UNIVERSITÀ
DEGLI STUDI DI BARI
ALDO MORO

Documentazione di Progetto ITSS

Dipartimento di Informatica

Corso di laurea “Informatica e Tecnologie per la Produzione del software”

AsseMA
Filippo Leone – f.leone44@studenti.uniba.it – 717824
Samuele Giovanni Fiorino – s.fiorino3@studenti.uniba.it – 725038

Sommario

HOMEWORK 1 ..... *3*

INTRODUZIONE **3**

SPECIFICATION-BASED TESTING **4**

1) COMPRENDERE I REQUISITI 4
2) ESPLORARE COSA FA IL METODO 4
3) ESPLORARE INPUT, OUTPUT E PARTIZIONI 6
4) BOUNDARY CASES 6
5) DEFINIRE I CASI DI TEST 6
6) AUTOMATIZZARE I CASI DI TEST 7
7) AUMENTARE LA SUITE DI TEST CON LA CREATIVITÀ E L'ESPERIENZA 8

HOMEWORK 2 (task 1) *9*

STRUCTURAL TESTING **9**

1) ESAMINARE CODE COVERAGE DOPO BLACK BOX TESTING 9
2) ANALISI DELLE LINEE DI CODICE NON COPERTE 10
3) DEFINIRE I TEST CASES 12

HOMEWORK 2 (task 2) *14*

INTRODUZIONE **14**

STRUCTURAL TESTING **15**

1) ANALISI LINEE DI CODICE 15
2) DEFINIRE I TEST CASES 15
3) OTTIMIZZARE LA TEST SUITE 16

SPECIFICATION-BASED TESTING **17**

INTRODUZIONE 17
1) ESPLORARE INPUT, OUTPUT E PARTIZIONI 17
3) DEFINIRE I CASI DI TEST 18
4) AUTOMATIZZARE I CASI DI TEST 18
5) AUMENTARE LA SUITE DI TEST CON LA CREATIVITÀ E L'ESPERIENZA 20

HOMEWORK 1

INTRODUZIONE

Abbiamo deciso di testare il metodo *public static int solution(String s)* recuperato da un colloquio con un azienda tenuto da parte di uno dei componenti del gruppo. Di seguito viene riportato il codice.

```java
public static int solution (String s) {

        int WordMaxLenght = -1;

        if (s != null && !s.isEmpty() ) {

            // divido la stringa in un array di stringhe;
            String [] splitted = s.split(" ");

            int NWords = splitted.length;

            for ( int i = 0; i < NWords; i++) {

                if (splitted[i].matches("^[a-zA-Z0-9]*$") == true) {
                    int countDigit = 0;

                    for (int j = 0; j < splitted[i].length(); j++) {
                        if (Character.isDigit(splitted[i].charAt(j)) == true) {
                            countDigit++;
                        }
                    }

                    if ( (countDigit % 2) == 0 ) {
                        continue;
                    }

                    int countChar = 0;

                    for (int j = 0; j < splitted[i].length(); j++) {
                        if (Character.isLetter(splitted[i].charAt(j)) == true) {
                            countChar++;
                        }
                    }

                    if ( (countChar % 2) != 0) {
                        continue;
                    }

                    if ( WordMaxLenght < splitted[i].length()) {
                        WordMaxLenght = splitted[i].length();
                    }
                }
            }
        }
        return WordMaxLenght;
```

Il metodo prende in input una stringa (*String s*) in cui ci sono diverse password e conta il numero di caratteri della password più lunga che contiene solo caratteri alfanumerici, un numero pari di caratteri e un numero dispari di numeri.

- *s* – La stringa che contiene le password da validare. Può essere *null*.

Il metodo restituisce il numero di caratteri della password più lunga trovata, -1 altrimenti.

SPECIFICATION-BASED TESTING

1) COMPRENDERE I REQUISITI

L'obiettivo del metodo è di calcolare il numero di caratteri di una parola presente all'interno di una stringa (fornita dall'utente) che soddisfa determinati criteri.

