

**Gruppo “crushTheBug”**

Integrazione e Test di Sistemi Software

C.dL Informatica e T.P.S.

A.A. 2022/23

**Realizzato da:**

Antonio Pio Caputo – 736253 – a.caputo79@studenti.uniba.it

Salvatore Forte – 737321 – [s.forte16@studenti.uniba.it](mailto:s.forte16@studenti.uniba.it)



Sommario

[**Home work n°1 -> Black-box Testing** 4](#_Toc148087844)

[Metodo n°1 -> add(T data) 4](#_Toc148087845)

[TESTING WORKFLOW FOR SPECIFICATION-BASED TESTING 4](#_Toc148087846)

[TESTING WORKFLOW FOR STRUCTURAL TESTING 8](#_Toc148087847)

[Metodo n°2 -> remove(T data) 9](#_Toc148087848)

[TESTING WORKFLOW FOR SPECIFICATION-BASED TESTING 10](#_Toc148087849)

[TESTING WORKFLOW FOR STRUCTURAL TESTING 13](#_Toc148087850)

[Metodo n°3 -> get(int index) 17](#_Toc148087851)

[TESTING WORKFLOW FOR SPECIFICATION-BASED TESTING 17](#_Toc148087852)

[TESTING WORKFLOW FOR STRUCTURAL TESTING 20](#_Toc148087853)

[Metodo n°4 -> size() | isEmpty() 23](#_Toc148087854)

[TESTING WORKFLOW FOR SPECIFICATION-BASED TESTING 23](#_Toc148087855)

[TESTING WORKFLOW FOR STRUCTURAL TESTING 24](#_Toc148087856)

[Metodo n°5 -> node(T data) 26](#_Toc148087857)

[TESTING WORKFLOW FOR SPECIFICATION-BASED TESTING 26](#_Toc148087858)

[TESTING WORKFLOW FOR STRUCTURAL TESTING 28](#_Toc148087859)

[**Home work n°2 -> White-box Testing** 32](#_Toc148087860)

[**Classe:** StringManipulator 32](#_Toc148087861)

[**Specification-based testing Metodo1: concatenate(String str1, String str2)** 36](#_Toc148087862)

[**TESTING WORKFLOW STRUCTURAL TESTING** 37](#_Toc148087863)

[**1.** **Understanding the requirements** 37](#_Toc148087864)

[**2.** **Explore what the program does for various inputs** 37](#_Toc148087865)

[**3.** **Explore inputs, outputs and identify partitions** 38](#_Toc148087866)

[**4.** **Identify boundary cases** 39](#_Toc148087867)

[**5.** **Devise test cases** 39](#_Toc148087868)

[**6.** **Automate test cases** 39](#_Toc148087869)

[**7.** **Augment the test suite with creativity and experience** 39](#_Toc148087870)

[**LINK DI COLLEGAMENTO AI TEST CASES EFFETTUATI** 39](#_Toc148087871)

[**Specification-based testing Metodo2: reverse(String str)** 40](#_Toc148087872)

[**TESTING WORKFLOW STRUCTURAL TESTING** 40](#_Toc148087873)

[**1.** **Understanding the requirements** 40](#_Toc148087874)

[**2.** **Explore what the program does for various inputs** 40](#_Toc148087875)

[**3.** **Explore inputs, outputs and identify partitions** 41](#_Toc148087876)

[**4.** **Identify boundary cases** 42](#_Toc148087877)

[**5.** **Devise test cases** 42](#_Toc148087878)

[**6.** **Automate test cases** 43](#_Toc148087879)

[**7.** **Augment the test suite with creativity and experience** 43](#_Toc148087880)

[**LINK DI COLLEGAMENTO AI TEST CASES EFFETTUATI** 43](#_Toc148087881)

[**Specification-based testing Metodo3: isPalindrome(String str)** 43](#_Toc148087882)

[**TESTING WORKFLOW STRUCTURAL TESTING** 43](#_Toc148087883)

[**1.** **Understanding the requirements** 43](#_Toc148087884)

[**2.** **Explore what the program does for various inputs** 44](#_Toc148087885)

[**3.** **Explore inputs, outputs and identify partitions** 44](#_Toc148087886)

[**4.** **Identify boundary cases** 45](#_Toc148087887)

[**5.** **Devise test cases** 45](#_Toc148087888)

[**6.** **Automate test cases** 45](#_Toc148087889)

[**7.** **Augment the test suite with creativity and experience** 45](#_Toc148087890)

[**LINK DI COLLEGAMENTO AI TEST CASES EFFETTUATI** 45](#_Toc148087891)

[**Home work n°3** 46](#_Toc148087892)

[**Classe: CodeHW3** 46](#_Toc148087893)

[**Metodo: evenOrOdd(int value)** 47](#_Toc148087894)

[ **Devise partition:** 47](#_Toc148087895)

[ **Identify proprieties based on the requirements** 47](#_Toc148087896)

[ **PBT for “value % 2 == 0” and “value % 2 != 0” property** 47](#_Toc148087897)

[ **Collection and Reporting Statistics** 47](#_Toc148087898)

[**Metodo: isPrime(int value)** 48](#_Toc148087899)

[ **Devise partition:** 48](#_Toc148087900)

[ **Identify proprieties based on the requirements** 48](#_Toc148087901)

[ **PBT for “value == Primo” and “value != Primo”** 49](#_Toc148087902)

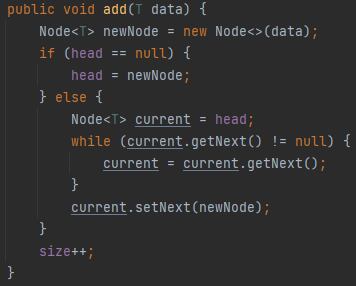
[ **Collection and Reporting Statistics** 50](#_Toc148087903)

[**Considerazioni:** 50](#_Toc148087904)

# **Home work n°1 -> Black-box Testing**

Il codice scelto contiene dei metodi per la gestione di una lista concatenata [LinkedList] generica.

## Metodo n°1 -> add(T data)



### TESTING WORKFLOW FOR SPECIFICATION-BASED TESTING

* **Understanding the requirements**

Il metodo add() fa parte della classe LinkedList per la gestione delle liste concatenate.

La sua funzione è quella di aggiungere un nuovo nodo contenente un dato di tipo generico <T> alla fine della lista.

Ecco cosa fa nello specifico il metodo:

Viene creato un nuovo nodo [newNode] che contiene il dato passato come argomento.

Si verifica se la lista è vuota controllando se il nodo head è nullo.

Se la risposta è affermativa (head è nullo), il nodo newNode diventa il nuovo nodo head della lista, quindi inizializzando la lista con un singolo elemento.

Se la lista non è vuota (head non è nullo), viene inizializzato un nodo current con il valore di head.

Quindi, viene eseguito un ciclo iterativo che scorre la lista finché non si raggiunge l'ultimo nodo.

Questo viene fatto verificando se current.getNext() (il prossimo nodo) è diverso da nullo e, in tal caso, il puntatore current viene spostato al nodo successivo.

Una volta raggiunto l'ultimo nodo (current -> ultimo nodo della lista), il nuovo nodo

[newNode] viene collegato come nodo successivo dell'ultimo nodo (in coda).

Questo significa che la lista, grazie a quest'ultima aggiunta si sarà estesa.

Infine, la dimensione della lista (size) viene incrementata di uno per tenere traccia del numero di elementi totali nella lista.

* **Explore what the program does for various input**

Testiamo cosa fa il programma fornendo vari input e osserviamo il risultato prodotto.

Immagine che contiene testo, schermata, Carattere

Descrizione generata automaticamente

Immagine che contiene testo, Carattere, schermata, tipografia

Descrizione generata automaticamente

* **Explore inputs, outputs and identify partitions**

*Individual inputs:*

* **Input valido**: qualsiasi tipo di dato

*Classes of (expected) outputs:*

* Aggiunta elemento nella LinkedList ed incremento size.
* **Identify boundary cases**

Abbiamo identificato i seguenti boundary cases.

* Aggiunta di un elemento in una lista vuota
* Aggiunta di elementi duplicati

Questi test sono coperti dai test sui vari input.

* **Devise test cases**

Per quanto riguarda il metodo add() abbiamo pensato ai seguenti test:

T1 – “Aggiunta di un elemento in una lista vuota”

T2 – “Aggiunta di elementi cosecutivi”

T3 – “Aggiunta di elementi duplicati”

* **Automate test cases**

**Immagine che contiene testo, schermata, Carattere

Descrizione generata automaticamente**

* Il test T1 aggiunge un singolo elemento in una LinkedList appena istanziata e ne controlla attraverso gli assert il valore, se corrisponde all’inserimento effettuato, e la grandezza (size) della lista stessa.

**Immagine che contiene testo, schermata, Carattere

Descrizione generata automaticamente**

* Il test T2 aggiunge due elementi consecutivi in una LinkedList e ne controlla il corretto inserimento attraverso gli assert.

**Immagine che contiene testo, schermata, Carattere

Descrizione generata automaticamente**

* Il test T3 aggiunge due elementi duplicati, quindi uguale, e ne controlla il corretto inserimento nella lista attraverso gli assert.

Dopo aver lanciato i suddetti test, il risultato ottenuto è stato sin da subito quello sperato.

**Immagine che contiene testo, Carattere, schermata

Descrizione generata automaticamente**

* **Argument the test suite with creativity and experience**

La natura del metodo in esame e i test già scritti dimostrano chiaramente che il metodo in questione esegue operazioni di base e non presenta complessità significative. I test attuali coprono scenari comuni e limiti, come l'aggiunta di elementi in una lista vuota, l'aggiunta di elementi consecutivi e la gestione di elementi duplicati. Inoltre, i test attuali soddisfano gli obiettivi di copertura del codice.L'espansione ulteriore dei casi di test per questo metodo potrebbe comportare uno sforzo aggiuntivo senza apportare significativi benefici in termini di qualità o copertura dei test.

