UNIVERSITÀ DEGLI STUDI DI BARI ALDO MORO

Esame di Integrazione e Test di Sistemi Software

a.a. 2022/23

Gruppo MGA

Mariniuc George Alexandru

Matricola: 735396

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

Data: 15/06/2023

UNIVERSITÀ DEGLI STUDI DI BARI ALDO MORO

Sommario

Introduzione

La presente relazione comprende le principali metodologie e tecniche usate nel processo di valutazione di un software o di un sistema per verificare se funziona correttamente e soddisfa i requisiti e le specifiche prestabilite. In altre parole, il testing del software.

Il testing del software può includere diverse attività, come l'analisi dei requisiti, la progettazione dei casi di test e l'esecuzione dei test.

Tutto avviene in una serie di fasi o cicli, in cui il software viene testato in modo sempre più dettagliato e approfondito fino a quando non viene soddisfatto un certo livello di qualità.

Specification-based testing

Lo specification-based testing (o test basato sulla specifica) è una tecnica di testing del software che si basa sulla verifica delle specifiche del software per identificare i casi di test da eseguire. Questa tecnica di testing è anche nota come test basato sui requisiti o test funzionale.

Il test basato sulla specifica parte dall'analisi dei requisiti del software per comprendere cosa il software deve fare e quali sono i suoi comportamenti attesi. Successivamente, vengono progettati i casi di test per verificare che il software si comporti come previsto in ogni situazione possibile.

Ciò implica la creazione di scenari di test che rappresentano tutti i possibili casi d'uso del software, anche quelli non previsti o non desiderati, al fine di individuare eventuali errori o malfunzionamenti. Inoltre, questa tecnica di testing si concentra sulla copertura dei requisiti funzionali del software, cioè verifica che tutte le funzionalità richieste siano implementate correttamente.

Il vantaggio del test basato sulla specifica è che garantisce un alto livello di copertura dei requisiti, poiché il processo di progettazione dei test si basa direttamente sulle specifiche del software. Tuttavia, questa tecnica non garantisce la copertura dei requisiti.

Specification-based testing workflow

Un approccio molto efficace ed efficiente per lo specification-based testing è quello a sette step che consiste nel:

1. Capire i requisiti (che cosa deve fare il programma, i suoi input e output)
2. Esplorare il programma per capire come reagirà in base ai vari input.
3. "Esplorare" gli input, gli output e identificare le partizioni.
4. Identificare i "casi limite" (boundary cases aka corner cases).
5. Ideare i casi di test.
6. Automatizzare i casi di test.
7. Migliorare la suite di test con l'esperienza acquisita e creatività.

I sette step possono essere seguiti per effettuare il testing di una unit, che può essere un metodo, una classe, ecc.

L'approccio è utilizzato di seguito per effettuare il testing di un metodo.

Capire i requisiti

Il metodo ha la funzione di generare tutte le possibili permutazioni, di una lunghezza specificata dall'utente, dei caratteri presenti in una stringa in input.

Il programma, quindi, ha bisogno di due input:

- str, la stringa contenente i caratteri necessari per effettuare le permutazioni;
- length, la dimensione delle permutazioni;

Infine, come output, il metodo dovrà restituire una lista di stringhe contenenti le permutazioni.

Esplorare il programma per capire come reagirà in base ai vari input

Questo passaggio è particolarmente importante se il codice non è stato realizzato personalmente e si vuole avere un chiaro modello mentale di come dovrebbe funzionare il programma.

```java
public static List<String> generatePermutations(String str, int length) {
    List<String> result = new ArrayList<>();
    generatePermutationsHelper(str, length, "", result);
    return result;
}
private static void generatePermutationsHelper(String str, int length, String current, List<String> result) {
    if (current.length() == length) {
        result.add(current);
    } else {
        for (int i = 0; i < str.length(); i++) {
            char c = str.charAt(i);
            generatePermutationsHelper(str, length, current + c, result);
        }
    }
}
```

"Esplorare" gli input, gli output e identificare le partizioni

Input

- str:
    1) Null.
    2) Empty.
    3) String of length 1.
    4) String of length >1.

- length:
    1. <0
    2. 0
    3. >0

Classi di output attesi

Array di stringhe (permutazioni):

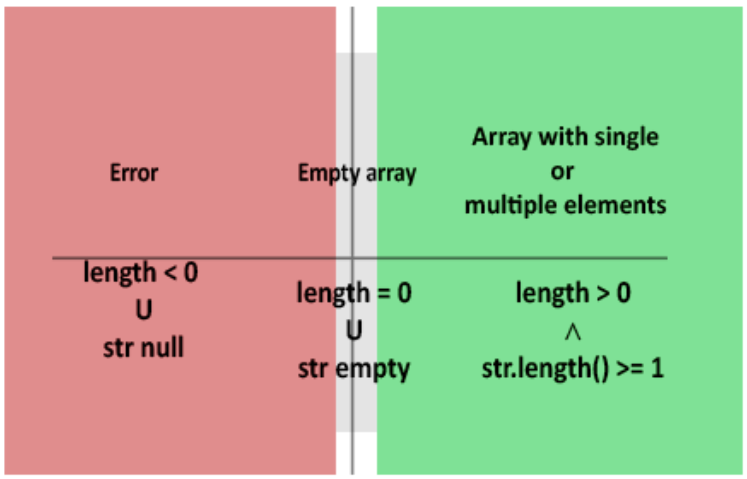
1. Empty array
2. Single item
3. Multiple items

Dove ciascuna permutazione può essere:

1. Empty
2. Single character
3. Multiple character

Identificare i "casi limite" (boundary cases aka corner cases).