Il metodo riceve un parametro:

I) *s* - la stringa presa in input dall'utente.

Il metodo restituisce il numero (*int*) di caratteri della password accettata più lunga.

2) ESPLORARE COSA FA IL METODO

Abbiamo effettuato dei test per verificare che il metodo rispettasse le nostre aspettative in base ai vari input forniti.

```java
@Test
public void testEmptyString() {
    String s = "";
    int result = Solution.solution(s);
    Assert.assertEquals(-1, result);
}

@Test
public void testNullString() {
    String s = null;
    int result = Solution.solution(s);
    Assert.assertEquals(-1, result);
}

@Test
public void testValidWord() {
    String s = "abc1234";
    int result = Solution.solution(s);
    Assert.assertEquals(7, result);
}

@Test
public void testInvalidDigitCount() {
    String s = "a1b2c3d4e";
    int result = Solution.solution(s);
    Assert.assertEquals(-1, result);
}

@Test
public void testInvalidAlphabeticCount() {
    String s = "a1234";
    int result = Solution.solution(s);
    Assert.assertEquals(-1, result);
}

@Test
public void testMinValidWord() {
    String s = "1";
    int result = Solution.solution(s);
    Assert.assertEquals(1, result);
}
```

3) ESPLORARE INPUT, OUTPUT E PARTIZIONI

Analizziamo i vari input che possono essere forniti al metodo, le varie combinazioni che essi possono avere e i valori di ritorno corrispondenti.

INPUTS

String s parametri:

1 – *Null String*

2 – Empty *String*

3 – *String* of lenght 1

4 – *String* of lenght > 1

COMBINAZIONI DI INPUTS

Non presenti in quanto il metodo prevede un unico parametro.

OUTPUTS

I) *WordMaxLenght*

II) -1

4) BOUNDARY CASES

I *Boundary Case* identificano situazioni estreme o limiti degli input che possono influenzare il comportamento del metodo. Testando questi casi, è possibile garantire una copertura completa delle condizioni e degli scenari previsti nel codice. Analizzando il metodo, abbiamo constatato che l'unico caso è:

- La stringa ***s*** contiene un solo carattere numerico.

5) DEFINIRE I CASI DI TEST

La combinazione di test con tutti i possibili input sono: 4X1=4 test.

Casi eccezionali
T1) *s* è *null*
T2) *s* è *empty*.

***s* lenght == 1**
T3) *s* contiene un solo carattere

***s* lenght > 1**
T4) *s* contiene più di un carattere

6) AUTOMATIZZARE I CASI DI TEST

Casi eccezionali

```java
@Nested
@DisplayName("Test casi eccezionali")
class TestCasiEccezionali{

    @Test
    @DisplayName("Test stringa vuota")
    void emptyString() {
        String s = "";
        int result = App.solution(s);
        assertEquals(-1, result);
    }

    @Test
    @DisplayName("Test stringa nulla")
    public void nullString() {
        String s = null;
        int result = App.solution(s);
        assertEquals(-1, result);
    }

}
```

s lenght == 1

```java
@DisplayName("Test s lenght == 1")
@Test
public void testMinValidWord() {
    String s = "1";
    int result = Solution.solution(s);
    assertEquals(1, result);
}
```

s lenght > 1

```java
@DisplayName("Test s lenght > 1")
@Test
public void validWord() {
    String s = "abcd123";
    int result = App.solution(s);
    assertEquals(7, result);
}
```

7) AUMENTARE LA SUITE DI TEST CON LA CREATIVITÀ E L'ESPERIENZA

T5) *s* contiene degli spazi all'interno della stringa

```java
@Test
public void WordOddCharacter() {
    String input = "Testing1 is fun";
    int result = App.solution(input);
    assertEquals(-1, result);
}
```