### TESTING WORKFLOW FOR STRUCTURAL TESTING

* **Perform Specification-based testing (7-Steps-Approach)**

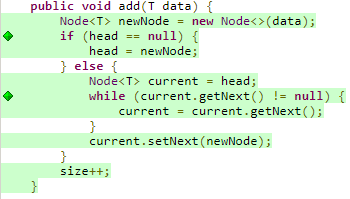
Approccio a 7 steps eseguito nei punti precedenti.

* **Read the implementation: understand the code**

Il metodo add() inserisce un nuovo elemento, del tipo specificato al momento dell’implementazione di una LinkedList, alla fine di una lista concatenata estendendola con quest’ultimo e incrementando la sua dimensione.

* **Run test suite with a code coverage tool (to identify a NON COVERED part in the code)**

Il tool di coverage utilizzato è “jaCOCO” e i risultati ottenuti in seguito al suo utilizzo sono i seguenti:





* **For each piece of code “not covered”:**

Non è necessario eseguire questo punto poiché, come sopra indicato, tutte le linee di codice relative al metodo add() sono coperte dal test.

* **Go back to point 3**

Non è necessario eseguire questo punto poiché, come sopra indicato, tutte le linee di codice relative al metodo add() sono coperte dal test.

ANALISI COPERTURA DEL CODICE

Il tool di code coverage JaCOCO da noi utilizzato sfrutta il criterio della line coverage. Per un’analisi di copertura del codice più approfondita abbiamo deciso di applicare al metodo add() il criterio Condition + Branch coverage.

Immagine che contiene testo, Carattere, schermata, linea

Descrizione generata automaticamente

(il tutto x100)

Eseguendo gli opportuni calcoli, siamo arrivati alla seguente conclusione:

**B** = 2 , **C** = 2 , **NB** = 2 , **NC** = 2

(e quindi)

**C+B** = [(2+2)/(2+2)]\*100 -> metodo add() **100% covered!**

## Metodo n°2 -> remove(T data)

**Immagine che contiene testo, schermata, software

Descrizione generata automaticamente**

### TESTING WORKFLOW FOR SPECIFICATION-BASED TESTING

* **Understanding the requirements**

Il metodo remove(), come lascia intuire dal nome, rimuove un elemento dalla lista eseguendo questi passaggi:

* verifica se la lista è vuota; se lo è restituisce false, indicando che non è stato possibile rimuovere l'elemento.
* se l'elemento da rimuovere è il primo elemento della lista (il nodo head), lo rimuove collegando quest'ultimo al successivo nodo nella lista e ne decrementa la dimensione restituendo true per indicare che l'elemento è stato rimosso con successo.
* se l'elemento da rimuovere non è il primo elemento, cerca l'elemento nella lista attraversando i nodi ed una volta trovato, lo rimuove collegando il nodo precedente al successivo e decrementa la dimensione della lista. Anche qui, ad operazione terminata, vieine restitutito true.
* se l'elemento da rimuovere non è presente nella lista, restituisce false per indicare che l'elemento non è stato trovato e non è stato possibile rimuoverlo.
* **Explore what the program does for various input**

Come anche i test, successivamente, ci dimostreranno; indipendentemente dal tipo di dato con cui andremo ad istanziare la lista il metodo produrrà lo stesso risultato.

Seguono gli allegati con due esempi per tipo di dato:

**Immagine che contiene testo, schermata, Carattere

Descrizione generata automaticamente**

**Immagine che contiene testo, Carattere, schermata

Descrizione generata automaticamente**

* **Explore inputs, outputs and identify partitions**

*Individual inputs:*

* **Input valido**: integer che indichi la posizione dell’elemento da rimuovere
* **Input non valido**: altro tipo di dato/posizione non appartenente alla lista

*Classes of (expected) outputs:*

* IF (input valido) -> Rimozione elemento indicato.
* Else -> sollevamento di un eccezione.
* **Identify boundary cases**

Abbiamo identificato i seguenti boundary cases.

* Rimozione di un elemento in testa alla lista
* Rimozione di un elemento non presente in lista

Questi test sono coperti dai test sui vari input.

* **Devise test cases**

Per quanto riguarda il metodo remove() abbiamo pensato ai seguenti test:

T4 – “Rimozione di un elemento esistente dalla lista”

T5 – “Tentativo di rimuovere un elemento non presente nella lista”

T5(a) – “Rimozione di un elemento dalla testa della lista”

* **Automate test cases**

**Immagine che contiene testo, schermata, Carattere

Descrizione generata automaticamente**

* Il test T4 aggiunge tre elementi in una LinkedList e successivamente ne controlla la rimozione di uno in particolare e il corretto ridimensionamento della size relativa alla lista.

**Immagine che contiene testo, schermata, Carattere

Descrizione generata automaticamente**

* Il Test T5 aggiunge tre elementi in una LinkedList e successivamente tenta di rimuovere un quarto elemento fittizio (quindi non presente in lista) e controlla attraverso gli assert innanzitutto il riscontro sulla rimozione dell’elemento fallita e il non avvenuto ridimensionamento della lista.

**Immagine che contiene testo, schermata, Carattere

Descrizione generata automaticamente**

* **Argument the test suite with creativity and experience**

Ulteriori test implementabili, potrebbero essere i seguenti:

* + Test per rimuovere un elemento nel mezzo: si assicura che l'elemento venga rimosso correttamente dal mezzo della lista e che la dimensione diminuisca.
  + Test per rimuovere un elemento dalla coda: verifica la rimozione di un elemento dalla coda della lista e la diminuzione della dimensione.
  + Test per rimuovere tutti gli elementi: verifica che tutti gli elementi siano rimossi correttamente dalla lista, lasciandola vuota alla fine.

### TESTING WORKFLOW FOR STRUCTURAL TESTING

* **Perform Specification-based testing (7-Steps-Approach)**

Approccio a 7 steps eseguito nei punti precedenti.

* **Read the implementation: understand the code**

Il metodo remove() rimuove un elemento dalla LinkedList verificando se la lista è vuota, se l’elemento da rimuovere è il primo nella lista o se si trova nel mezzo della lista (e avvia le rispettive procedure di gestione in caso di rimozione dell’elemetno), e restituisce true se l’elemento è stato rimosso con successo e false se l’elemento non è stato trovato o la lista è vuota.

* **Run test suite with a code coverage tool (to identify a NON COVERED part in the code)**

Il tool di coverage utilizzato è “jaCOCO” e i risultati ottenuti in seguito al suo utilizzo sono i seguenti:

Immagine che contiene testo, schermata, Carattere, numero

Descrizione generata automaticamente

* **For each piece of code “not covered”:**

Come indicato dal tool, il problema viene riscontrato nel primo if statement e l’avviso relativo è “1 of 2 branches missed”

Immagine che contiene testo, Carattere, schermata, linea

Descrizione generata automaticamente

**B** = 4 , **C** = 3 , **NB** = 4 , **NC** = 4

(e quindi)

**C+B** = [(4+3)/(4+4)]\*100 ->**87%**

* **Go back to point 3**

Per testare quel ramo e quindi coprire la coverage del metodo al 100% si è pensato di aggiungere i seguenti test ->

Immagine che contiene testo, schermata, Carattere, software

Descrizione generata automaticamente

* Il test T5(a) si occupa di aggiungere tre elementi e successivamente rimuovere quello di testa per poi verificare la corretta rimozione; quindi la diminuzione della size e il rilevamento dei valori rimasti.

Immagine che contiene testo, schermata, Carattere

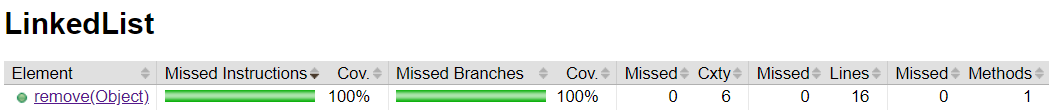
Descrizione generata automaticamente

* Il test T5(b) prova ad effettuare la rimozione di un elemento da una lista vuota. Poiché la lista è vuota, il metodo remove dovrebbe restituire false, e la dimensione della lista rimarrà 0.

Facendo girare di nuovo il tool jaCOCO il risultato è quello sperato, problema risolto al 100%

Immagine che contiene testo, schermata, Carattere, numero

Descrizione generata automaticamente



ANALISI COPERTURA DEL CODICE

Il tool di code coverage JaCOCO da noi utilizzato sfrutta il criterio della line coverage. Per un’analisi di copertura del codice più approfondita abbiamo deciso di applicare al metodo remove() il criterio Condition + Branch coverage.

Immagine che contiene testo, Carattere, schermata, linea

Descrizione generata automaticamente

(il tutto x100)

Eseguendo gli opportuni calcoli, siamo arrivati alla seguente conclusione:

**B** = 4 , **C** = 4 , **NB** = 4 , **NC** = 4

(e quindi)

**C+B** = [(4+4)/(4+4)]\*100 -> metodo remove() **100% covered!**

## Metodo n°3 -> get(int index)

**Immagine che contiene testo, schermata, software, Carattere

Descrizione generata automaticamente**

### TESTING WORKFLOW FOR SPECIFICATION-BASED TESTING

* **Understanding the requirements**

Il metodo get(int index) restituisce l'elemento della LinkedList situato all'indice specificato.

Ecco una spiegazione delle sue azioni:

* Verifica se l'indice specificato è valido. Se l'indice è minore di 0 o maggiore o uguale alla dimensione della lista (size), solleva un'exception per indicare che l'indice è fuori dai limiti della lista.
* Attraversa la lista iterativamente, avanzando da un nodo all'altro fino a quando currentIndex non coincide con l'indice specificato. In sosranza scorre la lista finché non raggiunge l'elemento desiderato.
* Una volta raggiunto l'elemento all'indice specificato, restituisce il valore contenuto in quel nodo tramite current.getData().
* **Explore what the program does for various input**

Come anche i test, successivamente, ci dimostreranno; indipendentemente dal tipo di dato con cui andremo ad istanziare la lista il metodo produrrà lo stesso risultato.