1. length < 0 and/or str null.
2. str empty and/or length = 0.
3. str.length() >= 1 and length > 0.



Ideare i casi di test

TB1: str is null.

TB2: length < 0.

TB3: str is empty.

TB4: length = 0.

TB5: str = 1 and length = 1.

TB6: str > 1 and length = 1.

TB7: str > 1 and length > 1.

(Per ciascun test dove la stringa non è vuota/nulla bisogna controllare se, dati in input stringhe case-sensitive e/o contenenti caratteri speciali, l'output risultante coincide con quello atteso)

Automatizzare i casi di test

TB1

```
@ParameterizedTest
@DisplayName("TB1")
@NullSource
void t1StrIsNull(String value) {
    assertThrows(IllegalArgumentException.class, ()-> { SpecificationBasedTesting.generatePermutations(value, length: 1); });
}
```

TB2

```
@ParameterizedTest
@ValueSource(strings = {"abC","dEf", "$€ "})
@DisplayName("TB2")
void t2LengthLowerThan0(String str) {
    assertThrows(IllegalArgumentException.class, ()-> { SpecificationBasedTesting.generatePermutations(str, length: -1); });
}
```

TB3

```
@ParameterizedTest
@EmptySource
@DisplayName("TB3")
void t3StrIsEmpty(String value) {
    assertEquals(new ArrayList<String>(),SpecificationBasedTesting.generatePermutations(value, length: 1));
}
```

TB4

```
@ParameterizedTest
@ValueSource(strings = {"abC","dEf", "$€ "})
@DisplayName("TB4")
void t4Length0(String str) {
    assertEquals(Arrays.asList(""),SpecificationBasedTesting.generatePermutations(str, length: 0));
}
```

TB5

```
@ParameterizedTest
@ValueSource(strings = {"a","E","$", " "})
@DisplayName("TB5")
void t5Str1AndLength1(String str) {
    assertEquals(Arrays.asList(str),SpecificationBasedTesting.generatePermutations(str, length: 1));
}
```

TB6

```
@ParameterizedTest
@ValueSource(strings = {"abC","dEf", "$e "})
@DisplayName("TB6")
void tóStrGreaterThan1AndLength1(String str) {
    assertEquals(Arrays.asList(Character.toString(str.charAt(0)),Character.toString(str.charAt(1)),Character.toString(str.charAt(2))),SpecificationBasedTesting.generatePermutations(str, length: 1));
}
```

TB7

```
@ParameterizedTest
@MethodSource("t7args")
@DisplayName("TB7")
void t7StrGreaterThan1AndLengthGreaterThan1(ArrayList<String> expected, String str, int length) {
    assertEquals(expected,SpecificationBasedTesting.generatePermutations(str,length));
}

1 usage
static Stream<Arguments> t7args(){
    return Stream.of(
            Arguments.of( ...arguments: new ArrayList<String>(Arrays.asList("aa", "ab", "ac", "ba", "bb", "bc", "ca", "cb", "cc")), "abc", 2),
            Arguments.of( ...arguments: new ArrayList<String>(Arrays.asList("aaa", "aab", "aac", "aad", "aba", "abb", "abc", "abd", "aca", "acb", "acc", "acd",
                    "ada", "adb", "adc", "add", "baa", "bab", "bac", "bad", "bba", "bbb", "bbc", "bbd",
                    "bca", "bcb", "bcc", "bcd", "bda", "bdb", "bdc", "bdd", "caa", "cab", "cac", "cad",
                    "cba", "cbb", "cbc", "cbd", "cca", "ccb", "ccc", "ccd", "cda", "cdb", "cdc", "cdd",
                    "daa", "dab", "dac", "dad", "dba", "dbb", "dbc", "dbd", "dca", "dcb", "dcc", "dcd",
                    "dda", "ddb", "ddc", "ddd")), "abcd", 3),
            Arguments.of( ...arguments: new ArrayList<String>(Arrays.asList("aaa", "aaD", "aa$", "aa ", "aDa", "aDD", "aD$", "aD ", "a$a", "a$D", "a$$", "a$ ",
                    "a a", "a D", "a $", "a  ", "Daa", "DaD", "Da$", "Da ", "DDa", "DDD", "DD$", "DD ",
                    "D$a", "D$D", "D$$", "D$ ", "D a", "D D", "D $", "D  ", "$aa", "$aD", "$a$", "$a ",
                    "$Da", "$DD", "$D$", "$D ", "$$a", "$$D", "$$$", "$$ ", "$ a", "$ D", "$ $", "$  ",
                    " aa", " aD", " a$", " a ", " Da", " DD", " D$", " D ", " $a", " $D", " $$", " $ ",
                    "  a", "  D", "  $", "   ")), "aD$ ",3)
    );
}
```

Per maggiori dettagli consultare il file [“GeneratePermutations\Test cases GeneratePermutations.xls ”](#).

