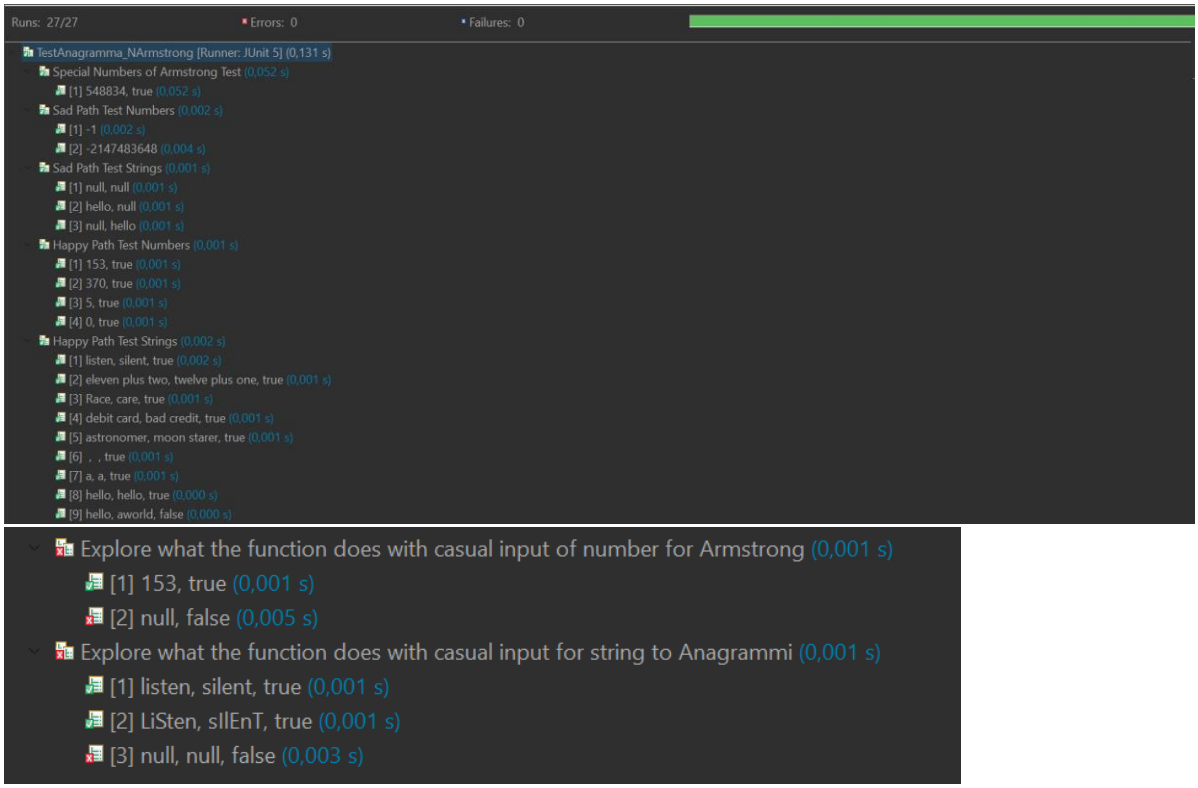


Figura 8: Report di Test con i Test falliti;

Si segnala il fallimento di 2 Test Case, uno per metodo:

TestCaseID	Descrizione	Risultato Atteso	Risultato Ottenuto
T8.	**Str1 & Str2** sono nulle	False	IllegalArgumentException
T21.	**Numero** = null	False	IllegalArgumentException(Il numero non può essere null)