T6) *s* contiene il carattere “ “

```java
@Test
public void stringWithOnlySpace() {
    String s = "";
    int result = Solution.solution(s);
    assertEquals(-1, result);
}
```

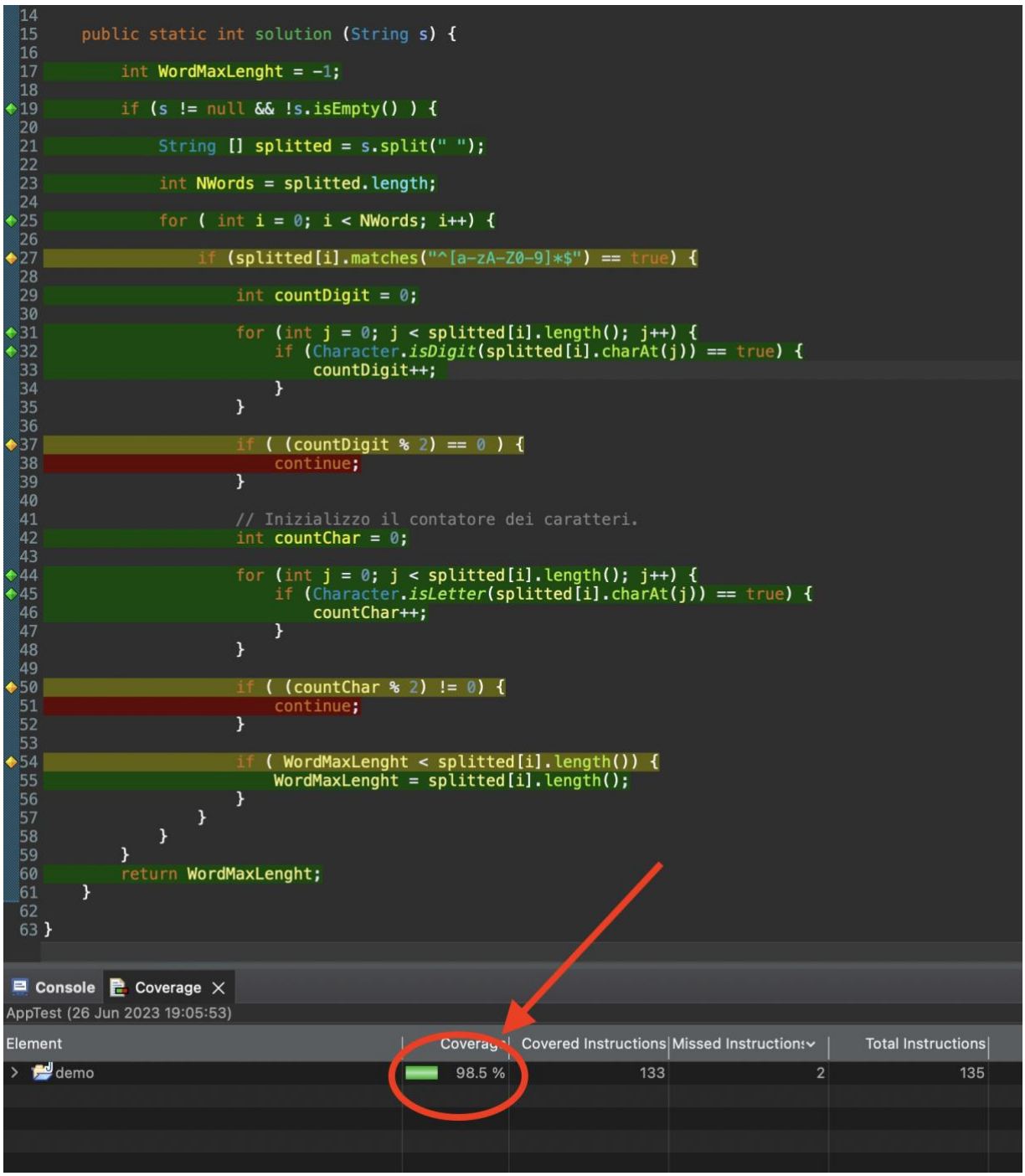
HOMEWORK 2 (task 1)

STRUCTURAL TESTING

1) ESAMINARE CODE COVERAGE DOPO BLACK BOX TESTING

Dopo aver implementato i Specification-Based Test e aver eseguito la suite di test con un tool di code coverage (in questo caso abbiamo utilizzato Jacoco) il metodo testato risulta essere coperto al 98.5%.