Durante questa fase alcuni test sono falliti poiché il programma non presentava la robustezza necessaria; alcuni casi d’uso non erano previsti (T1, T2).

Prima dei test:

```java
public static List<String> generatePermutations(String str, int length) {
    List<String> result = new ArrayList<>();
    generatePermutationsHelper(str, length, "", result);
    return result;
}
private static void generatePermutationsHelper(String str, int length, String current, List<String> result) {
    if (current.length() == length) {
        result.add(current);
    } else {
        for (int i = 0; i < str.length(); i++) {
            char c = str.charAt(i);
            generatePermutationsHelper(str, length, current + c, result);
        }
    }
}
```

Dopo:

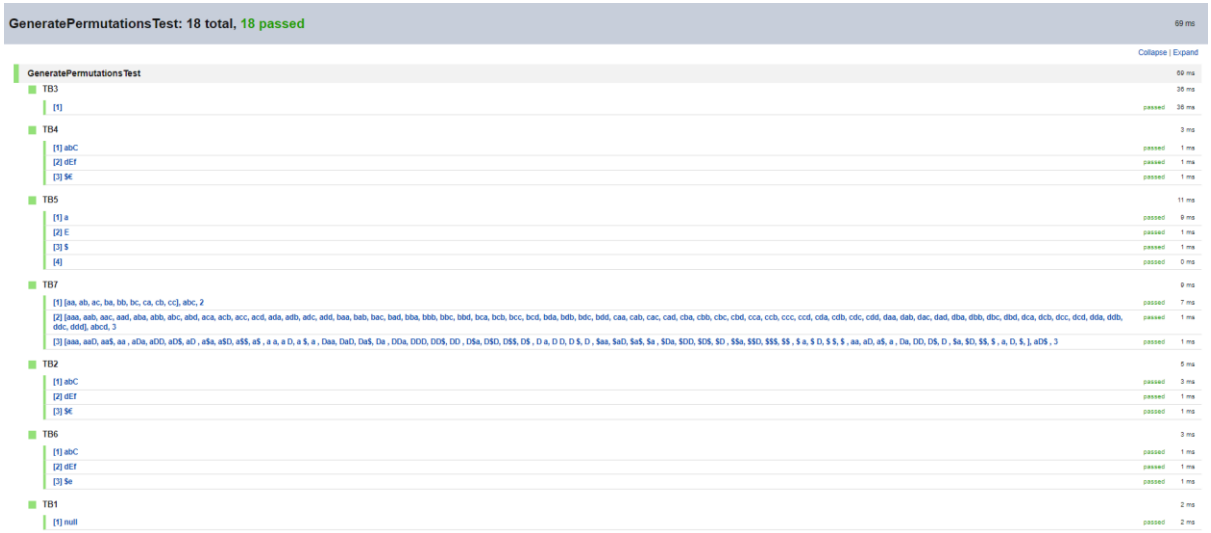
```java
public static List<String> generatePermutations(String str, int length) {
    if(str == null){
        throw new IllegalArgumentException("str can't be null!");
    }else if(length < 0){
        throw new IllegalArgumentException("length can't be negative!");
    }
    List<String> result = new ArrayList<>();
    generatePermutationsHelper(str, length, "", result);
    return result;
}
private static void generatePermutationsHelper(String str, int length, String current, List<String> result) {
    if (current.length() == length) {
        result.add(current);
    } else {
        for (int i = 0; i < str.length(); i++) {
            char c = str.charAt(i);
            generatePermutationsHelper(str, length, current + c, result);
        }
    }
}
```

Migliorare la suite di test con l'esperienza acquisita e creatività

È possibile arricchire la suite di test con strumenti di supporto che facilitano il lavoro del tester.

Ad esempio, durante la fase di automatizzazione, sono stati usati test parametrici, annotazioni che permettono di individuare i test in maniera efficiente, modelli di codice predefiniti (live templates) e sono stati generati rapporti dei risultati dei test.

Test results



Per maggiori dettagli consultare il file [“GeneratePermutations\Test Results – GeneratePermutationsTest.html”]

Structural testing

Un'altra tecnica di testing molto usata e che spesso è complementare a quella black-box è il white-box testing (aka structural testing).

Lo structural testing è una tecnica di testing del software che si concentra sulla valutazione degli aspetti interni dell'applicazione, come la struttura del codice sorgente. In altre parole, questo tipo di testing viene utilizzato per testare le funzionalità del software valutando direttamente il codice sorgente.

L'obiettivo principale dello structural testing è quello di identificare eventuali errori, bug o problemi di prestazioni nel codice sorgente dell'applicazione, in modo da poterli correggere prima che il software venga rilasciato.

Esistono diverse tecniche di structural testing, tra cui il code coverage testing, il line coverage testing, il branch coverage testing, il condition coverage testing, il path coverage testing e il Modified Condition/Decision Coverage (MC/DC) testing, ognuna delle quali si concentra su un aspetto diverso della struttura del codice sorgente. In generale, lo structural testing è una componente fondamentale di qualsiasi strategia di testing del software, poiché aiuta a garantire la qualità e l'affidabilità del software stesso.