Bug Fixing

TestCaseID	Errore	Bug Fix
T8.	Inserendo nelle due variabili **str1 e str2** due valori nulli, otterrò un lancio di un'eccezione che mi darà l'errore di aver inserito come valori null;	Controllo sull'inserimento dei valori nei parametri, che nel caso siano null, otterrò in questo caso un semplice boolean tipo false, che non li riconosce come anagrammi.

```
7●      public boolean sonoAnagrammi(String str1, String str2) {
8           if (str1 == null || str2 == null) {
9               throw new IllegalArgumentException("Le stringhe non possono essere nulle");
10          }
11
12          String cleanedStr1 = str1.replaceAll("\\s", "").toLowerCase();
13          String cleanedStr2 = str2.replaceAll("\\s", "").toLowerCase();
14
15          if (cleanedStr1.length() != cleanedStr2.length()) {
16              return false;
17          }
18
19          char[] chars1 = cleanedStr1.toCharArray();
20          char[] chars2 = cleanedStr2.toCharArray();
21
22          Arrays.sort(chars1);
23          Arrays.sort(chars2);
24
25          return Arrays.equals(chars1, chars2);
26      }
27
```

Figura 9: Codice Pre Fixing;

```
5  public class Anagramma_NArmstrong {
6
7●      public boolean sonoAnagrammi(String str1, String str2) {
8           if (str1 == null || str2 == null) {
9               return false;
10          }
11
12          String cleanedStr1 = str1.replaceAll("\\s", "").toLowerCase();
13          String cleanedStr2 = str2.replaceAll("\\s", "").toLowerCase();
14
15          if (cleanedStr1.length() != cleanedStr2.length()) {
16              return false;
17          }
18
19          char[] chars1 = cleanedStr1.toCharArray();
20          char[] chars2 = cleanedStr2.toCharArray();
21
22          Arrays.sort(chars1);
23          Arrays.sort(chars2);
24
25          return Arrays.equals(chars1, chars2);
26      }
```

Figura 10: Codice Post Fixing;

T21.	Nel caso si inserisse come valore nel parametro **Numero** il valore null, mi genererà un errore lanciando un eccezione	Aggiunta di un controllo nella funzione isNumeroArmstrong(): con un if si controlla che la variabile numero non sia null, altrimenti ritornerà un boolean false che lo categorizzerà come non numero di Armstrong;

```
28●         public boolean isNumeroArmstrong(Integer numero) {
29
30              String numeroStr = Integer.toString(numero);
31              int lunghezza = numeroStr.length();
32
33              int somma = 0;
34              int n = numero;
35              while (n > 0) {
36                  int cifra = n % 10;
37                  somma += Math.pow(cifra, lunghezza);
38                  n /= 10;
39              }
40
41              return somma == numero;
42          }
43      }
44
```

Figura 11: Codice Pre Fixing;

```
28●         public boolean isNumeroArmstrong(Integer numero) {
29
30              if(numero == null) {
31                  return false;
32              }
33
34              String numeroStr = Integer.toString(numero);
35              int lunghezza = numeroStr.length();
36
37              int somma = 0;
38              int n = numero;
39              while (n > 0) {
40                  int cifra = n % 10;
41                  somma += Math.pow(cifra, lunghezza);
42                  n /= 10;
43              }
44
45              return somma == numero;
46          }
47      }
```

Figura 12: Codice Post Fixing;

Esito finale

Di seguito il report di test dopo il Bug Fixing

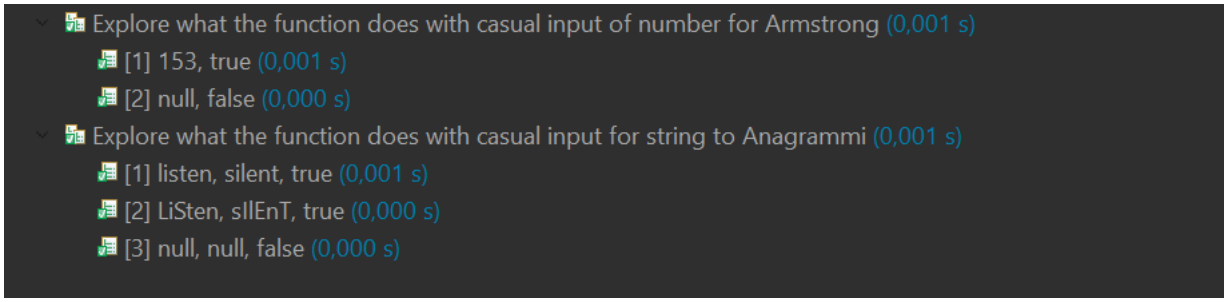


Figura 13: Report di Test con il successo di ogni Test Case;

Black-Box (Specification-Based Testing)

L'analisi del Black-Box si concentra sul comportamento esterno della funzione, senza considerare l'implementazione interna.