Seguono gli allegati con due esempi per tipo di dato:

**Immagine che contiene testo, schermata, Carattere

Descrizione generata automaticamente**

**Immagine che contiene testo, Carattere, schermata, numero

Descrizione generata automaticamente**

* **Explore inputs, outputs and identify partitions**

*Individual inputs:*

* + **Input valido**: integer che indichi la posizione dell’elemento da ottenere
  + **Input non valido**: altro tipo di dato/posizione non appartenente alla lista

*Classes of (expected) outputs:*

* + IF (input valido) -> Ottenimento dell’elemento indicato.
  + Else -> sollevamento di un eccezione.
* **Identify boundary cases**

Abbiamo identificato i seguenti boundary cases.

* Tentativo di ottenere un elemento da una posizione non valida
* Ottenere un elemento da una lista vuota
* Ottenere un elemento da una posizione maggiore della dimensione

Questi test sono coperti dai test sui vari input.

* **Devise test cases**

Per quanto riguarda il metodo get() abbiamo pensato ai seguenti test:

T6 – “Ottenimento di un elemento esistente dalla lista”

T7 – “Tentativo di ottenere un elemento dalla posizione non valida”

T7(a) – “Tentativo di ottenere un elemento da una lista vuota”

T7(b) – “Tentativo di ottenere un elemento da una posizione negativa”

T7(c) – “Tentativo di ottenere un elemento da una posizione maggiore della dimensione”

* **Automate test cases**

**Immagine che contiene testo, schermata, Carattere

Descrizione generata automaticamente**

* Il test T6 aggiunge tre elementi in una LinkedList e testa il metodo get() attraverso un assert che verifica il combaciare dell’elemento richiesto con quello ottenuto/pescato dal metodo.

**Immagine che contiene testo, schermata, Carattere

Descrizione generata automaticamente**

* Il test T7, dopo aver inserito tre elementi in una LinkedList, cerca di prelevare un elemento in una posizione “vuota” e quindi inesistente e si assicura di avere un riscontro positivo grazie ad un assert che conferma il sollevamento di una exception.

Immagine che contiene testo, schermata, Carattere

Descrizione generata automaticamente

* **Argument the test suite with creativity and experience**

Ulteriori test implementabili, potrebbero essere i seguenti:

* + Test per ottenere un elemento da una lista con un solo elemento: verifica che il metodo restituisca l'unico elemento correttamente quando la lista ha un solo elemento.
  + Test per ottenere un elemento da un indice intermedio: verifica che il metodo restituisca l'elemento corretto.
  + Test per ottenere un elemento dall'ultimo indice: verifica che il metodo restituisca l'elemento corretto quando la lista contiene più elementi e viene fornito l'ultimo indice.

### TESTING WORKFLOW FOR STRUCTURAL TESTING

* **Perform Specification-based testing (7-Steps-Approach)**

Approccio a 7 steps eseguito nei punti precedenti.

* **Read the implementation: understand the code**

Il metodo get(int index) recupera l'elemento dalla LinkedList in base all'indice fornito, ma prima verifica se l'indice è valido per evitare errori di accesso out-of-bounds. Se l'indice è valido, restituisce il valore dell'elemento corrispondente all'indice specificato.

* **Run test suite with a code coverage tool (to identify a NON COVERED part in the code)**

Il tool di coverage utilizzato è “jaCOCO” e i risultati ottenuti in seguito al suo utilizzo sono i seguenti:

**Immagine che contiene testo, schermata, Carattere, linea

Descrizione generata automaticamente**

* **For each piece of code “not covered”:**

Come segnalato dal tool di coverage, i test da noi scritti non coprono interamente tutte le branche del metodo, quindi analizzando meglio abbiamo pensato di aggiungerne qualche altro.

* **Go back to point 3**

Riprendendo dal “point 3” i test che abbiamo pensato di aggiungere sono i seguenti:

Immagine che contiene testo, schermata, Carattere

Descrizione generata automaticamente

* Il test T7(a) si occupa di testare il caso in cui si cerchi di ottenere un elemento da una LinkedList vuota.

Immagine che contiene testo, schermata, Carattere

Descrizione generata automaticamente

* Il test T7(b) si occupa di testare il caso in cui si cerchi di pescare un elemento negativo attraverso un assert che rileva il sollevamento di una exception.

Immagine che contiene testo, software, Carattere, Software multimediale

Descrizione generata automaticamente

* Il test T7(c) si occupa di testare il caso in cui si voglia ottenere un elemento posizionato in una posizione maggiore della size e quindi inesistente.

Successivamente, utilizzando il tool di coverage, il risultato dopo l’aggiunta di questi ulteriori test è stato quello sperato.

Immagine che contiene testo, schermata, Carattere

Descrizione generata automaticamente

ANALISI COPERTURA DEL CODICE

Il tool di code coverage JaCOCO da noi utilizzato sfrutta il criterio della line coverage. Per un’analisi di copertura del codice più approfondita abbiamo deciso di applicare al metodo get() il criterio Condition + Branch coverage.

Immagine che contiene testo, Carattere, schermata, linea

Descrizione generata automaticamente

(il tutto x100)

Eseguendo gli opportuni calcoli, siamo arrivati alla seguente conclusione:

**B** = 2 , **C** = 6 , **NB** = 2 , **NC** = 6

(e quindi)

**C+B** = [(2+6)/(2+6)]\*100 -> metodo get() **100% covered!**

## Metodo n°4 -> size() | isEmpty()

**Immagine che contiene testo, Carattere, schermata

Descrizione generata automaticamente**

### TESTING WORKFLOW FOR SPECIFICATION-BASED TESTING

* **Understanding the requirements**

Il metodo size() ritorna la grandezza della LinkedList, mentre il metodo isEmpty() restituisce un valore booleano TRUE or FALSE a seconda se la LinkedList istanziata contiene o meno elementi.

* **Explore what the program does for various input**

Utilizzando la LinkedList precedentemente istanziata, per verificare il metodo get(), invochiamo i metodi sopra citati:

****

**Immagine che contiene testo, Carattere, schermata

Descrizione generata automaticamente**

* **Explore inputs, outputs and identify partitions**

Il metodo size una volta invocato restituisce un Integer mentre il metodo isEmpty restituisce un flag True o False (valore Booleano)

* **Devise test cases**

Per quanto riguarda i metodi size() ed isEmpty() abbiamo pensato di unificare i test in uno singolo ma completo:

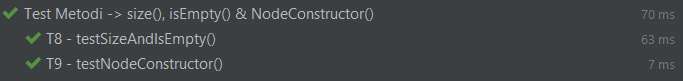
T8 – “TestSizeAndIsEmpty()”

* **Automate test cases**

**Immagine che contiene testo, schermata, Carattere

Descrizione generata automaticamente**

* Il test T8 copre due metodi: isEmpty() e size():
* Per il testing relativo al primo metodo attraverso degli assert verifichiamo il valore restituito dal metodo prima e dopo l’inserimento di un elemento nella lista.
* Molto simile, per il secondo metodo, andiamo a controllare con degli assert la grandezza restituita ad elemento aggiunto e successivamente ad elemento rimosso.



* **Argument the test suite with creativity and experience**

Essendo size() ed isEmpty() metodi, seppur utili, molto semplici i testi implementati coprono tutti i possibili casi di utilizzo.

### TESTING WORKFLOW FOR STRUCTURAL TESTING

* **Perform Specification-based testing (7-Steps-Approach)**

Approccio a 7 steps eseguito nei punti precedenti.

* **Read the implementation: understand the code**

Il metodo size() restituisce il valore relativo alla grandezza della LinkedList, mentre il metodo isEmpty() restituisce un flag [TRUE or FALSE] in base alla presenza o meno di elementi nella lista.

* **Run test suite with a code coverage tool (to identify a NON COVERED part in the code)**

Il tool di coverage utilizzato è “jaCOCO” e i risultati ottenuti in seguito al suo utilizzo sono i seguenti:

**Immagine che contiene testo, Carattere, schermata, linea

Descrizione generata automaticamente**

* **For each piece of code “not covered”:**

Non è necessario eseguire questo punto poiché, come sopra indicato, tutte le linee di codice relative ai metodi size() & isEmpty() sono coperte dal test.

* **Go back to point 3**

Non è necessario eseguire questo punto poiché, come sopra indicato, tutte le linee di codice relative ai metodi size() & isEmpty() sono coperte dal test.

## Metodo n°5 -> node(T data)

**Immagine che contiene testo, schermata, software

Descrizione generata automaticamente**

### TESTING WORKFLOW FOR SPECIFICATION-BASED TESTING

* **Understanding the requirements**

La classe Node<T> è una classe generica che rappresenta un nodo in una LinkedList.

Ecco le sue principali caratteristiche e dei suoi metodi:

* private T data: questo campo memorizza il valore (dato) contenuto nel nodo.
* private Node<T> next: questo campo rappresenta il riferimento al nodo successivo nella lista collegata.
* public Node(T data): questo è il costruttore della classe Node. Riceve un valore di tipo generico T e crea un nuovo nodo con quel valore. Inizializza il campo data con il valore fornito e imposta next su null perché inizialmente il nodo non ha un successivo.
* public T getData(): questo metodo restituisce il valore (dato) contenuto nel nodo.
* public Node<T> getNext(): questo metodo restituisce il riferimento al nodo successivo nella lista collegata.
* public void setNext(Node<T> next): questo metodo consente di impostare il riferimento al nodo successivo. Permette di collegare il nodo corrente a un altro nodo, in modo da poter costruire una lista collegata.
* **Explore what the program does for various input**

//costruttore

* **Explore inputs, outputs and identify partitions**

//costruttore

* **Identify boundary cases**

La classe Node<T> in sé non ha direttamente boundary cases, poiché è principalmente una rappresentazione di un nodo in una LinkedList e contiene solo campi dati e metodi di accesso.