Structural testing workflow

Il workflow del structural testing può essere definito come segue:

1. Eseguire il testing basato sulle specifiche (approccio a sette step).
2. Comprendere il codice se non è stato realizzato personalmente.
3. Verificare il code coverage.
4. Per ogni pezzo di codice "non coperto":
    a) Chiedersi perché non è stato coperto?
    b) Decidere se quel pezzo di codice ha bisogno di un test (se sì, passare al punto c.)
    c) Implementare il test.
5. Torna al punto 3.

In seguito a una rapida esecuzione dello strumento di code coverage presente in IntelliJ la test suite realizzata in precedenza si è rilevata essere efficace data la "copertura" totale del codice:

```
public static List<String> generatePermutations(String str, int length) {
    // Specification: Given a string str and an integer length,
    // generate all possible permutations of length characters from str
    // and return them as a list of strings.
    if(str == null){
        throw new IllegalArgumentException("str can't be null!");
    }else if(length < 0){
        throw new IllegalArgumentException("length can't be negative!");
    }
    List<String> result = new ArrayList<>();
    generatePermutationsHelper(str, length, "", result);
    return result;
}
private static void generatePermutationsHelper(String str, int length, String current, List<String> result) {
    if (current.length() == length) {
        result.add(current);
    } else {
        for (int i = 0; i < str.length(); i++) {
            char c = str.charAt(i);
            generatePermutationsHelper(str, length, current + c, result);
        }
    }
}
```

Per maggiori dettagli consultare il file "GeneratePermutations\htmlReports\index.html".

Ciò però non ci garantisce che il metodo sia bug-free, ci possono essere casi d'uso non previsti nonostante la suite di test e la copertura totale.

Possiamo rendere il structural testing più esaustivo/completo utilizzando un'altra delle tecniche elencate prima come ad esempio il branch + condition coverage.

Per eseguire il branch + condition testing basta analizzare in dettaglio il codice e calcolare la seguente formula:

$$\text{c+b coverage} = \frac{\text{branches covered} + \text{conditions covered}}{\text{number of branches} + \text{number of conditions}} \times 100\%$$

Nel nostro caso, per ottenere il 100% di branch + condition coverage dobbiamo assicurarci di aver implementato i seguenti test:

1. **(str == null)**

a) str è null.
b) str è non nulla

2. **(length < 0)**

a) length è uguale o maggiore a 0.
b) length è un valore negativo.

3. **(current.length() == length)**

a) current.length() = 0 e length = 0
b) current.length() > 0 e length = 0
c) current.length() = 0 e length > 0

4. **for (int i = 0; i < str.length(); i++)**

a) str.length() = 0.
b) str.length() > 0.

Guardando la suite realizzata in precedenza notiamo che i test risultanti dal branch + coverage testing sono già stati implementati.

Un altro strumento utilizzato per effettuare il white-box testing è il PIT Mutation Testing.

PIT Mutation Testing

Il PIT (short for "Mutation Testing with PiT") è un framework di testing automatico che consente di eseguire test di mutazione su codice sorgente Java. In pratica, il PIT introduce piccole modifiche (o mutazioni) nel codice sorgente originale, creando quindi una versione mutata del codice. Queste mutazioni possono includere, ad esempio, l'inversione di operatori, la rimozione di istruzioni, la sostituzione di valori costanti, e così via.

Una volta che sono state create le versioni mutate del codice, il framework PIT esegue i test su queste nuove versioni, cercando di rilevare se ci sono test che falliscono. Se un test fallisce quando viene eseguito su una versione mutata del codice, allora si può presumere che la mutazione abbia introdotto un errore nel codice. Questo significa che il test in questione è in grado di rilevare un errore se presente nel codice originale.

In sintesi, l'obiettivo del PIT Mutation Testing è di migliorare la qualità del codice sorgente, consentendo ai tester di individuare le parti del codice che sono mal testate o che presentano delle vulnerabilità.

Risultati PIT Mutation Testing

Pit Test Coverage Report

Package Summary

org.example

Number of Classes	Line Coverage		Mutation Coverage		Test Strength	
1	93%	13/14	100%	9/9	100%	9/9

Breakdown by Class

Name	Line Coverage		Mutation Coverage		Test Strength	
GeneratePermutations.java	93%	13/14	100%	9/9	100%	9/9

Report generated by PIT 1.11.6

GeneratePermutations.java

```
Mutations
10 1. negated conditional → KILLED
12 1. changed conditional boundary → KILLED
   2. negated conditional → KILLED
16 1. removed call to org/example/GeneratePermutations::generatePermutationsHelper → KILLED
17 1. replaced return value with Collections.emptyList for org/example/GeneratePermutations::generatePermutations → KILLED
20 1. negated conditional → KILLED
23 1. negated conditional → KILLED
   2. changed conditional boundary → KILLED
25 1. removed call to org/example/GeneratePermutations::generatePermutationsHelper → KILLED
```

Active mutators

- CONDITIONALS_BOUNDARY
- EMPTY_RETURNS
- FALSE_RETURNS
- INCREMENTS
- INVERT_NEGS
- MATH
- NEGATE_CONDITIONALS
- NULL_RETURNS
- PRIMITIVE_RETURNS
- TRUE_RETURNS
- VOID_METHOD_CALLS

