

**RELAZIONE INTEGRAZIONE E TEST DI SISTEMI SOFTWARE**

**A.A. 2023/2024**

**STUDENTI:**

**Umberto Messina 675494 u.messina1** **@studenti.uniba.it**

**Omar Balde 759522 o.balde** **@studenti.uniba.it**

Indice

Indice.…………………………………………………………………………………………………………………………………………….2

Introduzione………………………………………………………………………………………………………………………………..…3

Perché fare testing………………………………………………………………………………………………………………3

Il nostro codice……………………………………………………………………………………………………………………3

Metodi della classe *RomanToDecimalConverter*………………………………………………………………….4

convertRomanToDecimal…………………………………………………………………………………………4

getRomanValue……………………………………………………………………………………………………….4

convertDecimalToRoman…………………………………………………………………………………………5

isValidRomanNumber………………………………………………………………………………………………5

areInterdipendent……………………………………………………………………………………………………6

isPalindrome……………………………………………………………………………………………………………6

Capitolo 1 – Homework 1

Specification based test – Approccio a 7 step

Implementazione dei test

Risultati dei test

Analisi Code Coverage iniziale

Soluzioni trovate

Analisi Code Coverage finale

Capitolo 2 – Homework 2

Property based test

Pianificazione dei casi di test

Implementazione dei casi di test

Statistiche raccolte

**INTRODUZIONE**

In questa relazione, presenteremo il processo di progettazione e sviluppo della nostra suite di test per la classe *RomanToDecimalConverter*. Esamineremo le metodologie utilizzate per creare i casi di test, gli strumenti impiegati per valutare la copertura del codice e l'efficacia complessiva della nostra suite di test nel garantire la correttezza e l'affidabilità del software sviluppato.

**PERCHE’ FARE TESTING**

La creazione di una suite di test efficace è un passo cruciale nello sviluppo del software, in quanto ci consente di verificare il corretto funzionamento delle funzionalità implementate e di individuare eventuali errori o comportamenti indesiderati. Nel contesto del nostro progetto, abbiamo sviluppato una suite di test per la classe *RomanToDecimalConverter*, che offre funzionalità per la conversione di numeri romani in numeri decimali e viceversa.

**IL NOSTRO CODICE**

Il nostro codice si occupa della conversione tra numeri romani e decimali (e viceversa). Consiste nell'interpretare correttamente una stringa che rappresenta un numero romano e convertirla in un numero decimale equivalente, o viceversa. Ad esempio, il numero romano "IV" deve essere convertito nel numero decimale 4 e il numero decimale 1994 deve essere convertito nel numero romano "MCMXCIV". Inoltre, abbiamo implementato 3 metodi che aggiungo ulteriori funzionalità, ovvero:

1. Verifica se una stringa è un numero romano valido (controlla se contiene solo caratteri validi (I, V, X, L, C, D, M).
2. Verifica se due numeri romani sono interdipendenti (controlla se il resto della loro divisione è 0)
3. Verifica se un numero romano è palindromo (leggibile da sinistra verso destra e da destra verso sinistra).

**METODI DELLA CLASSE *RomanToDecimalConverter***

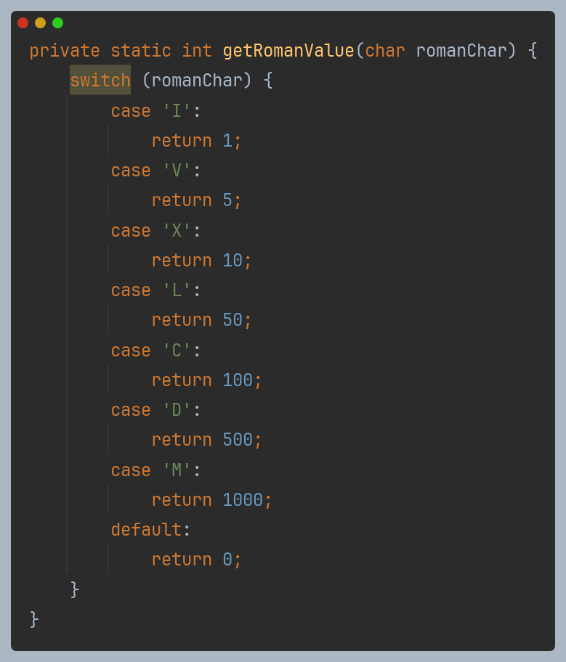
1. ***convertRomanToDecimal***

Immagine che contiene testo, elettronica, schermata, software

Descrizione generata automaticamente

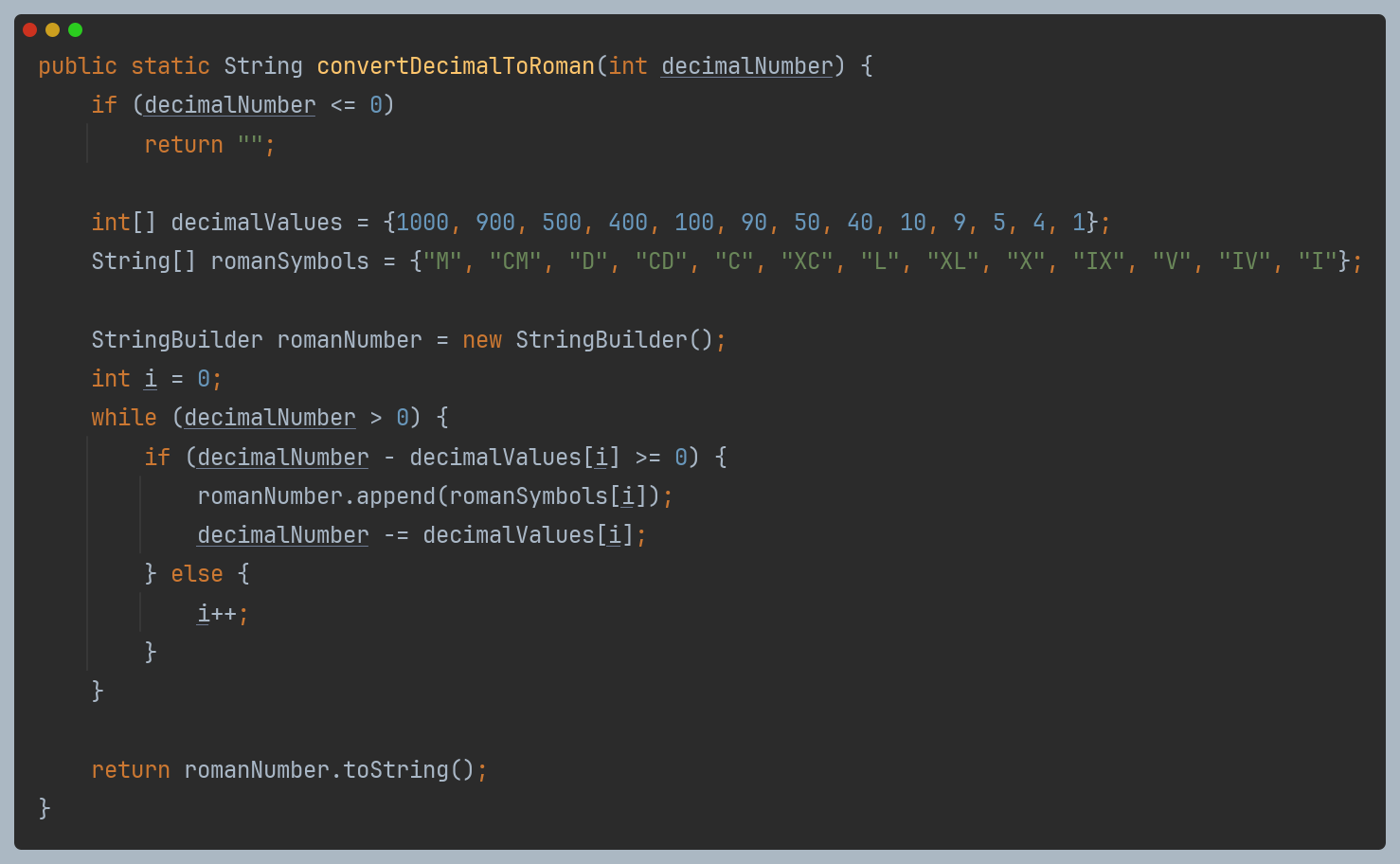
Il metodo *convertRomanToDecimal* prende in input una stringa *romanNumber*, che rappresenta un numero romano, e restituisce il suo valore decimale corrispondente. Inizia controllando se l'input è valido (non nullo e non vuoto). Successivamente, esegue un'iterazione sui caratteri della stringa dal più significativo al meno significativo. Durante l'iterazione, converte ciascun carattere romano in un valore decimale utilizzando il metodo *getRomanValue*, tenendo conto delle regole di somma e sottrazione. Infine, restituisce il valore decimale totale.