Procediamo dunque ad implementare gli Structural-Test in modo da ottenere la percentuale di Code Coverage più alta possibile.



2) ANALISI DELLE LINEE DI CODICE NON COPERTE

Dopo aver esaminato il Code Coverage report risultano esserci le seguenti linee di codice da sottoporre a test:

LINEA 27

L'istruzione *if* controlla se la password in esame contiene solo caratteri alfanumerici. Nei test effettuati vengono testate solo password alfanumeriche.

Effettuiamo un Branch Coverage test in cui esaminiamo solamente il Branch in cui nella password sono presenti caratteri speciali (non ammessi).

Branches covered=1 *total number of branches=2*

$$Branch\ coverage = \frac{branches\ covered}{total\ number\ of\ branches} \times 100\% = \frac{1}{2} \times 100\% = 50\%$$

LINEA 37

L'istruzione *if* controlla se la password in esame contiene una quantità di numeri dispari. Nei test effettuati vengono testate solo password con un numero pari di caratteri numerici.

Effettuiamo un Branch Coverage test in cui esaminiamo solamente il Branch in cui nella password è presente un numero dispari di caratteri numerici.

Branches covered=1 *total number of branches=2*

$$Branch\ coverage = \frac{branches\ covered}{total\ number\ of\ branches} \times 100\% = \frac{1}{2} \times 100\% = 50\%$$

LINEA 50

L'istruzione *if* controlla se la password in esame contiene una quantità dispari di caratteri alfabetici. Nei test effettuati vengono testate solo password con un numero dispari di caratteri alfabetici.

Effettuiamo un Branch Coverage test in cui esaminiamo solamente il Branch in cui nella password è presente un numero pari di caratteri alfabetici.

Branches covered=1 *total number of branches=2*

$$Branch\ coverage = \frac{branches\ covered}{total\ number\ of\ branches} \times 100\% = \frac{1}{2} \times 100\% = 50\%$$

LINEA 54

L'istruzione *if* controlla se il numero di caratteri della password più lunga trovata tra tutte le password finora esaminate, è minore della password attualmente esaminata. Nei test effettuati non viene mai inserita una serie di password in cui la password iesima è più grande delle successive (esempio di test per coprire il Branch covered *String s*= "ciao1 casa321 ciaociao12345 ciao123 ciaociao1").

Effettuiamo un Branch Coverage test in cui esaminiamo solamente il Branch in cui nella password è presente un numero pari di caratteri alfabetici.

$$Branches\ covered=1 \qquad total\ number\ of\ branches=2$$

$$Branch\ coverage = \frac{branches\ covered}{total\ number\ of\ branches} \times 100\% = \frac{1}{2} \times 100\% = 50\%$$

3) DEFINIRE I TEST CASES

Coprendo la linea 37 o la linea 50 o la linea 54 viene automaticamente coperto anche il branch della linea 27.

T1: s contiene una password con un numero di caratteri numerici pari.

```
1  @Test
2  public void wordEvenNumbers() {
3      String s = "abcd1234";
4      int result = App.solution(s);
5      assertEquals(-1, result);
6  }
```

T2: s contiene una password con un numero di caratteri alfabetici dispari.

```
1  public void WordOddCharacter() {
2      String input = "Testing1 is fun";
3      int result = App.solution(input);
4      assertEquals(-1, result);
5  }
```

T3: s contiene una password che è più lunga della precedente.

```java
@Test
public void multiWord() {
    String s = "ciao1 casa321 ciaociao12345 ciao123 ciaociao1";
    int result = App.solution(s);
    assertEquals(13, result);
}
```

Eseguendo i test sopra indicati è possibile coprire il 100% del Code Coverage.