Tests examined

- GeneratePermutationsTest.[engine:junit-jupiter]/[class:GeneratePermutationsTest]/[test-template:t6StrGreaterThan1AndLength1(java.lang.String)]/[test-template-invocation:#1] (0 ms)
- GeneratePermutationsTest.[engine:junit-jupiter]/[class:GeneratePermutationsTest]/[test-template:t6StrGreaterThan1AndLength1(java.lang.String)]/[test-template-invocation:#2] (0 ms)
- GeneratePermutationsTest.[engine:junit-jupiter]/[class:GeneratePermutationsTest]/[test-template:t2LengthLowerThan0(java.lang.String)]/[test-template-invocation:#2] (1 ms)
- GeneratePermutationsTest.[engine:junit-jupiter]/[class:GeneratePermutationsTest]/[test-template:t1StrIsNull(java.lang.String)]/[test-template-invocation:#1] (1 ms)
- GeneratePermutationsTest.[engine:junit-jupiter]/[class:GeneratePermutationsTest]/[test-template:t4Length0(java.lang.String)]/[test-template-invocation:#1] (0 ms)
- GeneratePermutationsTest.[engine:junit-jupiter]/[class:GeneratePermutationsTest]/[test-template:t2LengthLowerThan0(java.lang.String)]/[test-template-invocation:#3] (1 ms)
- GeneratePermutationsTest.[engine:junit-jupiter]/[class:GeneratePermutationsTest]/[test-template:t7StrGreaterThan1AndLengthGreaterThan1(java.util.ArrayList, java.lang.String, int)]/[test-template-invocation:#3] (0 ms)
- GeneratePermutationsTest.[engine:junit-jupiter]/[class:GeneratePermutationsTest]/[test-template:t7StrGreaterThan1AndLengthGreaterThan1(java.util.ArrayList, java.lang.String, int)]/[test-template-invocation:#2] (0 ms)
- GeneratePermutationsTest.[engine:junit-jupiter]/[class:GeneratePermutationsTest]/[test-template:t7StrGreaterThan1AndLengthGreaterThan1(java.util.ArrayList, java.lang.String, int)]/[test-template-invocation:#1] (4 ms)
- GeneratePermutationsTest.[engine:junit-jupiter]/[class:GeneratePermutationsTest]/[test-template:t2LengthLowerThan0(java.lang.String)]/[test-template-invocation:#1] (1 ms)
- GeneratePermutationsTest.[engine:junit-jupiter]/[class:GeneratePermutationsTest]/[test-template:t5Str1AndLength1(java.lang.String)]/[test-template-invocation:#1] (6 ms)
- GeneratePermutationsTest.[engine:junit-jupiter]/[class:GeneratePermutationsTest]/[test-template:t5Str1AndLength1(java.lang.String)]/[test-template-invocation:#2] (1 ms)
- GeneratePermutationsTest.[engine:junit-jupiter]/[class:GeneratePermutationsTest]/[test-template:t3StrIsEmpty(java.lang.String)]/[test-template-invocation:#1] (35 ms)
- GeneratePermutationsTest.[engine:junit-jupiter]/[class:GeneratePermutationsTest]/[test-template:t4Length0(java.lang.String)]/[test-template-invocation:#2] (1 ms)
- GeneratePermutationsTest.[engine:junit-jupiter]/[class:GeneratePermutationsTest]/[test-template:t4Length0(java.lang.String)]/[test-template-invocation:#3] (1 ms)
- GeneratePermutationsTest.[engine:junit-jupiter]/[class:GeneratePermutationsTest]/[test-template:t6StrGreaterThan1AndLength1(java.lang.String)]/[test-template-invocation:#3] (0 ms)
- GeneratePermutationsTest.[engine:junit-jupiter]/[class:GeneratePermutationsTest]/[test-template:t5Str1AndLength1(java.lang.String)]/[test-template-invocation:#3] (1 ms)
- GeneratePermutationsTest.[engine:junit-jupiter]/[class:GeneratePermutationsTest]/[test-template:t5Str1AndLength1(java.lang.String)]/[test-template-invocation:#4] (1 ms)

Per maggiori dettagli consultare il file “GeneratePermutations\pit\index.html”

Infine, possiamo constatare che se si realizza un buon specification-based testing, e quindi una buona progettazione ed esecuzione di test partendo dalle specifiche del programma, il lavoro necessario per implementare il white-box testing sarà decisamente inferiore.

Per dimostrare ciò e quanto detto in precedenza creeremo il minor numero di test necessari per avere il 100% di code coverage e analizzeremo se i test risultanti sono sufficienti per il black-box testing.

Line code coverage del metodo isPalindrome()

```
public static boolean isPalindrome(String str, int start, int end) {
    if (str == null || str.isEmpty() || start < 0 || end < 0 || end >= str.length() || end < start) {
        throw new IllegalArgumentException("Invalid input parameters");
    }
    while (start < end) {
        if (str.charAt(start) != str.charAt(end)) {
            return false;
        }
        start++;
        end--;
    }
    return true;
}
```

Per raggiungere il 100% di line code coverage ci basteranno 3 test:

1) T1W: str = null

Questo ci permetterà di rendere la prima condizione vera e quindi "sollevare" l'eccezione.