1. ***getRomanValue***

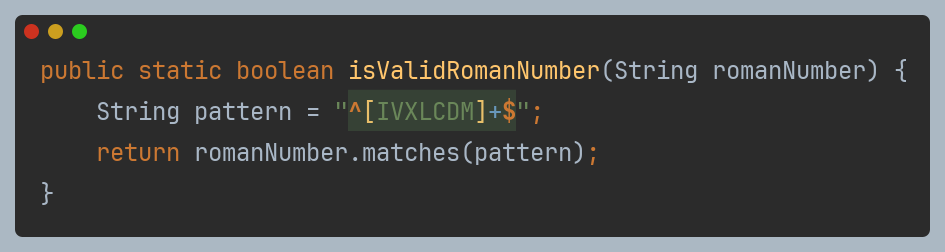
****

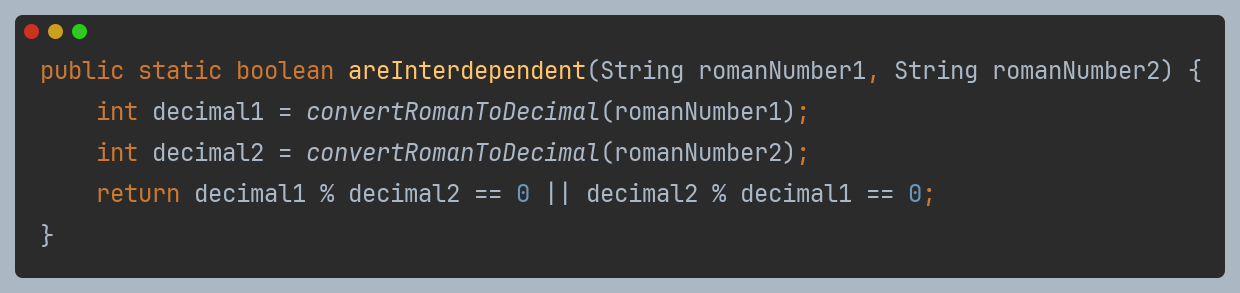
Il metodo *getRomanValue* accetta un carattere *romanChar* che rappresenta un simbolo romano e restituisce il suo valore decimale corrispondente. Utilizza un'istruzione switch per associare ciascun simbolo romano al suo valore decimale. Se il simbolo romano non è riconosciuto, restituisce 0.

1. ***convertDecimalToRoman***

****

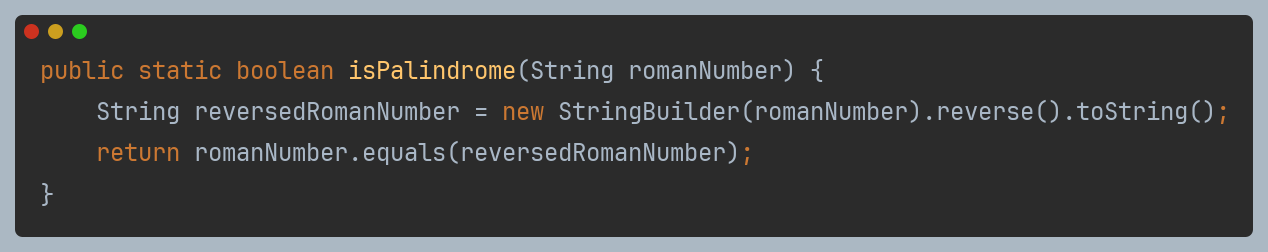
Il metodo *convertDecimalToRoman* converte un numero decimale in un numero romano. Inizia controllando se il numero decimale è inferiore o uguale a zero; in tal caso restituisce una stringa vuota. Successivamente, utilizza due array paralleli *decimalValues* e *romanSymbols* per associare i valori decimali ai simboli romani corrispondenti. Utilizza un ciclo while per iterare attraverso gli array, sottraendo il valore corrente dal numero decimale finché il numero decimale non diventa zero. Durante ogni iterazione, aggiunge il simbolo romano corrispondente al StringBuilder *romanNumber*. Infine, restituisce la rappresentazione romana del numero decimale convertito come una stringa.

1. ***isValidRomanNumber***  
     
   Il metodo *isValidRomanNumber* verifica se una stringa rappresenta un numero romano valido. Utilizza un'espressione regolare per definire il pattern di un numero romano valido, che include solo i simboli romani consentiti: I, V, X, L, C, D, e M. Il metodo, quindi, controlla se la stringa *romanNumber* corrisponde al pattern definito. Se la corrispondenza viene trovata, il metodo restituisce true, altrimenti restituisce false.
2. ***areInterdipedent***



Il metodo *areInterdependent* verifica se due numeri romani sono interdipendenti, ossia se uno dei due numeri è divisibile per l'altro. Per fare ciò, converte entrambi i numeri romani in numeri decimali utilizzando il metodo *convertRomanToDecimal*. Quindi, verifica se uno dei numeri decimali è divisibile per l'altro. Se uno dei due numeri decimali è divisibile per l'altro, il metodo restituisce true; altrimenti, restituisce false.

1. ***isPalindrome***



Il metodo *isPalindrome* verifica se un numero romano è un palindromo, cioè se può essere letto allo stesso modo sia da sinistra a destra che da destra a sinistra. Per fare ciò, crea una stringa *reversedRomanNumber* che rappresenta il numero romano invertito utilizzando la classe StringBuilder per invertire la stringa *romanNumber*. Quindi, controlla se *romanNumber* è uguale a *reversedRomanNumber*. Se la condizione è vera, il metodo restituisce true, altrimenti restituisce false.