Tuttavia, l'uso di oggetti Node<T> all'interno di una LinkedList può comportare boundary cases, che nel nostro caso, sono gestiti nelle operazioni di inserimento, rimozione o accesso alla LinkedList in cui questi nodi sono utilizzati.

* **Devise test cases**

Per quanto concerne la classe Node e il suo metodo costruttore abbiamo pensato al seguente test:

(I metodi getData(), getNext() e setNext() sono stati esclusi dai test poiché non ne mostrano la necessità)

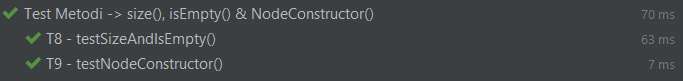
T9 – “testNodeConstructor()”

* **Automate test cases**

**Immagine che contiene testo, schermata, Carattere

Descrizione generata automaticamente**

* Il test T9 si assicura che il costruttore della classe Node sia in grado di creare correttamente un nodo con i dati forniti e che il riferimento al nodo successivo sia inizializzato correttamente come nullo se il nodo è l'ultimo nella lista.



* **Argument the test suite with creativity and experience**

//costruttore

### TESTING WORKFLOW FOR STRUCTURAL TESTING

* **Perform Specification-based testing (7-Steps-Approach)**

Approccio a 7 steps eseguito nei punti precedenti.

* **Read the implementation: understand the code**

Questa classe rappresenta un nodo generico in una LinkedList, con la capacità di memorizzare un valore di tipo generico T e di collegarsi al nodo successivo nella lista. È un componente fondamentale per la costruzione di liste.

* **Run test suite with a code coverage tool (to identify a NON COVERED part in the code)**

Il tool di coverage utilizzato è “jaCOCO” e i risultati ottenuti in seguito al suo utilizzo sono i seguenti:

Immagine che contiene testo, schermata, Carattere, numero

Descrizione generata automaticamente

* **For each piece of code “not covered”:**

Non è necessario eseguire questo punto poiché, come sopra indicato, tutte le linee di codice relative alla classe Node<T> e al costruttore soggetto al test sono coperte dal test.

* **Go back to point 3**

Non è necessario eseguire questo punto poiché, come sopra indicato, tutte le linee di codice relative alla classe Node<T> e al costruttore soggetto al test sono coperte dal test.

ANALISI COPERTURA DEL CODICE

Il tool di code coverage JaCOCO da noi utilizzato sfrutta il criterio della line coverage. Per un’analisi di copertura del codice più approfondita abbiamo deciso di applicare il criterio Condition + Branch coverage.

Immagine che contiene testo, Carattere, schermata, linea

Descrizione generata automaticamente

(il tutto x100)

Eseguendo gli opportuni calcoli, siamo arrivati alla seguente conclusione:

**B** = 1 , **C** = 1 , **NB** = 1 , **NC** = 1

(e quindi)

**C+B** = **100% covered!**

**Re-cap risultati test eseguiti:**

Immagine che contiene schermata, testo, linea

Descrizione generata automaticamente

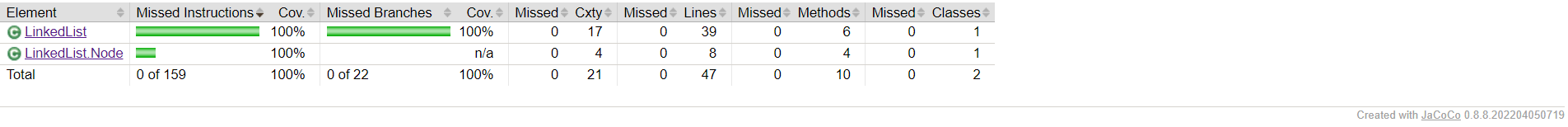


Immagine che contiene testo, schermata, numero, Carattere

Descrizione generata automaticamente

*MEAVEN TEST*

Dopo aver configurato il file pom.xml, che include le dipendenze necessarie per eseguire i test JUnit, abbiamo scelto di eseguire un ulteriore test utilizzando il plugin Maven fornito da Apache.

Questo approccio contribuisce a garantire una maggiore sicurezza e qualità del nostro codice attraverso l'esecuzione di test automatizzati.

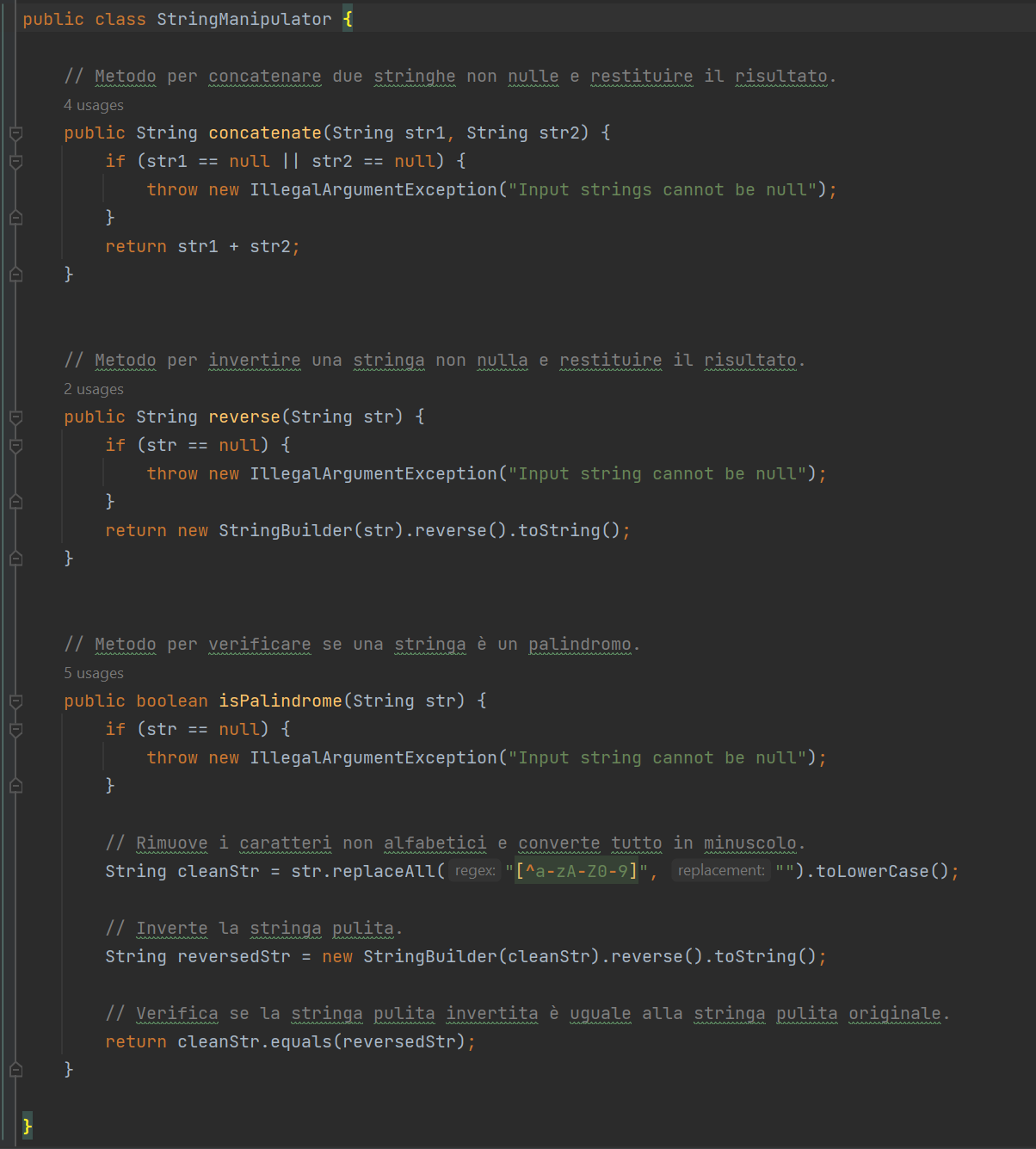
Il riscontro è stato il seguente:

Immagine che contiene testo, elettronica, schermata, software

Descrizione generata automaticamente

# **Home work n°2 -> White-box Testing**

## **Classe:** StringManipulator



**Read the implementation: understand the code**

Il codice rappresenta una classe chiamata `StringManipulator`.

Questa classe contiene tre metodi:

1. **`concatenate(String str1, String str2)`:**

Questo metodo prende due stringhe come input (`str1` e `str2`) e le concatena restituendo il risultato. Se una delle due stringhe è nulla, viene lanciata un'eccezione `IllegalArgumentException`.

1. **`reverse(String str)`:**

Questo metodo prende una stringa come input (`str`) e ne restituisce la versione invertita. Se la stringa di input è nulla, viene lanciata un'eccezione `IllegalArgumentException`.