2) T2W: str non nulla ∧ non palindroma ∧ start < end

In questo caso eviteremo il "sollevare" dell'eccezione presente nel primo ramo, entreremo nel ciclo dove ciascun'istruzione presente verrà eseguita almeno una volta.

3) T3W: str palindroma ∧ start < end

L'unica riga di codice non coperta prima del T3 è quella che ritorna valore true. Per raggiungere tal punto del programma ci basta fornire argomenti che abbino le caratteristiche imposte nella prima condizione, che il valore di start sia minore di end per poter entrare nel ciclo e che la parola sia palindroma poiché è necessario uscire dal ciclo senza ritornare il valore false.

È possibile consultare il report di code coverage contenuto nel seguente file: "IsPalindrome\htmlReports\index.html".

Come vedremo più avanti tali test non basteranno con l'approccio dello functional-based testing.

Specification-based testing del metodo isPalindrome()

Capire i requisiti

Il metodo ha la funzione di controllare se data una stringa e due interi “start” e “end” in input, la stringa formata dai caratteri presenti nell'intervallo [start , end] è palindroma.

Il programma, quindi, ha bisogno di tre input:

- str, una stringa da cui estrarre i caratteri necessari per formare la sottostringa sulla quale verrà effettuato il controllo palindromo;
- start, la posizione del primo carattere della sottostringa da analizzare;
- end, la posizione dell’ultimo carattere della sottostringa da analizzare;

Infine, come output, il metodo dovrà ritornare valore true se la sottostringa è palindroma, false altrimenti.

Input, output e partizioni

Input

- str:
    1) Null.
    2) Empty.
    3) Str.length() > 0

- start:
    1) <0
    2) 0
    3) >0

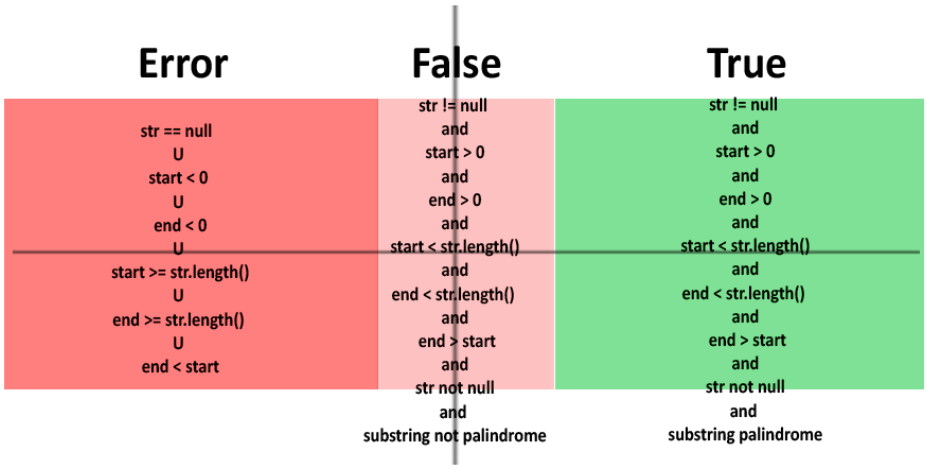
- end:
    1) <0
    2) 0
    3) >0

Classi di output attesi

Valore boolean:

1) True (sottostringa palindroma)
2) False (sottostringa non palindroma)

Boundary cases



Casi di test

T1B: str == null

T2B: str empty

T3B: start < 0

T4B: end < 0

T5B: start >= str.length()

T6B: end >= str.length()

T7B: end < start

T8B: str not null and end – start = 0

T9B: str not null and end – start > 1 and not palindrome

T10B: str not null and end – start > 1 and palindrome

(Per ciascun test dove la stringa non è vuota/nulla bisogna controllare se, dati in input stringhe case-sensitive e/o contenenti caratteri speciali, l'output risultante coincide con quello atteso)

È possibile osservare che i test risultanti dopo la fase di specification-based testing superano di gran lunga quelli ottenuti con il white-box testing; solo tre di quest'ultimi coincidono:

- T1 white-box ⇔ T1 black-box.
- T2 white-box ⇔ T9 black-box.
- T3 white-box ⇔ T10 black-box.

Automatizzare i casi di test

T1B

```
@ParameterizedTest
@NullSource
@DisplayName("T1W-T1B: strIsNull")
void t1strIsNull(String str) {
    assertThrows(IllegalArgumentException.class, () -> {
        IsPalindrome.isPalindrome(str, start: 0, end: 1);});
}
```

T2B

```
@ParameterizedTest
@EmptySource
@DisplayName("T2B: str empty")
void t2StrEmpty(String str) {
    assertThrows(IllegalArgumentException.class, () -> {
        IsPalindrome.isPalindrome(str, start: 0, end: 1);});
}
```

T3B

```
@Test
@DisplayName("T3B: start < 0")
void t3StartLowerThan0() {
    assertThrows(IllegalArgumentException.class, () -> {
        IsPalindrome.isPalindrome( str: "abcd", start: -1, end: 3);});
}
```

T4B

```
@Test
@DisplayName("T4B: end < 0")
void t4EndLowerThan0() {
    assertThrows(IllegalArgumentException.class, () -> {
        IsPalindrome.isPalindrome( str: "abcd", start: 0, end: -1);});
}
```