**CAPITOLO 1 – HOMEWORK 1**

Per l’homework 1 ci è stato chiesto di sviluppare una suite di test black-box (Specification-Based Testing) con gli structural testing (white-box) e con l’analisi di copertura del codice (code coverage). Nello specifico:

* Per test black box, intendiamo una metodologia di verifica del software che si concentra sul comportamento esterno di un sistema senza considerare il suo funzionamento interno. In altre parole, il tester non ha bisogno di conoscere il codice sorgente o la struttura interna del software per eseguire questi test.
* Per test white box o test strutturale, intendiamo una metodologia di verifica del software che si concentra sulla struttura interna e sul codice sorgente del programma. A differenza del test black box, che si focalizza sul comportamento esterno, il test white box analizza come il codice è stato implementato per individuare potenziali errori.
* Per code coverage intendiamo la percentuale del codice sorgente di un programma che viene effettivamente eseguita durante l'esecuzione dei test. In parole semplici, indica quanto del codice viene testato attivamente.

Per la creazione della nostra suite di test abbiamo seguito l’approccio a 7 step:

1. **Analisi dei requisiti:** Il programma *RomanToDecimalConverter* deve convertire numeri romani validi in numeri decimali e viceversa. Deve anche controllare la validità dei numeri romani e verificare se due numeri romani sono interdipendenti (uno divisibile per l'altro) o palindromi (leggibili allo stesso modo da sinistra e da destra).
2. **Esplorazione del comportamento del programma:** Sono stati testati diversi input validi e invalidi per le funzioni *convertRomanToDecimal*, *convertDecimalToRoman*, *isValidRomanNumber*, *areInterdependent*, e *isPalindrome*. Si è verificato che il programma producesse i risultati attesi per gli input validi e gestisse correttamente gli input invalidi.
3. **Identificazione di partizioni e casi limite:**

*Partizioni:*

* + Input validi e invalidi per *convertRomanToDecimal* e *convertDecimalToRoman*.
  + Numeri romani validi e invalidi per *isValidRomanNumber*.
  + Coppie di numeri romani interdipendenti e non interdipendenti per *areInterdependent*.
  + Stringhe palindrome e non palindrome per *isPalindrome*.

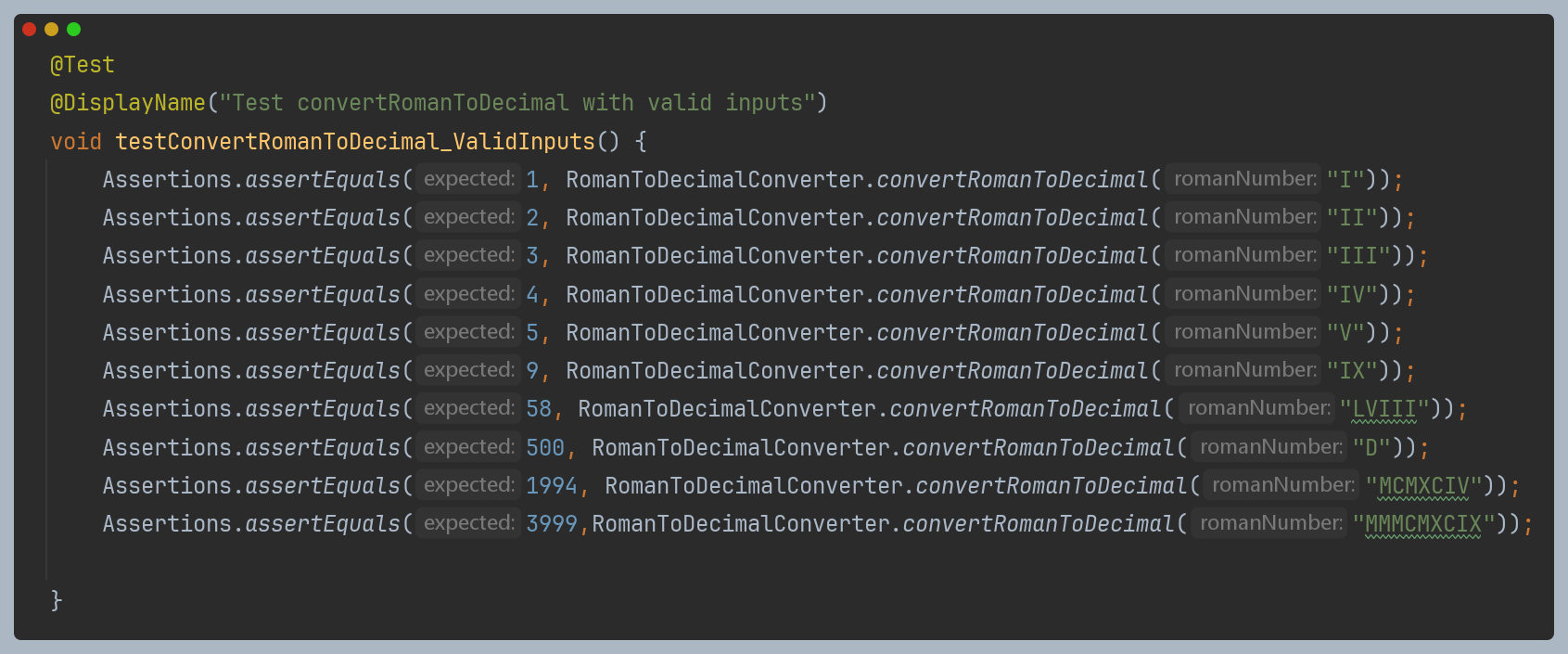
*Casi limite:*

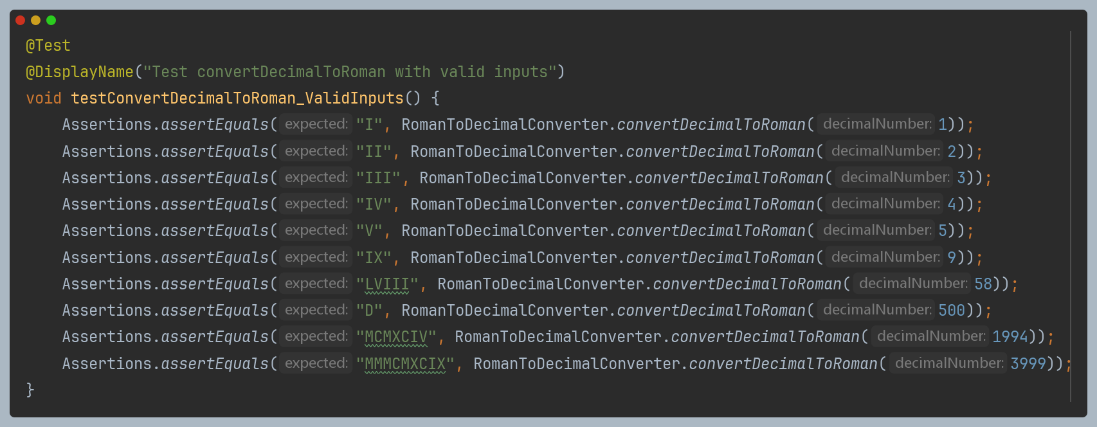
* + Input nullo o vuoto.
  + Negativi per *convertDecimalToRoman*.
  + Caratteri non validi in numeri romani.
  + Numeri romani con ripetizioni o sottrazioni.