HOMEWORK 2 (task 2)

INTRODUZIONE

Abbiamo deciso di testare il metodo *public static boolean isPalindromePermutation(String str)*. Di seguito viene riportato il codice.

```java
public class PalindromePermutation {

        public static boolean isPalindromePermutation(String str) {
        if (str == null) {
            return false;
        }

        String sanitizedStr = str.toLowerCase().replaceAll("\\s", "");
        int[] charCount = new int[26];

        for (char c : sanitizedStr.toCharArray()) {
            if (Character.isLetter(c)) {
                int index = c - 'a';
                charCount[index]++;
            }
        }

        int oddCount = 0;

        for (int count : charCount) {
            if (count % 2 != 0) {
                oddCount++;
            }
        }

        if(oddCount <= 1) {
            return true;
        } else {
            return false;
        }
    }
}
```

Il metodo verifica se una stringa può essere riarrangiata in modo da formare un palindromo.

Ad esempio, se consideriamo la stringa "Tact Coa", possiamo riarrangiare le lettere per formare la frase "taco cat", che è un palindromo. Pertanto, nel contesto di isPalindromePermutation, la stringa "Tact Coa" è considerata valida perché può essere riarrangiata in un palindromo.

- *str* – Stringa che contiene la frase da riarrangiare.

Restituisce *true* se la stringa soddisfa questa condizione, altrimenti restituisce *false*.

STRUCTURAL TESTING

1) ANALISI LINEE DI CODICE

Dopo aver esaminato il codice, risultano le seguenti linee di codice da sottoporre a test:

- Linea 7
- Linea 29

Linea 7

L'istruzione *if* controlla se la stringa in esame è nulla.

Effettuiamo un Branch Coverage test in cui esaminiamo solamente il Branch in cui la stringa non è nulla

Branches covered=1 *total number of branches=2*

$$Branch\ coverage = \frac{branches\ covered}{total\ number\ of\ branches} \times 100\% = \frac{1}{2} \times 100\% = 50\%$$

Linea 29

L'istruzione *if* controlla se è stata inserita una stringa palindroma restituendo true.

Effettuiamo un Branch Coverage test in cui esaminiamo solamente il Branch in cui la stringa non è palindroma

Branches covered=1 *total number of branches=2*

$$Branch\ coverage = \frac{branches\ covered}{total\ number\ of\ branches} \times 100\% = \frac{1}{2} \times 100\% = 50\%$$

2) DEFINIRE I TEST CASES

Linea 7

T1: La stringa *str* è nulla.

Linea 29

T2: La stringa *str* non è palindroma.

3) OTTIMIZZARE LA TEST SUITE

Per garantire il 100% del code coverage col minor numero di test, abbiamo bisogno di 2 test.

Di seguito vengono riportati i test da eseguire:

```java
@ParameterizedTest
@MethodSource("generateArguments")
void test(String str, boolean expectedResult) {
    Assert.assertEquals(Solution.isPalindromePermutation(str), expectedResult);
}

static Stream<Arguments> generateArguments() {
    return Stream.of(
            Arguments.of("hello", false),
            Arguments.of(null, true)
    );
}
```

Abbiamo sfruttato la possibilità di utilizzare dei *ParameterizedTest* che sfruttano un metodo generatore per eseguire i test sul metodo.

In questo caso il metodo *generateArguments()* crea uno stream di Arguments che viene poi passato al metodo testante *test()*.