T5B

```
@Test
@DisplayName("T5B: start >= str.length()")
void t5StartGreaterOrEqualStrLength() {
    assertThrows(IllegalArgumentException.class, () -> {
        IsPalindrome.isPalindrome( str: "abcd", start: 4, end: 3);});
}
```

T6B

```
@Test
@DisplayName("T6B: end >= str.length()")
void t6EndGreaterOrEqualsStrLength() {
    assertThrows(IllegalArgumentException.class, () -> {
        IsPalindrome.isPalindrome( str: "abcd", start: 0, end: 4);});
}
```

T7B

```
@Test
@DisplayName("T7B: end < start")
void t7EndLowerThanStart() {
    assertThrows(IllegalArgumentException.class, () -> {
        IsPalindrome.isPalindrome( str: "abcd", start: 2, end: 1);});
}
```

T8B

```
@Test
@DisplayName("T8B: str not null and end - start = 0")
void t8StrNotNullAndEndEqualsStart() { assertEquals( expected: true, IsPalindrome.isPalindrome( str: "abcd", start: 0, end: 0)); }
```

T9B

```
@ParameterizedTest
@MethodSource("randNotPalArgs")
@DisplayName("T2W-T9B: str not null and end - start > 1 and not palindrome")
void t9invalidPalindrome(String str, int start, int end) {
    assertEquals( expected: false, IsPalindrome.isPalindrome(str, start, end));
}
1 usage
static Stream<Arguments> randNotPalArgs(){
    return Stream.of(
            Arguments.of( ...arguments: "abcde", 1, 2),
            Arguments.of( ...arguments: "qwerty", 2, 4),
            Arguments.of( ...arguments: "asdfg", 1, 3),
            Arguments.of( ...arguments: "zxc vb", 2, 3),
            Arguments.of( ...arguments: "$e €fl", 3, 4)
    );
}
```

T10B

```
@ParameterizedTest
@MethodSource("palArgs")
@DisplayName("T3W-T10B: palindrome")
void t10validPalindrome(String str, int start, int end) {
    assertEquals( expected: true, IsPalindrome.isPalindrome(str,start,end));
}

1 usage
static Stream<Arguments> palArgs(){
    return Stream.of(
            Arguments.of( ...arguments: "otto", 1, 2),
            Arguments.of( ...arguments: "aerea", 2, 2),
            Arguments.of( ...arguments: "aveva", 1, 3),
            Arguments.of( ...arguments: "aci d ica", 2, 6),
            Arguments.of( ...arguments: "ingegni", 0, 6)
    );
}
```

Per maggiori dettagli consultare il file [“IsPalindrome\Test cases IsPalindrome.xls”].

Test results

Per maggiori dettagli consultare il file "IsPalindrome\Test Results – IsPalindromeTest.html"

Property-Based Testing

Il Property-Based Testing (PBT) è una tecnica di testing software che si concentra sulla verifica delle proprietà generali del software piuttosto che sui singoli casi di test specifici. Invece di creare casi di test manualmente, come nel testing tradizionale basato su esempi, il PBT genera automaticamente una serie di input casuali o semi-casuali e verifica che determinate proprietà o invarianze del sistema siano sempre valide per quegli input.

Le proprietà generali possono essere definite come regole o assunzioni che il software dovrebbe soddisfare in ogni caso. Ad esempio, se si sta testando una funzione di ordinamento, una proprietà generale potrebbe essere che l'output della funzione deve essere sempre una lista ordinata. Durante l'esecuzione del PBT, il sistema genera automaticamente una serie di input casuali, applica la funzione di test a ciascun input e verifica se la proprietà generale viene mantenuta per ogni esecuzione.

La tecnica verrà applicata di seguito al metodo generatePermutations testato precedentemente utilizzando l'approccio basato sulle specifiche.

Si è pensato di suddividere il workflow come segue:

1. Ideare le proprietà necessarie per il corretto funzionamento della unit e quelle per le quali il programma potrebbe generare fault;
2. Verificare il funzionamento del sistema generando input casuali;
3. Raccogliere opportunamente le statistiche;
4. Ripetere il ciclo se non siete soddisfatti dei risultati ottenuti.

Proprietà del metodo generatePermutations

Input

Per il corretto funzionamento, il metodo dovrà ricevere in input, come visto in precedenza, una stringa non nulla (anche vuota) e un intero maggiore o uguale a zero.

Output

Come output, il programma dovrà ritornare:

- Un array vuoto di stringhe se viene fornita una stringa vuota e/o un intero uguale a 0;
- Un array di stringhe, di lunghezza pari all'intero fornito in input, contenente le permutazioni.

Generazione di input casuali

La generazione di 500 input casuali avviene tenendo conto delle seguenti proprietà:

- Le stringhe possono essere:
  - Nulle;
  - Di lunghezza compresa tra 0 e 3.
- L'intero riguardante la lunghezza delle permutazioni può variare tra 0 e 3.