1. **Identificazione dei casi di test:** Sono stati definiti casi di test specifici per coprire le partizioni e i casi limite identificati. I test verificano sia i comportamenti attesi che quelli non attesi (es. input invalidi che dovrebbero generare errori).
2. **Definizione dei casi di test:**
   * I casi di test sono stati scritti utilizzando il framework JUnit e le sue annotazioni (@Test, @DisplayName).
   * Ogni test case ha un nome descrittivo che indica lo scenario che sta testando.
   * I test utilizzano affermazioni di JUnit (Assertions) per verificare i risultati attesi (es. assertEquals, assertTrue).
3. **Automatizzazione dei casi di test:** I casi di test sono stati eseguiti automaticamente utilizzando un runner di test JUnit; questo consente di eseguire i test in modo rapido e affidabile, riducendo il rischio di errori umani.
4. **Miglioramento della suite di test con creatività ed esperienza:** La suite di test può essere ulteriormente migliorata includendo
   * Test per scenari di combinazione o edge case più complessi.
   * Mock o stub per isolare singole unità e testare la loro interazione.
   * Test di regressione per garantire che le modifiche al codice non introducano regressioni.

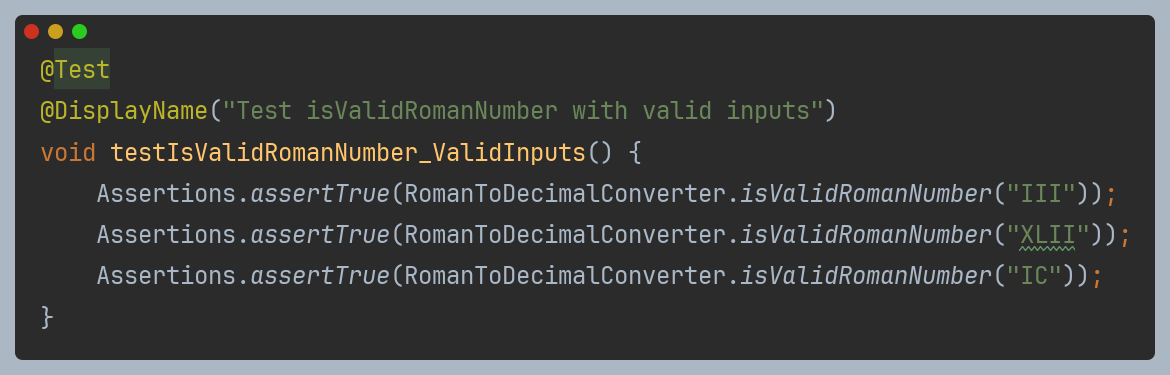
**IMPLEMENTAZIONE DEI CASI DI TEST**

Successivamente all’analisi dei 7 passi, abbiamo implementato la suite di test nel seguente modo:

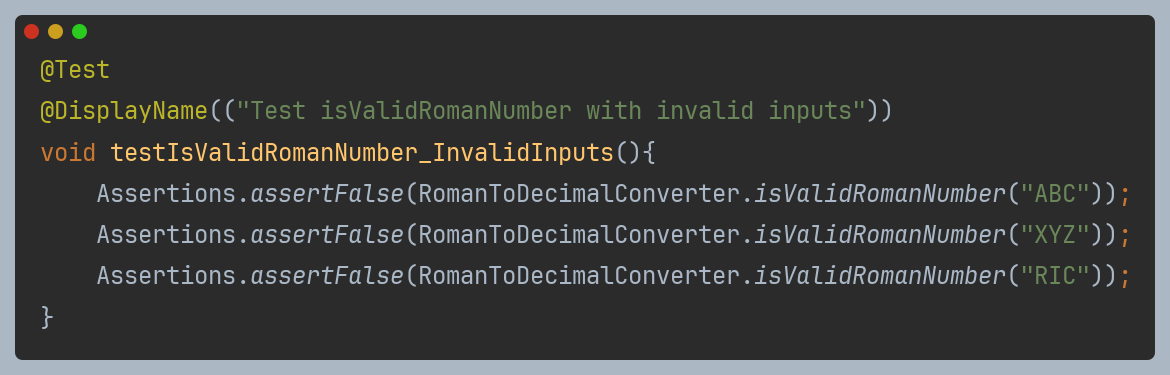
 ***testConvertRomanToDecimal\_ValidInputs***: Questo test verifica che il metodo *convertRomanToDecimal* converta correttamente i numeri romani validi in decimali. Vengono testati diversi numeri romani validi con i loro corrispondenti valori decimali.

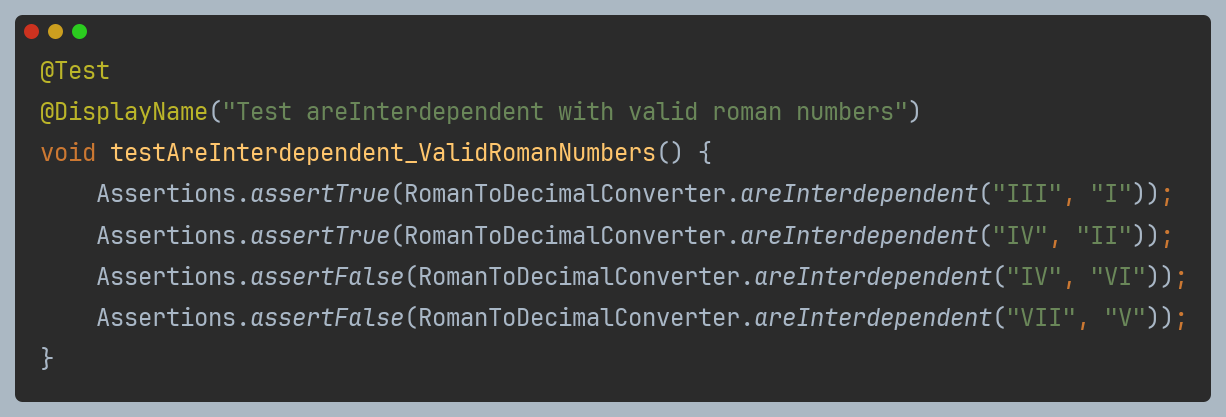


***testConvertDecimalToRoman\_ValidInputs***: Questo test verifica che il metodo *convertDecimalToRoman* converta correttamente i numeri decimali validi in numeri romani. Vengono testati diversi numeri decimali validi con i loro corrispondenti numeri romani.