Di seguito viene riportato anche il Code Coverage del metodo in analisi dopo aver aggiunto i nuovi:

```java
package org.example;

public class Solution {
    private Solution(){}

    public static boolean isPalindromePermutation(String str) {
        if (str == null) {
            return false;
        }

        String sanitizedStr = str.toLowerCase().replaceAll("\\s", "");
        int[] charCount = new int[26];

        for (char c : sanitizedStr.toCharArray()) {
            if (Character.isLetter(c)) {
                int index = c - 'a';
                charCount[index]++;
            }
        }

        int oddCount = 0;

        for (int count : charCount) {
            if (count % 2 != 0) {
                oddCount++;
            }
        }

        if(oddCount <= 1) {
            return true;
        } else {
            return false;
        }
    }
}
```

SPECIFICATION-BASED TESTING

INTRODUZIONE

Tramite il Code Coverage abbiamo constatato che i test coprono il 100% ma non abbiamo garantito che la qualità della suite di test sia esaustiva. Di seguito effettuiamo lo Specification Base Testing.

1) ESPLORARE INPUT, OUTPUT E PARTIZIONI

Analizziamo i vari input che possono essere forniti al metodo, le varie combinazioni che essi possono avere e i valori di ritorno corrispondenti.

INPUTS

String *str* parametri:

1 – *Null String*

2 – Empty *String*

3 – *String* of lenght 1

4 – *String* of lenght > 1

COMBINAZIONI DI INPUTS

Non presenti in quanto il metodo prevede un unico parametro.

OUTPUTS

I) *True se la stringa è palindroma*

II) False altrimenti

2) BOUNDARY CASES

I *Boundary Case* identificano situazioni estreme o limiti degli input che possono influenzare il comportamento del metodo. Testando questi casi, è possibile garantire una copertura completa delle condizioni e degli scenari previsti nel codice. Analizzando il metodo, abbiamo constatato che l'unico caso è:

- La stringa *str* è contiene un solo carattere ed ovviamente è possibile leggerla sia da destra che da sinistra

3) DEFINIRE I CASI DI TEST

La combinazione di test con tutti i possibili input sono: 4X1=4 test.

Casi eccezionali
T1) *str* è *null*
T2) *str* è *empty*.

***str* lenght == 1**
T3) *s* contiene un solo carattere

***str* lenght > 1**
T4) *s* contiene più di un carattere

4) AUTOMATIZZARE I CASI DI TEST

Casi eccezionali

```java
@Nested
    @DisplayName("Test casi eccezionali")
    class TestCasiEccezionali{

        @Test
        @DisplayName("Test caso stringa vuota")
        void emptyString() {
            String str = "";
            boolean result = Solution.isPalindromePermutation(str);
            assertTrue(result);
        }

        @Test
        @DisplayName("Test caso stringa nulla")
        public void nullString() {
            String str = null;
            boolean result = Solution.isPalindromePermutation(str);
            assertFalse(result);
        }

    }
```

Str lenght == 1

```java
@Test
@DisplayName("Test lunghezza str == 1")
public void testStringOneChar() {
    String str = "r";
    boolean result = Solution.isPalindromePermutation(str);
    assertTrue(result);
}
```

Str lenght > 1

```java
@Test
@DisplayName("Test caso stringa palindroma")
public void testPalindromeString() {
    String str = "radar";
    boolean result = Solution.isPalindromePermutation(str);
    assertTrue(result);
}
```

5) AUMENTARE LA SUITE DI TEST CON LA CREATIVITÀ E L'ESPERIENZA

T5) *s* contiene degli spazi all'interno della stringa

```java
@Test
@DisplayName("Test stringa con spazio")
public void testPalindromePermutationWithSpace() {
    String str = "taco cat";
    boolean result = Solution.isPalindromePermutation(str);
    Assert.assertTrue(result);
}
```

T6) *s* contiene il carattere " "

```java
@Test
@DisplayName("Test stringa contenente solo uno spazio")
public void testStringWithOnlySpace() {
    String str = " ";
    boolean result = Solution.isPalindromePermutation(str);
    Assert.assertTrue(result);
}
```