1. **Funzione sonoAnagrammi():**
    - **Casi di Test:**
        - Stringhe che sono anagrammi;
        - Stringhe che non sono anagrammi;
        - Stringhe che sono anagrammi con spazi, maiuscole, minuscole e caratteri speciali;
        - Stringhe con una stringa nulla;
        - Stringhe con entrambe le stringhe nulle;
    - **Output Atteso:**
        - **true** se le stringhe sono anagrammi, altrimenti **false**;
        - **false** se una o entrambe le stringhe sono nulle;
2. **Funzione isNumeroArmstrong:**
    - **Casi di Test:**
        - Numeri che sono numeri di Armstrong;
        - Numeri che non sono numeri di Armstrong;
        - Numeri negativi;
        - Caso null;
    - **Output Atteso:**
        - **true** se il numero è un numero di Armstrong, altrimenti **false**;
        - **IllegalArgumentException** se il numero è negativo;
        - **false** se il valore inserito è null;

White-Box (Structural-Based Testing)

L'analisi del white-box si concentra sulla struttura interna e sulla logica dell'implementazione.

1. **Funzione sonoAnagrammi():**
    - **Implementazione:**
        - Rimozione degli spazi e conversione in minuscolo delle stringhe;
        - Controllo delle lunghezze delle stringhe;
        - Ordinamento dei caratteri delle stringhe e confronto;
    - **Ramo di Condizione:**
        - Copertura completa del ramo in cui le stringhe sono nulle;
        - Copertura completa dei rami relativi alle lunghezze delle stringhe;
    - **Gestione degli Anagrammi:**

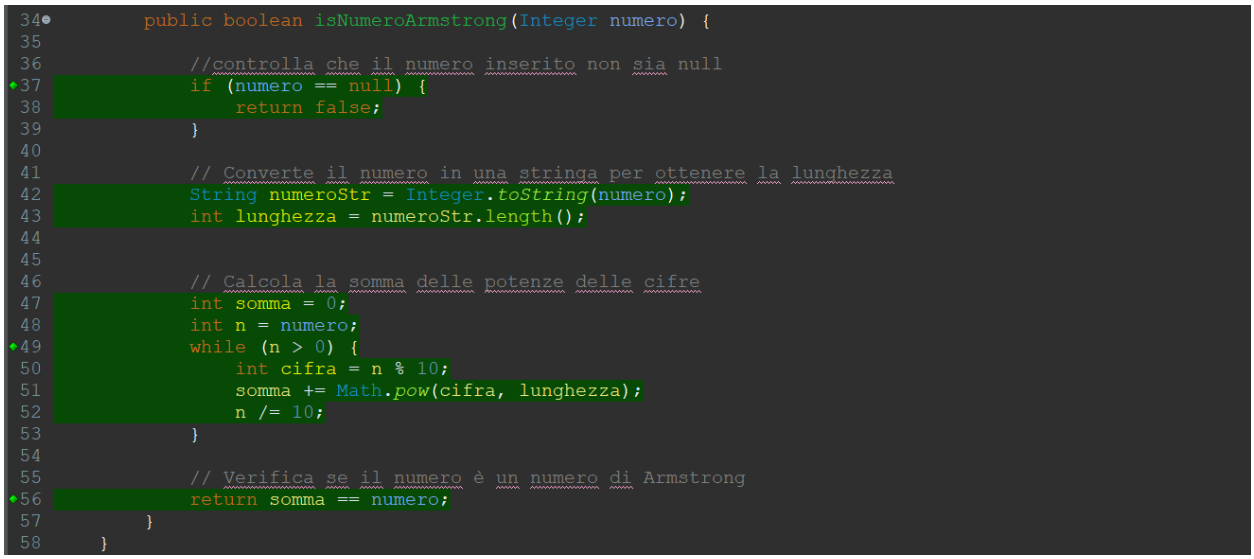
        - Si testa con una varietà di casi per assicurarci che la funzione identifichi correttamente gli anagrammi;
2. **Funzione isNumeroArmstrong():**
    - **Implementazione:**
        - Conversione del numero in una stringa;
        - Calcolo della somma delle potenze delle cifre;
    - **Ramo di Condizione:**
        - Copertura del ramo relativo alla negatività del numero;
        - Copertura del ramo relativo all'inserimento di un valore null;
    - **Gestione dei Numeri di Armstrong:**
        - Si testa con numeri noti di Armstrong e non di Armstrong;
        - Si verifica la corretta gestione dei numeri negativi;
        - Si verifica la corretta gestione dei valori null;