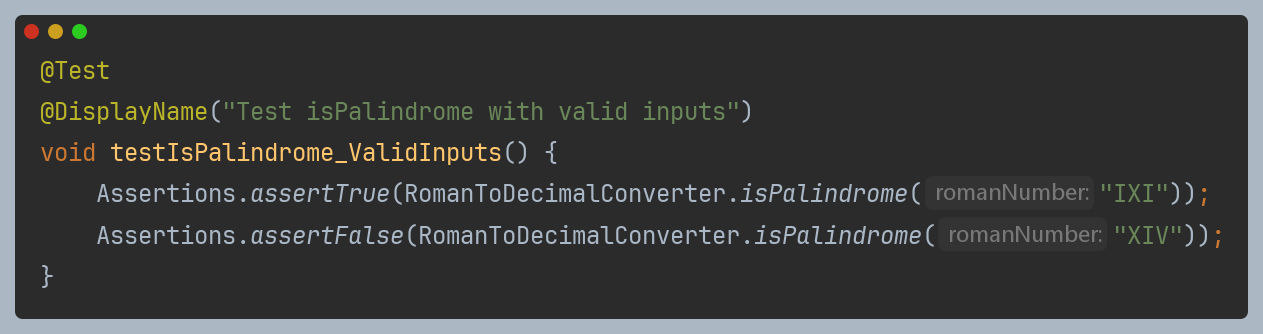


***testIsValidRomanNumber\_ValidInputs***: Questo test verifica che il metodo isValidRomanNumber restituisca true per numeri romani validi. Vengono testati diversi numeri romani validi.

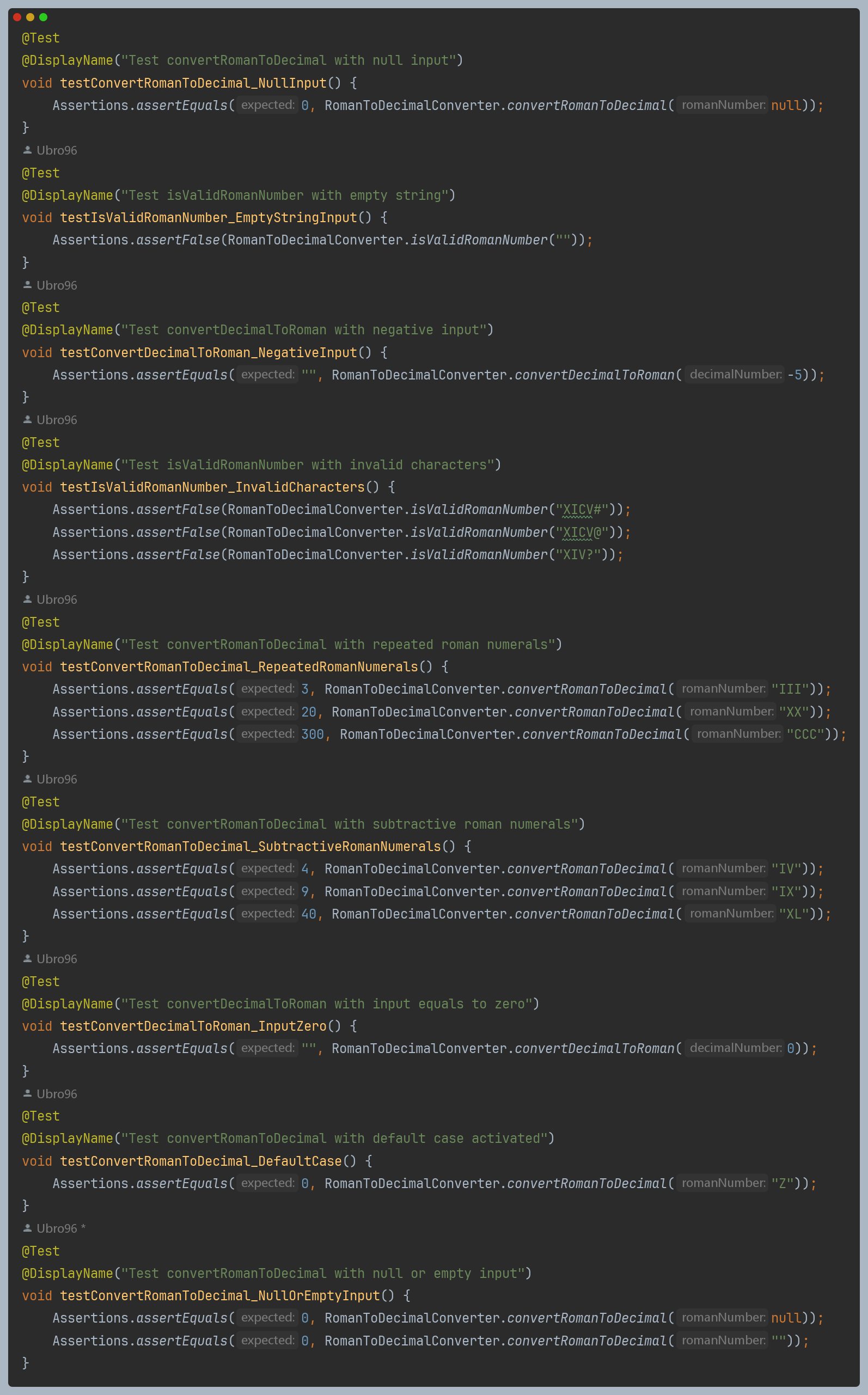
 ***testIsValidRomanNumber\_InvalidInputs***: Questo test verifica che il metodo *isValidRomanNumber* restituisca false per numeri romani non validi. Vengono testati diversi numeri romani non validi.



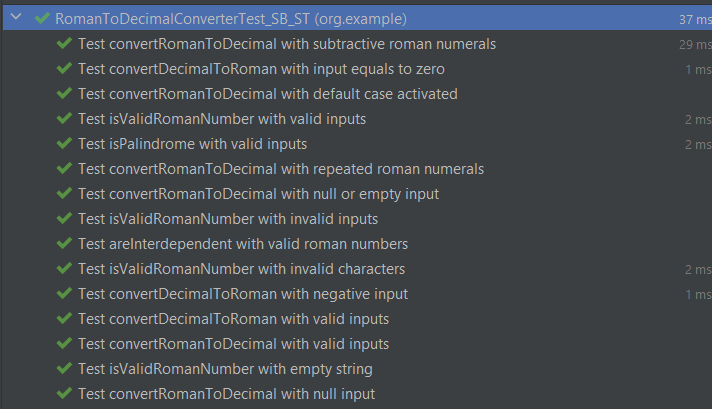
***testAreInterdependent\_ValidRomanNumbers***: Questo test verifica se il metodo *areInterdependent* identifica correttamente se due numeri romani sono interdipendenti o meno. Vengono testati diversi casi in cui i numeri romani possono essere interdipendenti o meno.