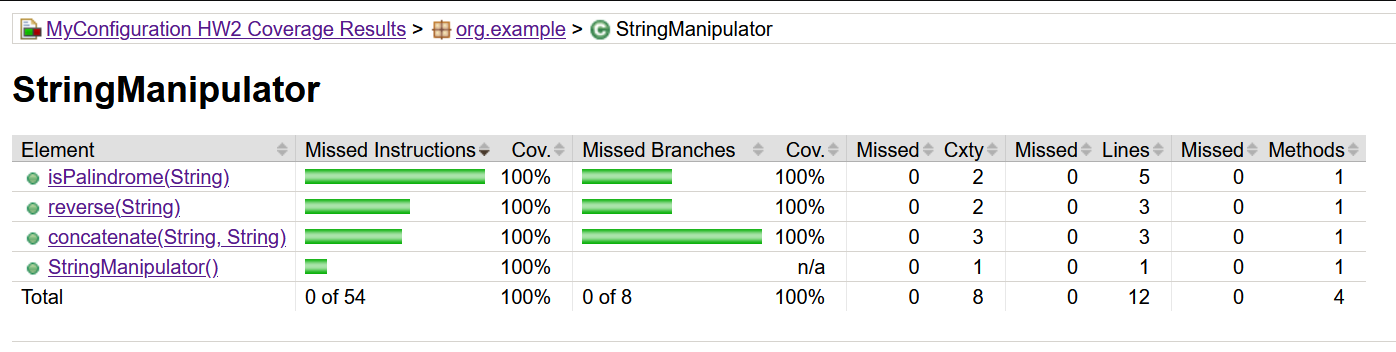
1. **`isPalindrome(String str)`:**

Questo metodo prende una stringa come input (`str`) e verifica se è un palindromo, ossia se può essere letta allo stesso modo da sinistra a destra e da destra a sinistra. Per fare ciò, rimuove i caratteri non alfabetici, converte tutto in minuscolo, inverte la stringa risultante e la confronta con la stringa originale dopo la pulizia. Se la stringa di input è nulla, viene lanciata un'eccezione `IllegalArgumentException`.

Questi metodi sono funzioni utili per manipolare e analizzare stringhe, ma è importante assicurarsi che vengano gestiti correttamente i casi in cui le stringhe di input sono null.

**100% di Code Coverage con il minor numero di test**

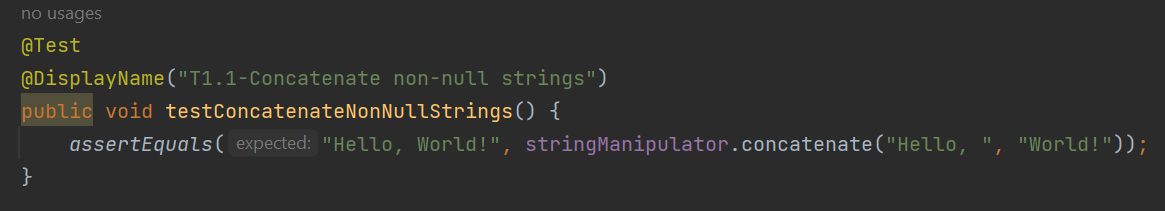
**Per raggiungere il 100% di Code Coverage per la classe StringManipulator ci è bastato implementare i seguenti test:**

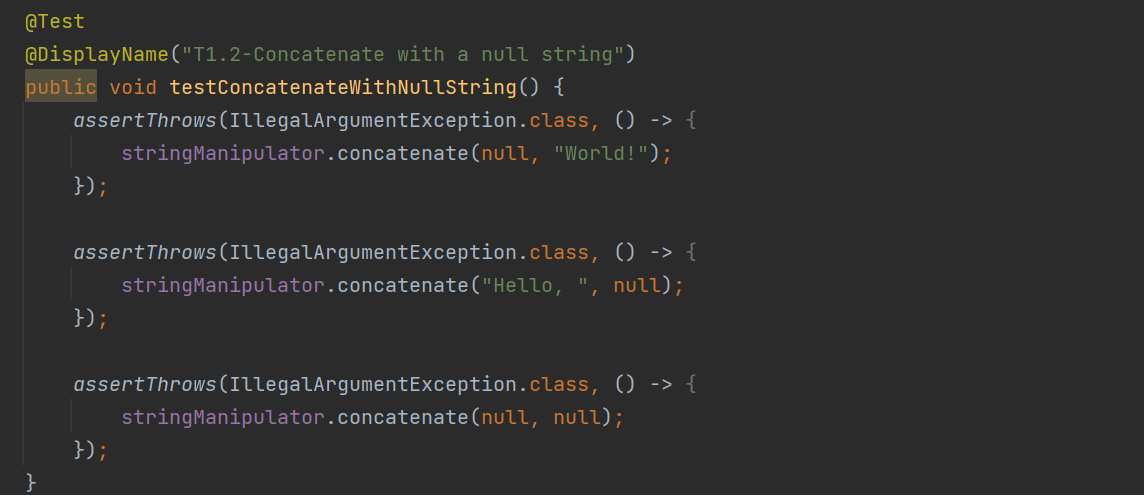
****

🡪Per testare il metodo **“concatenate”** abbiamo implementato i seguenti due test:

* **-T1.1**-Concatenate non-null strings
* **-T1.2**-Concatenate with a null string

**Implementazione dei casi di test:**



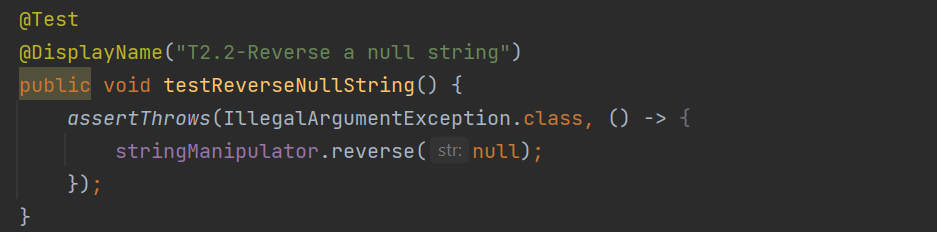


🡪Per testare il metodo **“reverse”** abbiamo implementato i seguenti due test:

* **-T2.1**-Reverse a non-null string
* **-T2.2**-Reverse a null string

**Implementazione dei casi di test:** Immagine che contiene schermata, testo, Carattere

Descrizione generata automaticamente



🡪Per testare il metodo **“isPalindrome”** abbiamo implementato i seguenti tre test:

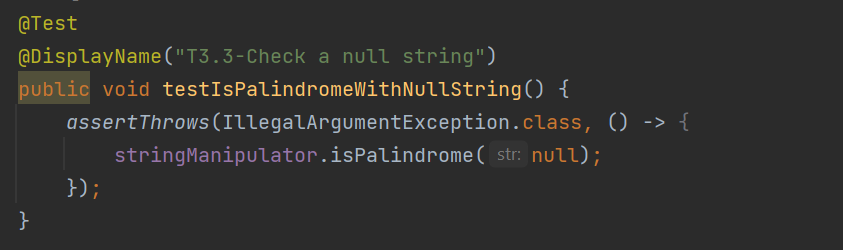
* **-T3.1**-Check a palindorme string
* **-T3.2**-Check a null string
* **-T3.3**-Check a non-palindrome string

**Implementazione dei casi di test:** Immagine che contiene testo, schermata, Carattere

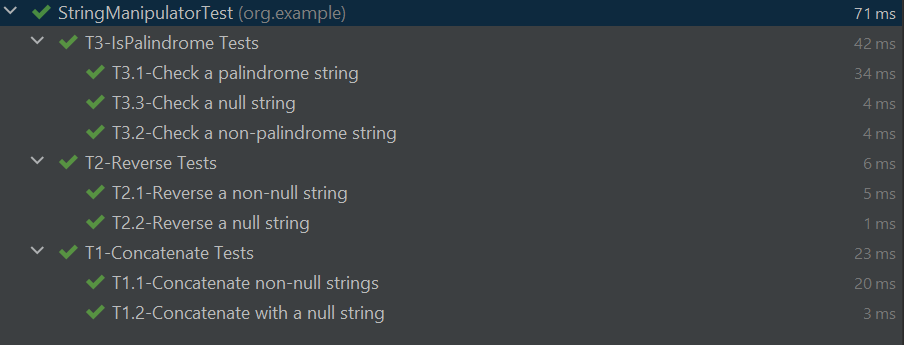
Descrizione generata automaticamente

Immagine che contiene testo, schermata, Carattere

Descrizione generata automaticamente



🡪**Abbiamo lanciato i test, ricevendo risultati positivi:**



**🡪Code Coverage al 100%:**



“Anche se, come si evince dal risultato della Code Coverage, abbiamo raggiunto il 100%, risulta evidente che ci sono molti casi di test che non sono stati implementati.

Per individuare i casi di test che non sono stati implementati, abbiamo utilizzato il testing workflow per gli Specification-based testing.”

### **Specification-based testing Metodo1: concatenate(String str1, String str2)**

Immagine che contiene testo, schermata, Carattere

Descrizione generata automaticamente

**Read the implementation: understand the code**

Il metodo `concatenate(String str1, String str2)` ha la responsabilità di concatenare due stringhe non nulle e restituire il risultato come una nuova stringa.