3. **Gestione delle Eccezioni:**
    - **Eccezione IllegalArgumentException:**
        - Si verifica che l'eccezione venga sollevata correttamente quando il numero è negativo.

Code Coverage

```
5  public class Anagramma_NArmstrong {
6
7          public boolean sonoAnagrammi(String str1, String str2) {
8              // Controlla se entrambe le stringhe sono non nulle
9              if (str1 == null || str2 == null) {
10                 return false;
11             }
12
13             // Rimuove spazi e converte le stringhe in minuscolo per la verifica dell'anagramma
14             String cleanedStr1 = str1.replaceAll("\\s", "").toLowerCase();
15             String cleanedStr2 = str2.replaceAll("\\s", "").toLowerCase();
16
17             // Control se le lunghezze delle stringhe sono uguali
18             if (cleanedStr1.length() != cleanedStr2.length()) {
19                 return false;
20             }
21
22             // Verifica se le stringhe sono anagrammi
23             char[] chars1 = cleanedStr1.toCharArray();
24             char[] chars2 = cleanedStr2.toCharArray();
25
26             // Ordina i caratteri delle due stringhe
27             Arrays.sort(chars1);
28             Arrays.sort(chars2);
29
30             // Confronta i caratteri ordinati
31             return Arrays.equals(chars1, chars2);
32         }
33
```

Figura 14: Code Coverage report (Jacoco), sonoAnagrammi();



```
34      public boolean isNumeroArmstrong(Integer numero) {
35
36          //controlla che il numero inserito non sia null
37          if (numero == null) {
38              return false;
39          }
40
41          // Converte il numero in una stringa per ottenere la lunghezza
42          String numeroStr = Integer.toString(numero);
43          int lunghezza = numeroStr.length();
44
45
46          // Calcola la somma delle potenze delle cifre
47          int somma = 0;
48          int n = numero;
49          while (n > 0) {
50              int cifra = n % 10;
51              somma += Math.pow(cifra, lunghezza);
52              n /= 10;
53          }
54
55          // Verifica se il numero è un numero di Armstrong
56          return somma == numero;
57      }
58  }
```

Figura 15: Code Coverage report (Jacoco), isNumeroArmstrong();

La **Code Coverage** indicata dal report di **Jacoco** sviluppata durante lo Specification-Based Test si attesta come copertura al 100%, coprendo del tutto le righe di codice. Si può notare che la suite di Test raggiunga anche la **Path Coverage.**

Condizioni:

1. **sonoAnagrammi()**:
    - Percorso 1: Le stringhe sono nulle;
    - Percorso 2: Le stringhe non sono nulle, ma hanno lunghezze diverse;
    - Percorso 3: Le stringhe non sono nulle, hanno lunghezze uguali, ma non sono anagrammi;
    - Percorso 4: Le stringhe non sono nulle, hanno lunghezze uguali, e sono anagrammi;
2. **isNumeroArmstrong()**:
    - Percorso 1: Il numero è nullo;
    - Percorso 2: Il numero non è nullo, ma non è un numero di Armstrong;
    - Percorso 3: Il numero non è nullo ed è un numero di Armstrong;