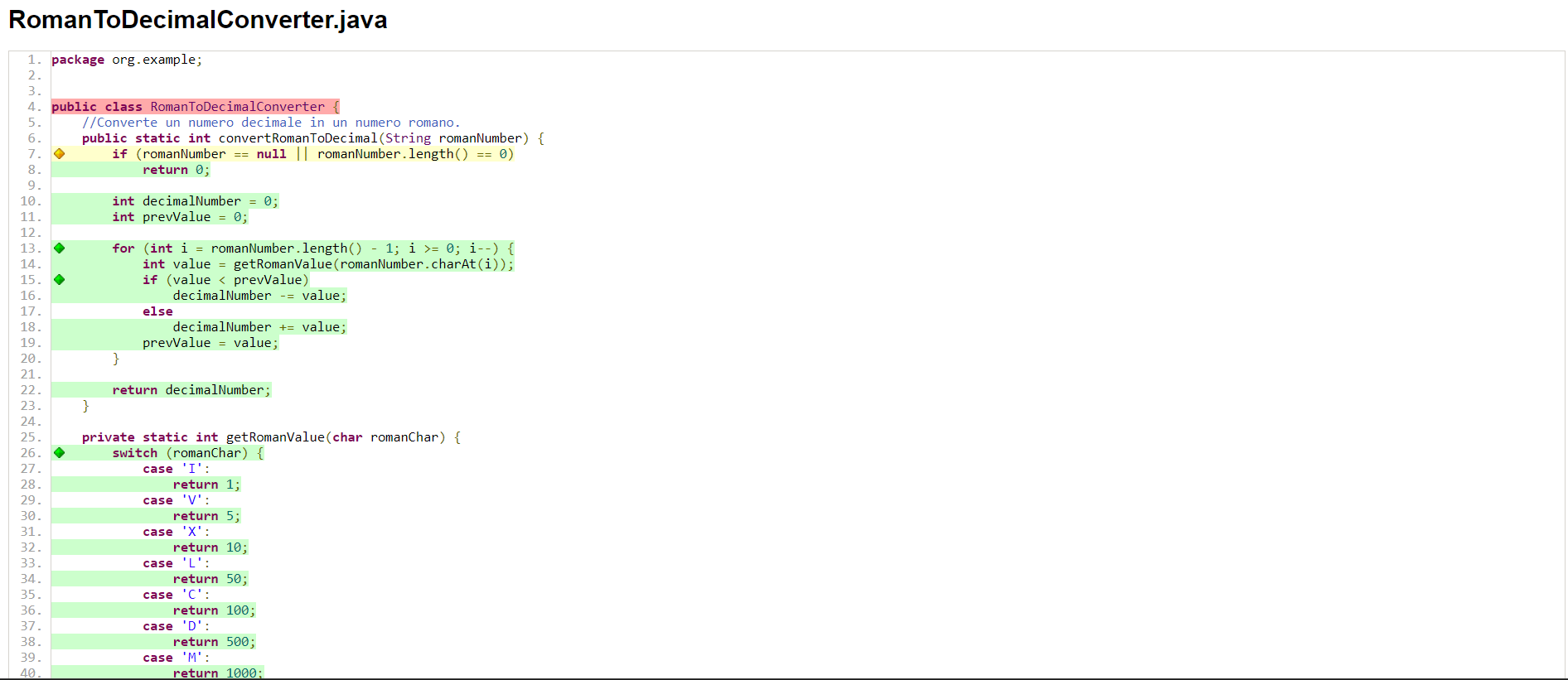
***testIsPalindrome\_ValidInputs***: Questo test verifica se il metodo *isPalindrome* identifica correttamente se un numero romano è un palindromo o meno. Vengono testati diversi numeri romani che sono palindromi e non palindromi.



**Altri test:** Gli altri test includono casi speciali come input nulli, input vuoti, input negativi e casi in cui i numeri romani contengono caratteri non validi o caratteri ripetuti. Questi test verificano il comportamento del software in situazioni anomale o limite.

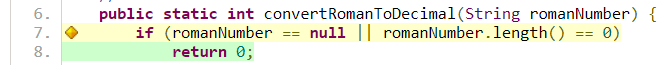
**RISULTATI DEI TEST**  


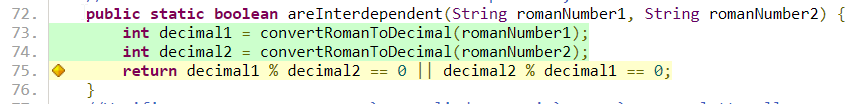
**ANALISI CODE COVERAGE INIZIALE**





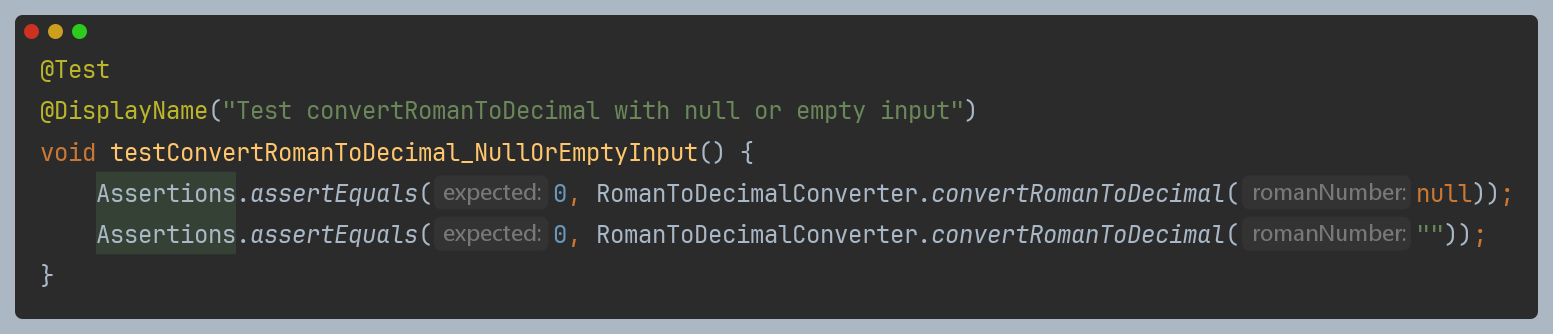
come possiamo vedere dagli screen la code coverage è quasi totalmente verde tranne che per due linee