#### **TESTING WORKFLOW STRUCTURAL TESTING**

##### **Understanding the requirements**

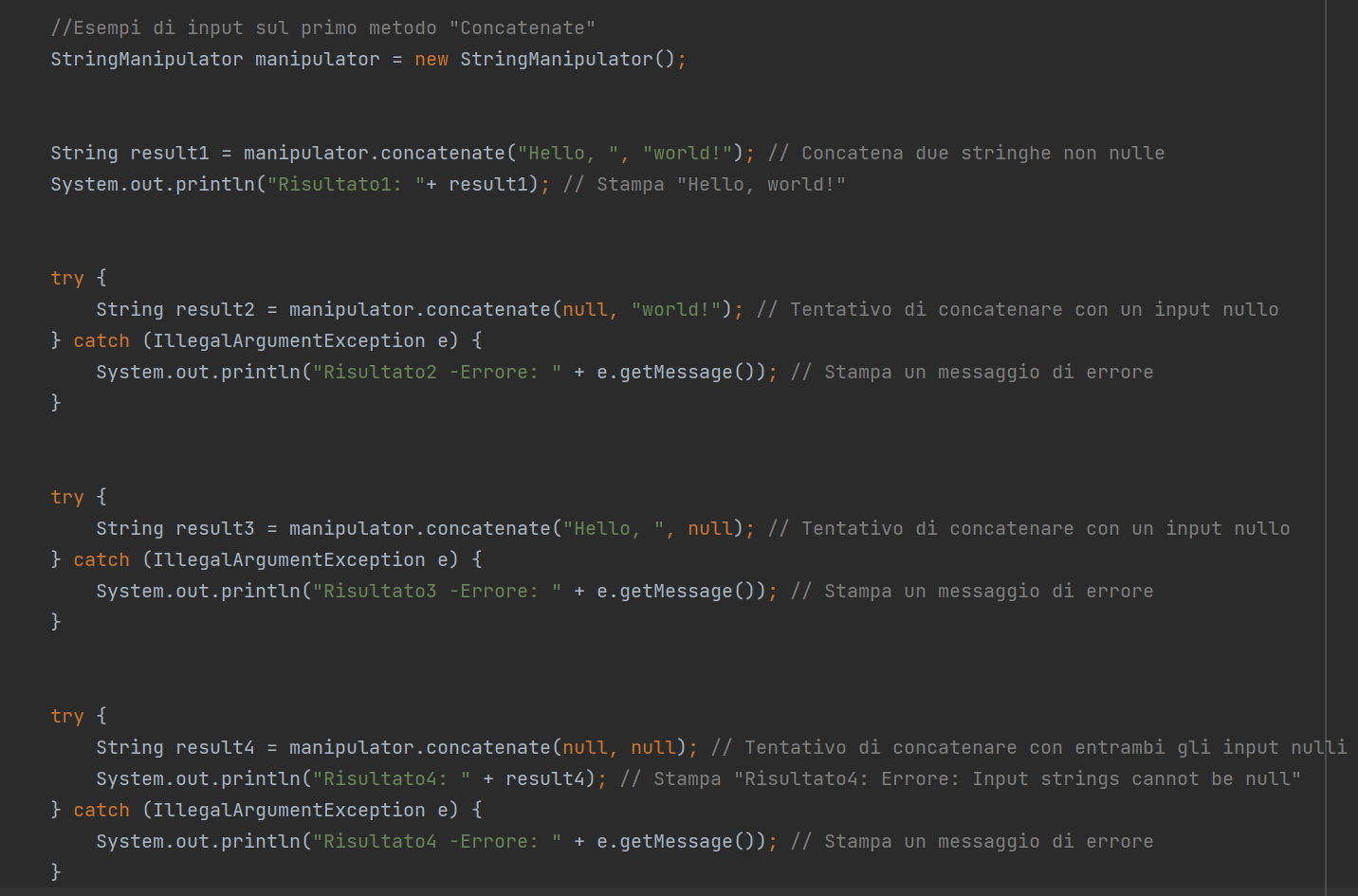
* Il metodo deve accettare due stringhe come input.
* Se una delle due stringhe di input è nulla, il metodo deve generare un'eccezione IllegalArgumentException.

🡪Altrimenti, il metodo deve concatenare le due stringhe e restituire il risultato.

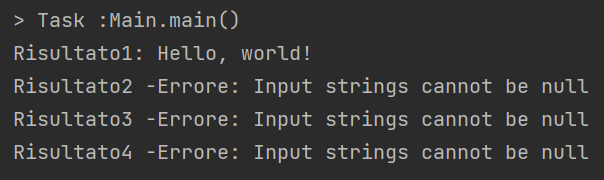
##### **Explore what the program does for various inputs**

Testiamo cosa fa il programma fornendogli valori di tipo String e valori null come input e osservando cosa fornisce in output.

* Entrambe le stringhe di input sono non nulle.
* Il primo input è nullo e il secondo è non nullo.
* Il primo input è non nullo e il secondo è nullo.
* Entrambi gli input sono nulli.



**Outputs:**



##### **Explore inputs, outputs and identify partitions**

Identifichiamo le partizioni principali degli input:

Individual inputs:

* **Input valido**: Entrambe le stringhe di input sono non nulle.
* **Input non valido**: con il primo parametro nullo.
* **Input non valido**: con il secondo parametro nullo.
* **Input non valido:** Entrambi gli input sono nulli.

Classes of (expected) outputs:

* **If** input valido: Stringa concatenata
* **Else** : Eccezione

##### **Identify boundary cases**

I casi limite possono includere:

* Stringhe vuote come input.
* Stringhe molto lunghe come input.
* Stringhe contenenti caratteri speciali o spazi.
* Combinazioni di input che includono stringhe non nulle e nulli.

##### **Devise test cases**

In base alle partizioni e ai casi limite identificati, abbiamo ideato casi di test specifici per verificare il comportamento del metodo:

* -Testiamo il caso in cui le stringhe siano vuote.

**-T1.3**-Test con stringhe vuote.

* -Testiamo il caso in cui le stringhe siano molto lunghe.

**-T1.4**-Test con stringhe molto lunghe.

* -Testiamo il caso in cui le stringhe contengano caratteri speciali.

**-T1.5**-Test con stringhe contenenti caratteri speciali.

##### **Automate test cases**

Una volta ideati i casi di test, potremo scrivere codice per automatizzare l'esecuzione di questi test utilizzando il framework di testing JUnit per questo scopo.

##### **Augment the test suite with creativity and experience**

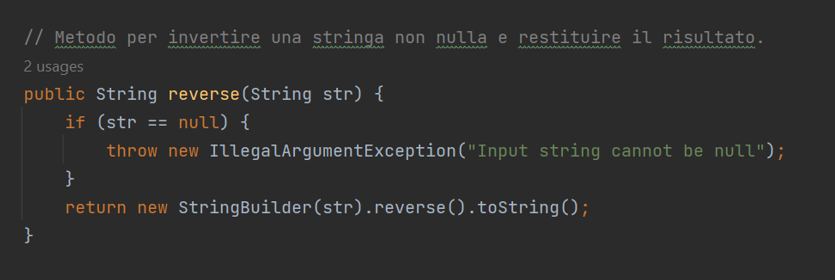
Possiamo arricchire ulteriormente la suite di test considerando scenari speciali, casi di errore, e situazioni che potrebbero emergere in condizioni reali di utilizzo. Possiamo anche eseguire test di performance o di carico se necessario.

* Test di performance:

eseguire test per misurare le prestazioni del metodo quando si concatenano grandi quantità di dati.

##### **LINK DI COLLEGAMENTO AI TEST CASES EFFETTUATI**

### **Specification-based testing Metodo2: reverse(String str)**



#### **TESTING WORKFLOW STRUCTURAL TESTING**

##### **Understanding the requirements**

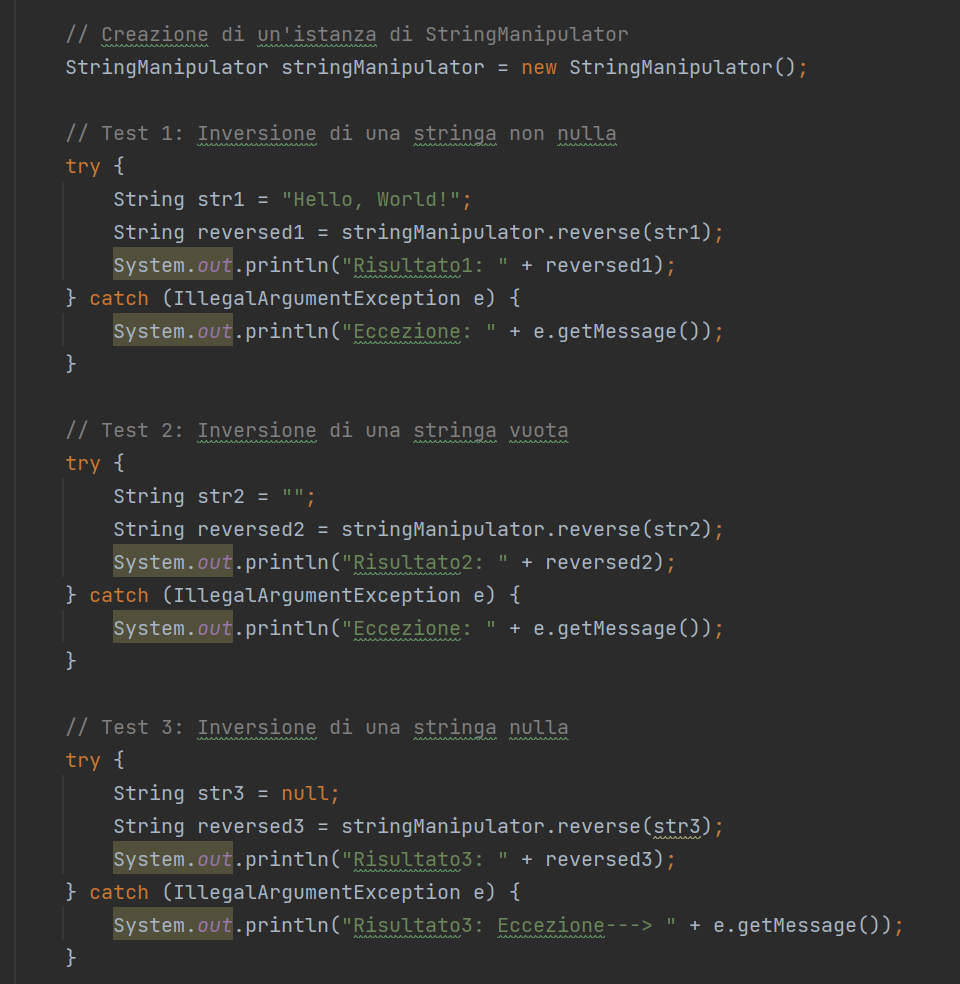
* Il metodo accetta una stringa come input.
* Se la stringa è nulla, il metodo deve generare un'eccezione IllegalArgumentException.

🡪Altrimenti, il metodo deve invertire la stringa e restituire il risultato.

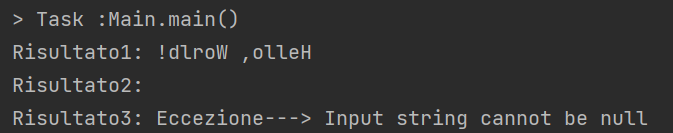
##### **Explore what the program does for various inputs**

Testiamo cosa fa il programma fornendogli valori di tipo String e valori null come input e osservando cosa fornisce in output.