Implementazione PBT

```
@Report(Reporting.GENERATED)
@Property(tries = 500)
@StatisticsReport(format = Histogram.class)
void generateValidPermutations(@ForAll("generateValidArgs") Tuple.Tuple3<String, Integer , List<String>> args){
    List<String> result = GeneratePermutations.generatePermutations(args.get1(), args.get2());
    String lunghezzaStringa = args.get1().length() <= 1 ? "Str.length () < 1" : "Str.length () > 1";
    String lunghezzaPerm = args.get2() <= 1 ? "Perm length <= 1 " : "Perm length > 1";
    Statistics.collect(lunghezzaStringa, lunghezzaPerm);
    assertTrue( condition: args.get3().size() == result.size() && result.containsAll(args.get3()) && args.get3().containsAll(result));
}
```

```
@Provide
Arbitrary<Tuple.Tuple3<String,Integer,List<String>>> generateValidArgs(){
    Arbitrary<Integer> length = Arbitraries.integers().between(0, 3);
    return length.flatMap(lunghezza -> Arbitraries.strings().ofMinLength(0).ofMaxLength(3).withCharRange('a','z').injectDuplicates( duplicateProbability: 0.0)
            .flatMap(stringa -> Arbitraries.strings().ofLength(lunghezza).withChars(stringa.toCharArray()).list().uniqueElements().filter(list-> list.stream().count() == Math.pow(stringa.length(),lunghezza))
                    .map(expected -> Tuple.of(stringa,lunghezza,expected)))).ignoreException(net.jqwik.api.TooManyFilterMissesException.class).ignoreException(net.jqwik.api.JqwikException.class);
}
```

```
@Report(Reporting.GENERATED)
@Property(tries = 500)
@StatisticsReport(format = Histogram.class)
void generateInvalidPermutations(@ForAll("generateInvalidArgs") Tuple.Tuple2<String, Integer> args){
    String strNull = args.get1() == null ? "str null" : "str not null";
    String invalid = args.get2() < 0 ? "length < 0" : "length >= 0";
    Statistics.collect( ...values: strNull," and ", invalid);
    assertThrows(IllegalArgumentException.class, () -> {GeneratePermutations.generatePermutations(args.get1(), args.get2());});
}
```

```
no usages
@Provide
Arbitrary<Tuple.Tuple2<String,Integer>> generateInvalidArgs(){
    Arbitrary<Integer> lengthInvalid = Arbitraries.integers().lessOrEqual( max: -1);
    Arbitrary<Integer> lengthValid = Arbitraries.integers().between(0,3);
    Arbitrary<String> strInvalid = Arbitraries.strings().injectNull( nullProbability: 1.0);
    Arbitrary<String> strValid = Arbitraries.strings().withCharRange('a','z').ofMinLength(0).ofMaxLength(3).injectDuplicates( duplicateProbability: 0.0);
    return Arbitraries.oneOf(Combinators.combine(strValid,lengthInvalid).as((s,l) -> Tuple.of(s,l)),
            Combinators.combine(strInvalid,lengthValid).as((s,l) -> Tuple.of(s,l)),
            Combinators.combine(strInvalid,lengthInvalid).as((s,l) -> Tuple.of(s,l))
    );
}
```

Statistiche

L'osservazione si concentrerà sui casi limite, più precisamente:

- Numero di stringhe nulle;
- Numero di stringhe di lunghezza inferiore o uguale a 1.
- Numero di stringhe di lunghezza superiore a 1.
- Lunghezza delle permutazioni inferiore o uguale a 1.
- Lunghezza delle permutazioni maggiore a 1.
- Lunghezza delle permutazioni negativa.

```
   # |                            label | count |
-----|----------------------------------|-------|-------------------------------------------------------------------------------
   0 | Str.length () < 1 Perm length <= 1  |   137 | ##############################################################################
   1 |   Str.length () < 1 Perm length > 1 |    99 | ###################################################
   2 |   Str.length () > 1 Perm length > 1 |   144 | ###################################################################################
   3 | Str.length () > 1 Perm length <= 1  |   120 | ###############################################################
```

```
timestamp = 2023-05-15T17:18:01.432395300, GeneratePermutationsPropertyTest:generateValidPermutations =
                          |----------------------jqwik----------------------
tries = 500               | # of calls to property
checks = 500              | # 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 = 12     | # of all combined edge cases
edge-cases#tried = 12     | # of edge cases tried in current run
seed = 6397295235249298144    | random seed to reproduce generated values
```

```
   # |                        label | count |
-----|------------------------------|-------|-----------------------------------------------------------------------------------
   0 | str not null  and  length < 0 |   171 | ##############################################################################
   1 |    str null  and  length >= 0 |   178 | ###################################################################################
   2 |     str null  and  length < 0 |   151 | ########################################################################
```

```
timestamp = 2023-05-15T17:16:42.599195, GeneratePermutationsPropertyTest:generateInvalidPermutations =
                          |----------------------jqwik----------------------
tries = 500               | # of calls to property
checks = 500              | # 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 = 20     | # of all combined edge cases
edge-cases#tried = 20     | # of edge cases tried in current run
seed = 3161514501036321199    | random seed to reproduce generated values
```

Conclusioni PBT

Effettuando un buon PBT, si può ottenere facilmente il 100% di code coverage e "coprire" tutti i casi limite ma questo non vuol dire che può sostituire le tecniche analizzate precedentemente.

Il PBT viene spesso utilizzato in combinazione con l'approccio di sviluppo guidato dai test (Test-Driven Development, TDD) o come complemento al testing tradizionale basato su esempi.