Property-Based Testing (Homework 2)

Codice Testato

```java
28●         public boolean isNumeroArmstrong(Integer numero) {
29
30              if(numero == null) {
31                  return false;
32              }
33
34              String numeroStr = Integer.toString(numero);
35              int lunghezza = numeroStr.length();
36
37              int somma = 0;
38              int n = numero;
39              while (n > 0) {
40                  int cifra = n % 10;
41                  somma += Math.pow(cifra, lunghezza);
42                  n /= 10;
43              }
44
45              return somma == numero;
46          }
47      }
```

Figura 16: Classe Anagramma_NArmstrong, metodo testato: isNumeroArmstrong();

Si è scelto di testare il metodo **isNumeroArmstrong()** della classe **Anagramma_NArmstrong** dell'homework1 degli **Specification e Structural-Based Test.**

Progettazione

La **Proprietà** testata è la correttezza della **verifica del Numero Di Armstrong** implementato.

Per tanto si è deciso di randomizzare i numeri da verificare, testando tutti i casi possibili, e vedendone i relativi report, pur notando che nel caso della verifica del "**null**" questo non si sia mai generato, e quindi poteva anche essere omesso.

Come set di dati ci si è concentrati su **parametri** che potessero rappresentare i valori che la funzione dovrà controllare in base ai vari casi, testando la proprietà su numeri generati casualmente compresi tra qualsiasi numero negativo con minimo in **Integer.MIN_VALUE** e qualsiasi numero positivo con massimo in **Integer.MAX_VALUE** compreso il caso dello 0.

Implementazione

```java
1  package es1_Anagramma_NumeroArmstrong;
2
3● import net.jqwik.api.*;
4  import net.jqwik.api.constraints.IntRange;
5  import net.jqwik.api.statistics.Histogram;
6  import net.jqwik.api.statistics.Statistics;
7  import net.jqwik.api.statistics.StatisticsReport;
8
9  class PBT {
10
11      Anagramma_NArmstrong nA = new Anagramma_NArmstrong();
12
13●     @Property
14      @StatisticsReport(format = Histogram.class)
15      void shouldReturnCorrectResultPropertyBased(@ForAll @IntRange(min = Integer.MIN_VALUE, max = Integer.MAX_VALUE)Integer numero) {
16          String range = nA.isNumeroArmstrong(numero) == true ? "Numeri Di Armstrong" : "Numeri Non di Armstrong";
17          String r2 = numero == null ? "Numeri Generati Null" : "Numeri Generati Non Null";
18          String r3 = numero < 0 ? "Numeri Negativi Generati (Non Armstrong)" : "Numeri Positivi Generati";
19          Statistics.label("Statistiche").collect(range).collect(r2).collect(r3);
20          Statistics.label("value").collect(numero);
21          }
22
23 }
```

Figura 17: metodo shouldReturnCorrectResultPropertyBased(), implementa i test Property per la funzione isNumeroArmstrong();

Oltre a porre i **vincoli** sopra stabiliti, si è posto anche:

- Un vincolo di accettazione e controllo del test per i **numeri generati con valore "Null"**;
- Un vincolo di accettazione e controllo del test per i **numeri negativi**;

Per Verificare che effettivamente il test sia stato effettuato correttamente utilizziamo le annotazioni forniteci dalla libreria **net.jqwik.api.Statistics** in particolare con .**Histogram, .Statistics e .statisticsReport,** che segnaleranno i casi di positività, negatività e di boundary case, generati randomicamente.