* La stringa di input è non nulla e contiene più di un carattere.
* La stringa di input è vuota “non contiene caratteri”.
* La stringa di input è null.



**Outputs:**

****

##### **Explore inputs, outputs and identify partitions**

Identifichiamo le partizioni principali degli input:

Individual inputs:

* **Input valido**: Stringa contenente caratteri.
* **Input valido**: Stringa vuota.
* **Input non valido**: Stringa nulla.

Classes of (expected) outputs:

* **If** input valido: Stringa invertita
* **Else** : Eccezione

##### **Identify boundary cases**

Nel metodo `reverse` fornito, i "boundary cases" si riferiscono agli scenari in cui il comportamento del metodo potrebbe essere diverso o significativo. In questo caso, i boundary cases includono:

* **Input non valido**: Questo è il caso in cui `str` è `null`. Il metodo gestisce questo caso sollevando un'eccezione `IllegalArgumentException` con il messaggio "Input string cannot be null".
* **Input vuoto**: Questo è il caso in cui `str` è una stringa vuota (`""`). Il metodo non solleva eccezioni per questo caso, ma restituirà semplicemente una stringa vuota come risultato dell'inversione.
* **Input con contenuto non vuoto**: Questo è il caso in cui `str` è una stringa con almeno un carattere. Il metodo eseguirà l'inversione della stringa e restituirà la stringa invertita come risultato.

Quindi, i boundary cases in questo codice riguardano principalmente l'input `null` e l'input vuoto, in cui il metodo gestisce esplicitamente l'input `null` sollevando un'eccezione, ma accetta l'input vuoto come valido e restituisce una stringa vuota come risultato dell'inversione.

##### **Devise test cases**

In base alle partizioni e ai casi limite identificati, abbiamo ideato casi di test specifici per verificare il comportamento del metodo:

* -Testiamo il caso in cui le stringhe siano molto lunghe.

**-T2.3**-Test con stringhe molto lunghe.

* -Testiamo il caso in cui le stringhe contengano caratteri speciali.

**-T2.4**-Test con stringhe contenenti caratteri speciali.

* -Testiamo il caso in cui le stringhe non contengono caratteri.

**-T2.5**-Test con stringhe vuote.

##### **Automate test cases**

Una volta ideati i casi di test, potremo scrivere codice per automatizzare l'esecuzione di questi test utilizzando il framework di testing JUnit per questo scopo.

##### **Augment the test suite with creativity and experience**

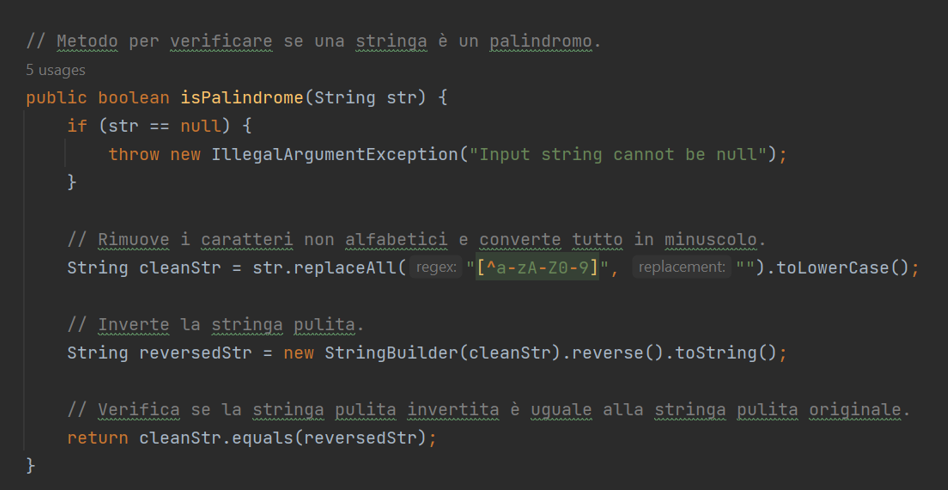
Possiamo arricchire ulteriormente la suite di test considerando scenari speciali, casi di errore, e situazioni che potrebbero emergere in condizioni reali di utilizzo. Possiamo anche eseguire test di performance o di carico se necessario.

* Test di performance:

eseguire test per misurare le prestazioni del metodo quando si invertono grandi quantità di dati.

##### **LINK DI COLLEGAMENTO AI TEST CASES EFFETTUATI**

### **Specification-based testing Metodo3: isPalindrome(String str)**



#### **TESTING WORKFLOW STRUCTURAL TESTING**

##### **Understanding the requirements**

* Il metodo accetta una stringa come input.
* Se la stringa è nulla, il metodo deve generare un'eccezione IllegalArgumentException.
* Altrimenti, il metodo restituisce:

🡪 True se la stringa è palindroma.

🡪False se la stringa non è palindroma.

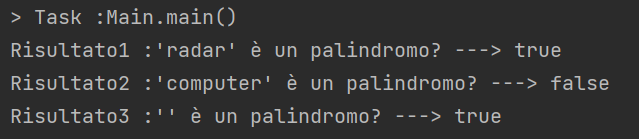
##### **Explore what the program does for various inputs**

Testiamo cosa fa il programma fornendogli valori di tipo String come input e osservando cosa fornisce in output.

* La stringa di input è non nulla e contiene più di un carattere.
* La stringa di input è vuota “non contiene caratteri”.



**Outputs:**

****

##### **Explore inputs, outputs and identify partitions**

Identifichiamo le partizioni principali degli input:

Individual inputs:

* **Input valido**: Stringa contenente caratteri.
* **Input valido**: Stringa vuota.
* **Input non valido**: Stringa nulla.

Classes of (expected) outputs:

* **If** input valido:

**🡪True** se Stringa palindroma

**🡪Altrimenti False** se Stringa non palindroma

* **Else** : Eccezione

##### **Identify boundary cases**

I **"boundary cases",** o casi limite, per il metodo `isPalindrome` che prendiamo in considerazione includono

* Stringhe vuote come input.
* Stringhe molto lunghe come input.
* Stringhe contenenti caratteri speciali o spazi.
* Combinazioni di input che includono stringhe non nulle e nulli.

**E’ importante assicurarsi che il metodo `isPalindrome` gestisca correttamente tutti questi casi limite per garantire una corretta verifica dei palindromi.**

##### **Devise test cases**

* -Testiamo il caso in cui le stringhe siano molto lunghe.

**-T3.4**-Test con stringhe molto lunghe.

* -Testiamo il caso in cui le stringhe contengano caratteri speciali.

**-T3.5**-Test con stringhe contenenti caratteri speciali.

##### **Automate test cases**

Una volta ideati i casi di test, potremo scrivere codice per automatizzare l'esecuzione di questi test utilizzando il framework di testing JUnit per questo scopo.

##### **Augment the test suite with creativity and experience**

Possiamo arricchire ulteriormente la suite di test considerando scenari speciali, casi di errore, e situazioni che potrebbero emergere in condizioni reali di utilizzo. Possiamo anche eseguire test di performance o di carico se necessario.

* Test di performance:

eseguire test per misurare le prestazioni del metodo quando si effettua la verifica su grandi quantità di dati.

##### **LINK DI COLLEGAMENTO AI TEST CASES EFFETTUATI**

# **Home work n°3**

## **Classe: CodeHW3**

Immagine che contiene testo, schermata, software

Descrizione generata automaticamente

### **Metodo: evenOrOdd(int value)**

Questo metodo accetta un parametro value di tipo intero.

Esamina il valore di value per determinare se è pari o dispari.

🡪Se il valore di value è divisibile per 2 (cioè il resto della divisione per 2 è uguale a zero), restituisce true, il che indica che value è un numero pari.

🡪In caso contrario, restituisce false, indicando che value è un numero dispari.

#### **Devise partition:**

🡪**value % 2 == 0**

🡪**value % 2 != 0**

#### **Identify proprieties based on the requirements**

**🡪True**: Se value è un numero pari, allora il metodo restituisce True.

**🡪False**: Se value non è un numero dispari, allora il metodo restituisce False.

#### **PBT for “value % 2 == 0” and “value % 2 != 0” property**

**Immagine che contiene testo, schermata, Carattere

Descrizione generata automaticamente**



#### **Collection and Reporting Statistics**

Il report delle statistiche generato è in allegato con il nome di “Test-Results – ConfigHW3.html”.

### **Metodo: isPrime(int value)**

Questo metodo accetta un parametro value di tipo intero.

Verifica se value è un numero primo o non primo.

Inizia con alcune condizioni speciali: se value è minore o uguale a 1, restituisce false (poiché i numeri minori o uguali a 1 non sono primi), e se value è minore o uguale a 3, restituisce true (poiché 2 e 3 sono numeri primi).

Successivamente, esegue un ciclo che verifica se value è divisibile per qualsiasi numero che segue una sequenza specifica: inizia con 5, verifica se value è divisibile per i o per i + 2, quindi incrementa i di 6 unità. Questo ciclo è un'ottimizzazione che riduce il numero di divisioni necessarie per determinare se un numero è primo.

Se il ciclo non trova alcun divisore, il metodo restituisce true, indicando che value è un numero primo. In caso contrario, restituisce false, indicando che value non è un numero primo.