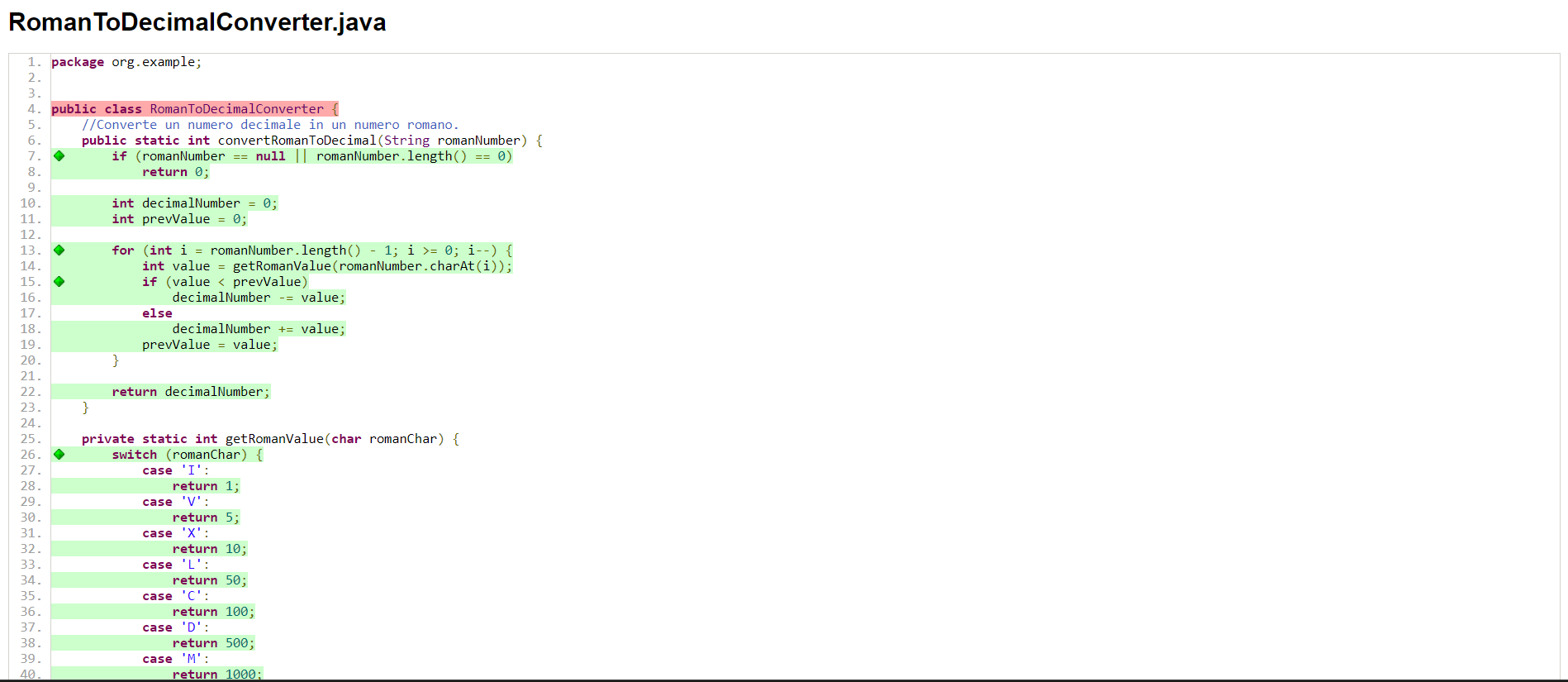


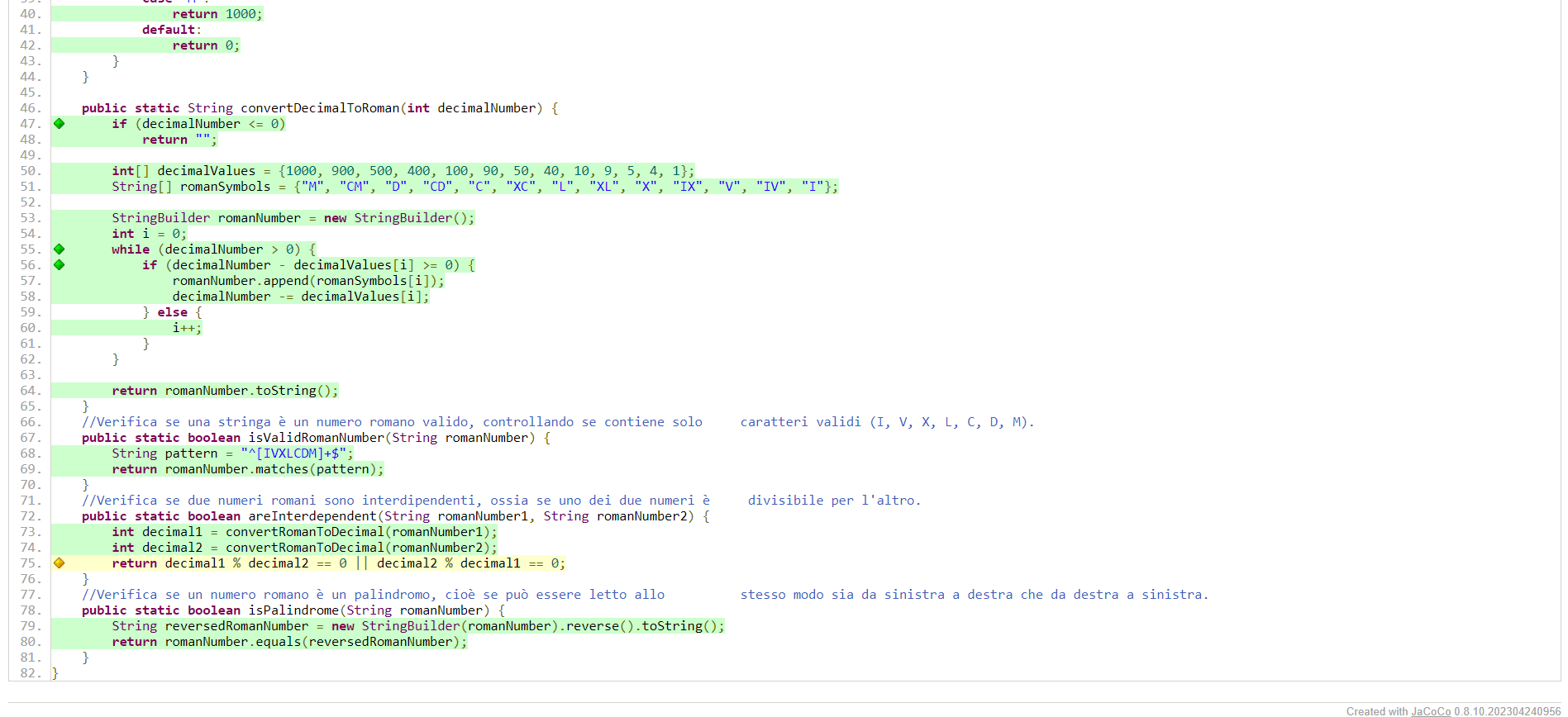
**SOLUZIONI TROVATE**

L’unica soluzione che abbiamo implementato copre soltanto la riga di codice numero 7; la riga 75 non può essere coperta totalmente dai test, perché i casi che mancano nella nostra suite sono quelli che gestiscono le divisioni con 0 o null e, a meno di refactor del codice sorgente, non è possibile implementarli nella classe di test.

**ANALISI CODE COVERAGE FINALE**

Dopo aver implementato la soluzione, la code coverage si presenta in questo modo





**CAPITOLO 2 – HOMEWORK 2**

**PROPERTY-BASED TESTING**

Il Property-Based Testing (PBT), o testing basato sulle proprietà, è una tecnica di testing che si basa sulla definizione di proprietà che il software dovrebbe soddisfare piuttosto che su singoli casi di test specifici.

Per svolgere il secondo homework assegnato, abbiamo deciso di utilizzare lo stesso codice dell’homework 1. Nonostante non fosse necessario utilizzare lo stesso codice, abbiamo deciso di riutilizzarlo in quanto già approvato dalla docente e familiare a noi.