Risultati

```
timestamp = 2023-12-04T12:16:00.516405300, [PBT:shouldReturnCorrectResultPropertyBased] (3000) Statistiche =
      # |                                      label | count |
    ----|--------------------------------------------|-------|----------------------------------------------------------------------------------------------------
      0 |                        Numeri Di Armstrong |    67 | #####
      1 |                   Numeri Generati Non Null |  1000 | ################################################################################
      2 | Numeri Negativi Generati (Non Armstrong) |   472 | ####################################
      3 |                    Numeri Non di Armstrong |   933 | #######################################################################
      4 |                   Numeri Positivi Generati |   528 | ########################################

timestamp = 2023-12-04T12:16:00.528402300, PBT:shouldReturnCorrectResultPropertyBased =
                                |------------------------jqwik------------------------
tries = 1000                    | # of calls to property
checks = 1000                   | # of not rejected calls
generation = RANDOMIZED         | parameters are randomly generated
after-failure = SAMPLE_FIRST    | try previously failed sample, then previous seed
when-fixed-seed = ALLOW         | fixing the random seed is allowed
edge-cases#mode = MIXIN         | edge cases are mixed in
edge-cases#total = 9            | # of all combined edge cases
edge-cases#tried = 9            | # of edge cases tried in current run
seed = 8398399545459799855      | random seed to reproduce generated values
```

Figura 18: statistiche per l'esecuzione della funzione di test shouldReturnCorrectResultPropertyBased();

```
timestamp = 2023-12-04T12:17:11.827788200, [PBT:shouldReturnCorrectResultPropertyBased] (1000) value =
      # |       label | count |
    ----|-------------|-------|-----------------------------------------------------------------------------
      0 | -2147483648 |     6 | ######
      1 | -2147483647 |     8 | ########
      2 | -2146371111 |     1 | #
      3 | -2046395837 |     1 | #
      4 | -2006635042 |     1 | #
      5 | -2003435692 |     1 | #
      6 | -1996630067 |     1 | #
      7 | -1910404138 |     1 | #
      8 | -1897249644 |     1 | #
      9 | -1885517756 |     1 | #
     10 | -1865083207 |     1 | #
```

Figura 19: Alcuni esempi generati randomicamente fino a 1000 casi dal test, raggruppati cosi dal comando statistics.label("VALUE").collect(numero);

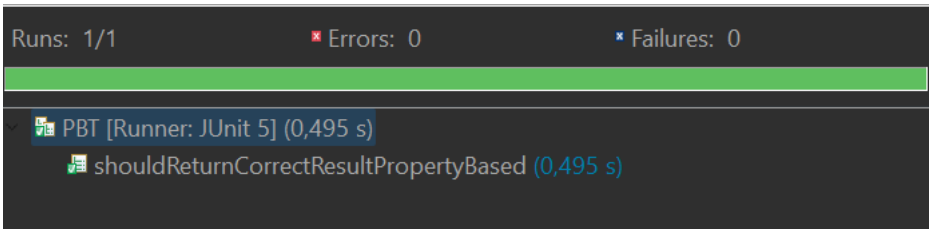


Figura 20: risultato dell'esecuzione dei test tramite la funzione shouldReturnCorrectResultPropertyBased();

Dal report delle statistiche si può osservare una distribuzione dei valori di input **centrata** sui valori stabiliti come:

- "**Numeri generati non Null**" il che rispecchia le nostre aspettative, in quanto nessun numero è stato generato con valore "Null";
- **"Numeri Generati non Di Armstrong"** in quanto in più di un test è stato verificato che la generazione randomica di valori favorisce i casi in cui un numero è non di Armstrong;
- Si può notare come da noi aspettato, che c'è un buon equilibrio tra i numeri generati positivi e negativi, e di conseguenza i **numeri negativi** non saranno numeri di Armstrong mentre i **numeri positivi** lo sono in una piccola parte;

Il report tuttavia ci indica che i numeri che verranno generati casualmente, non saranno mai "**null**" e questo comporterà la coverage della classe PBT vicina al solo 90%, poiché non tutti i casi saranno coperti (vedasi quanto detto sopra).