#### **Devise partition:**

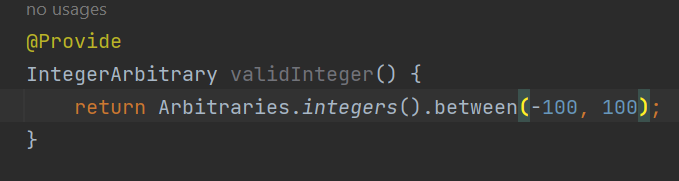
🡪**value == isPrimo**

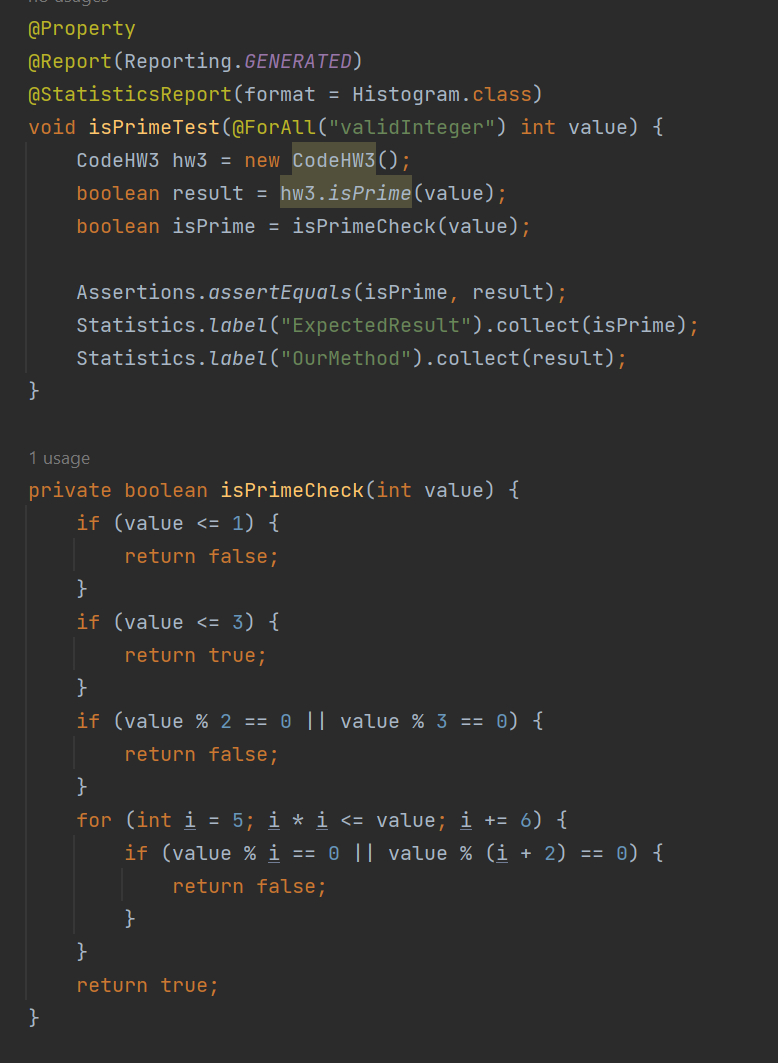
🡪**value != isPrimo**

#### **Identify proprieties based on the requirements**

* **True**: Se value è un numero primo, allora il metodo restituisce True.
* **False**: Se value non è un numero primo, allora il metodo restituisce False

#### **PBT for “value == Primo” and “value != Primo”**



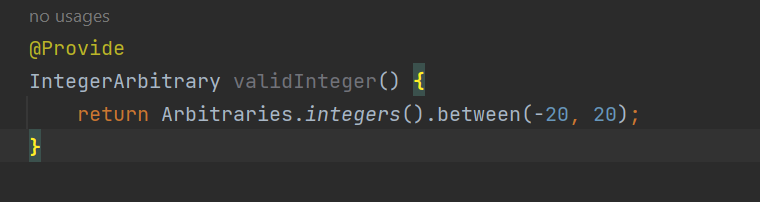


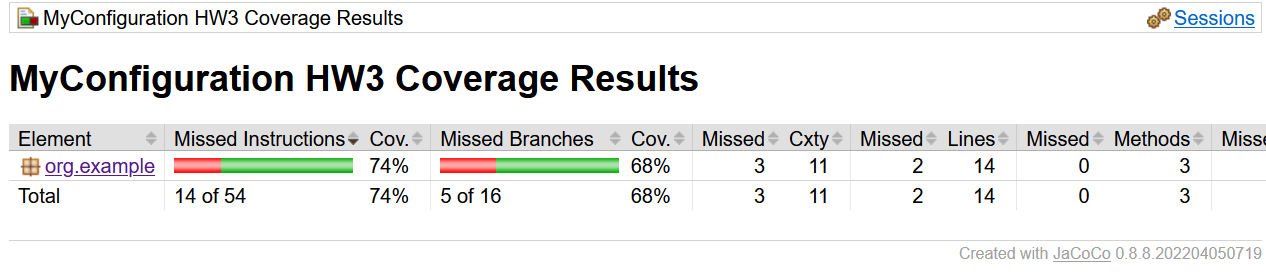
#### **Collection and Reporting Statistics**

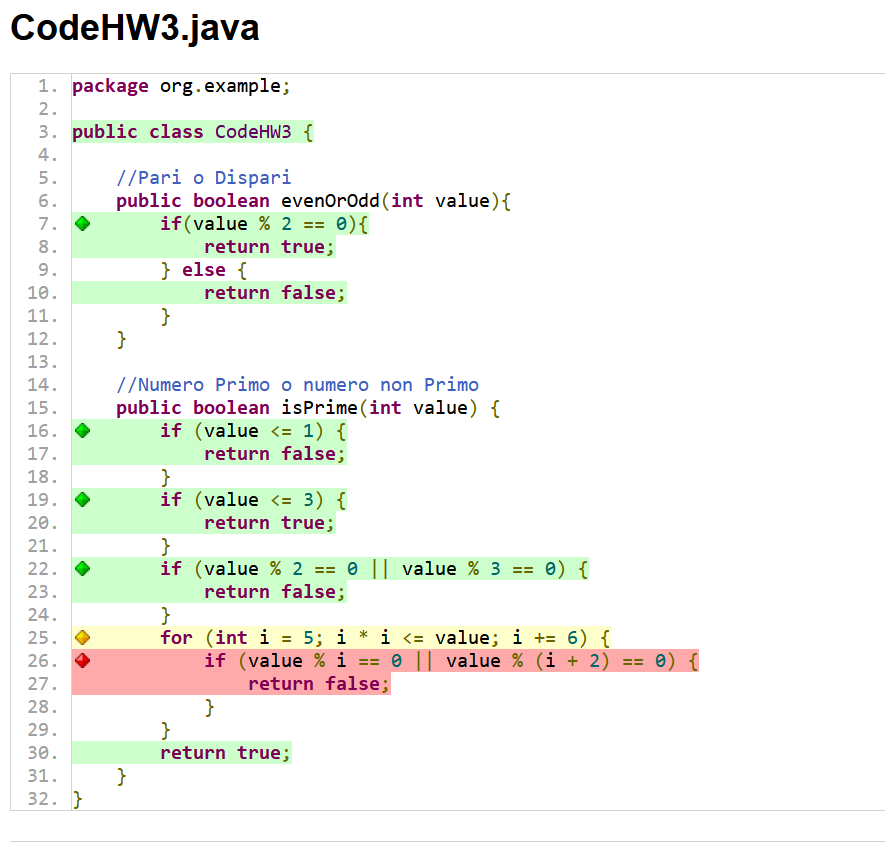
Il report delle statistiche generato è in allegato con il nome di “Test-Results – ConfigHW3.html”.

### **Considerazioni:**

Effettuando i test con un intervallo di range così specificato -> (-20 ,20), utilizzando il tool di Code Coverage JaCoco, abbiamo notato che il metodo isPrimeCheck (int value) non veniva testato al 100%



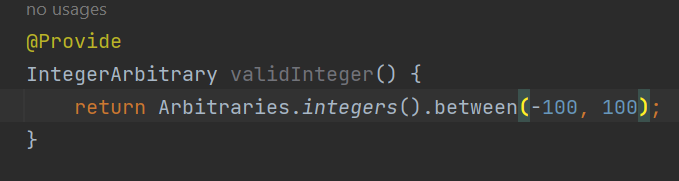


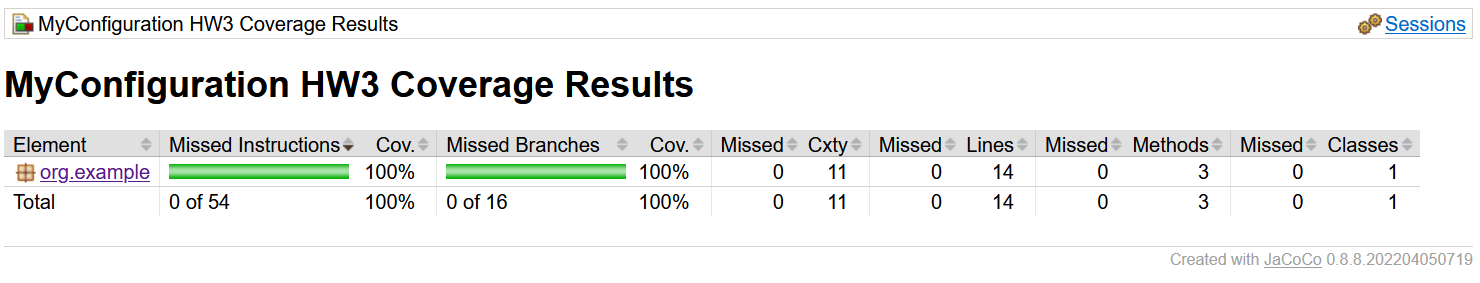


Dopo un attenta riflessione abbiamo stabilito che il problema era dovuto al fatto che il range di valori utilizzato era troppo piccolo e perciò non permetteva di passare in ingresso valori che testassero la condizione if che si trova all’interno del for;

**in particolare:**

🡪 Se il generatore di valori casuali non produce un valore di value che soddisfa la condizione i \* i <= value, il ciclo for potrebbe non essere eseguito nel contesto di un particolare test.  
  
Per questo motivo abbiamo deciso di ampliare il range di valori generati, considerando i valori che sono compresi nell’intervallo (-100,100) , ottenendo così un riscontro positivo;



  
Immagine che contiene testo, schermata, Carattere, numero

Descrizione generata automaticamente