Una volta fattasi l'idea che i numeri si siano correttamente generati e mai null, possiamo andare a modificare il codice, eliminando il **r2,** che indicava quanti numeri fossero **null o not null**, ottenendo come si può vedere dalle immagini qui sotto, una code coverage del 100% con questi risultati.

```java
package es1_Anagramma_NumeroArmstrong;

import net.jqwik.api.*;

class PBT {

        Anagramma_NArmstrong nA = new Anagramma_NArmstrong();

        @Property
        @StatisticsReport(format = Histogram.class)
        void shouldReturnCorrectResultPropertyBased(@ForAll @IntRange(min = Integer.MIN_VALUE, max = Integer.MAX_VALUE)Integer numero) {
            String range = nA.isNumeroArmstrong(numero) == true ? "Numeri Di Armstrong" : "Numeri Non di Armstrong";
            String range2 = numero < 0 ? "Numeri Negativi Generati (Non Armstrong)" : "Numeri Positivi Generati";
            Statistics.label("Statistiche").collect(range).collect(range2);
            Statistics.label("value").collect(numero);
            }

}
```

Figura 21: codice modificato della funzione shouldReturnCorrectResultPropertyBased();

```
timestamp = 2023-12-07T12:19:34.473662100, [PBT:shouldReturnCorrectResultPropertyBased] (2000) Statistiche =
      # |                                    label | count |
    ----|------------------------------------------|-------|-----------------------------------------------------------------------------------------
      0 |                     Numeri Di Armstrong |    53 | ####
      1 | Numeri Negativi Generati (Non Armstrong) |   495 | ########################################
      2 |                 Numeri Non di Armstrong |   947 | ###############################################################################################
      3 |               Numeri Positivi Generati |   505 | ########################################

timestamp = 2023-12-07T12:19:34.545049300, [PBT:shouldReturnCorrectResultPropertyBased] (1000) value =
      # |       label | count |
    ----|-------------|-------|-----------------------------------------------------------------------------------
      0 | -2147483648 |     7 | #######
      1 | -2147483647 |    10 | ##########
      2 | -2039277966 |     1 | #
      3 | -1997486434 |     1 | #
      4 | -1935614164 |     1 | #
      5 | -1906261867 |     1 | #
      6 | -1890372710 |     1 | #
      7 | -1787990443 |     1 | #
      8 | -1444131575 |     1 | #
      9 | -1424541787 |     1 | #
     10 | -1401143773 |     1 | #
     11 | -1392606691 |     1 | #
     12 | -1378853919 |     1 | #
     13 | -1328490100 |     1 | #
     14 | -1269321532 |     1 | #
     15 | -1259910000 |     1 | #
     16 | -1217755300 |     1 | #
     17 | -1147544889 |     1 | #
     18 | -1093721527 |     1 | #

Runs: 1/1          Errors: 0          Failures: 0

PBT [Runner: JUnit 5] (0,639 s)
  shouldReturnCorrectResultPropertyBased (0,639 s)
```

Figura 22: risultato dell'esecuzione dei test tramite la funzione shouldReturnCorrectResultPropertyBased() modificata;

Element	Coverage	Covered Instructions	Missed Instructions	Total Instructions
es1_Anagramma_NumeroArmstrong	100,0 %	733	0	733
es1_Anagramma_NumeroArmstrong	100,0 %	733	0	733
src	100,0 %	87	0	87
es1_Anagramma_NumeroArmstrong	100,0 %	87	0	87
Test	100,0 %	646	0	646
es1_Anagramma_NumeroArmstrong	100,0 %	646	0	646
es1_Anagramma_NumeroArmstrong	100,0 %	733	0	733
src	100,0 %	87	0	87
es1_Anagramma_NumeroArmstrong	100,0 %	87	0	87
Anagramma_NArmstrong.java	100,0 %	87	0	87
Anagramma_NArmstrong	100,0 %	87	0	87
Test	100,0 %	646	0	646
es1_Anagramma_NumeroArmstrong	100,0 %	646	0	646
PBT.java	100,0 %	52	0	52
TestAnagramma_NArmstrong.java	100,0 %	594	0	594

Figura 23: report del code coverage dell'intero progetto;