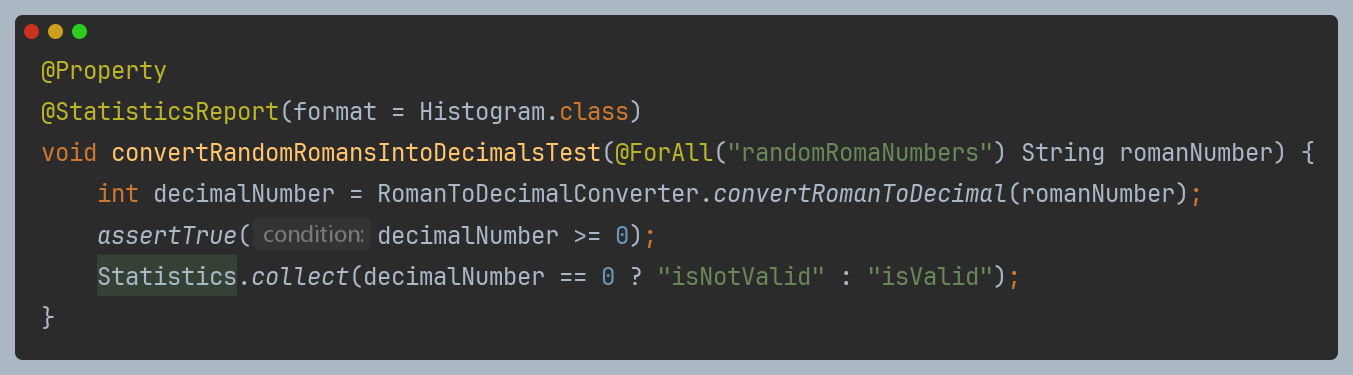
Avendo utilizzato lo stesso codice dell’homework 1; potete trovare l’analisi del suo funzionamento nel capitolo 1.

**PIANIFICAZIONE DEI CASI DI TEST**

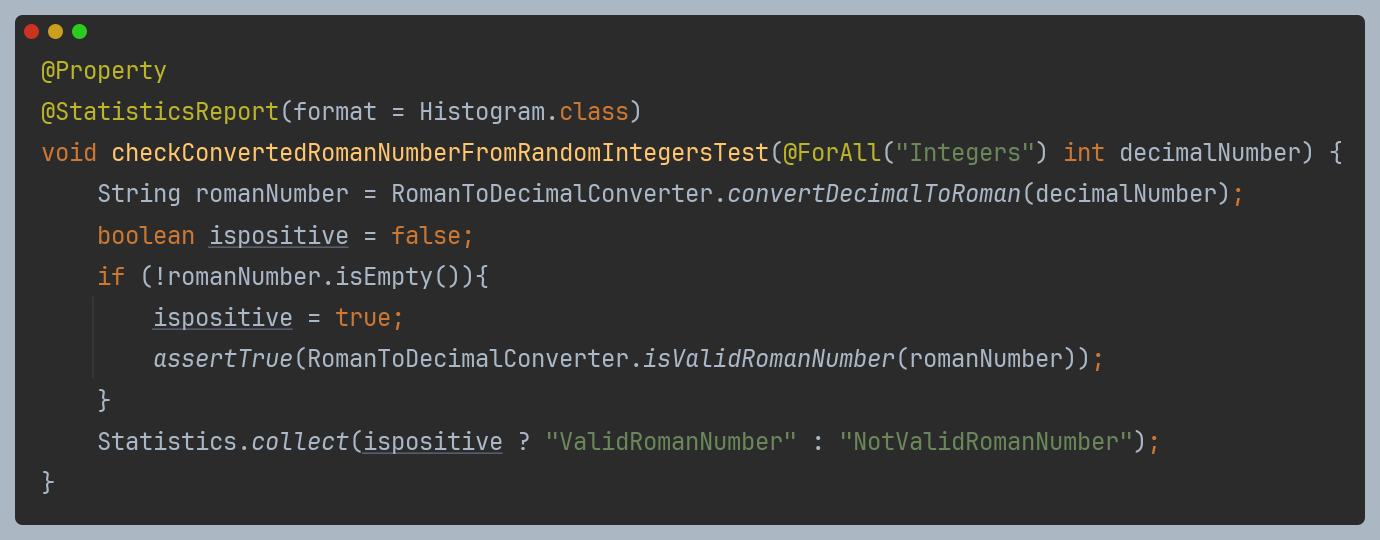
Dopo un’analisi del codice abbiamo individuato i seguenti punti chiave sui quali sviluppare la nostra suite di test:

1. **Copertura completa delle funzionalità**: I metodi scelti per la suite di test devono coprire tutte le principali funzionalità offerte dalla classe *RomanToDecimalConverter*. Questo include la conversione da numeri romani a decimali (*convertRomanToDecimal*), la conversione da decimali a numeri romani (*convertDecimalToRoman*), la validazione dei numeri romani (*isValidRomanNumber*), la verifica dell'interdipendenza tra due numeri romani (*areInterdependent*), e la verifica se un numero romano è un palindromo (*isPalindrome*).
2. **Gestione di casi limite**: I metodi devono includere test specifici per casi limite, come input nulli, stringhe vuote, numeri negativi, numeri romani non validi e altri casi che potrebbero portare a comportamenti inaspettati o eccezioni. Questo assicura che il software funzioni correttamente in una vasta gamma di situazioni.
3. **Test con dati casuali e generati**: Utilizzando il Property-Based Testing, dobbiamo includere test che utilizzano dati casuali o generati. Questo approccio aiuta a esplorare ampiamente lo spazio degli input possibili e a identificare comportamenti inaspettati o problemi che potrebbero non essere stati considerati nei test tradizionali.
4. **Raccolta di statistiche**: Oltre alla verifica del comportamento del software, la suite di test deve raccogliere statistiche sull'esecuzione dei test stessi. Questo fornisce informazioni utili sulle prestazioni e sulle metriche dei test, consentendo una valutazione più completa della qualità del software.

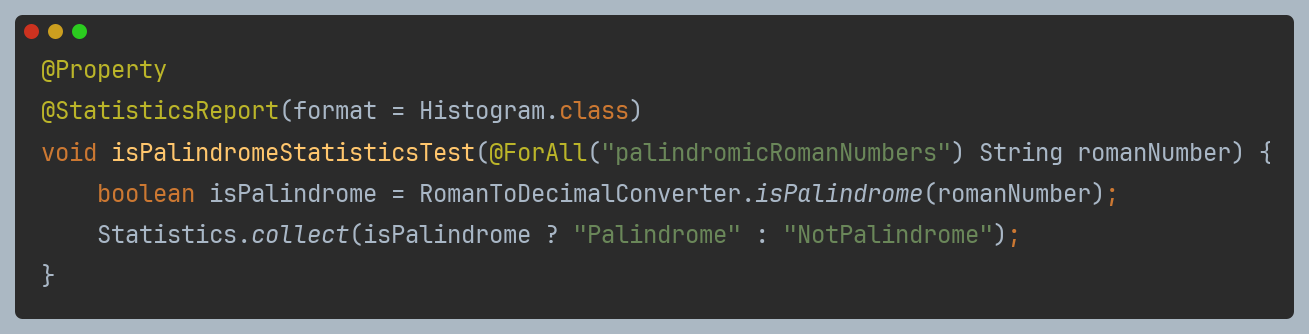
**IMPLEMENTAZIONE DEI CASI DI TEST**



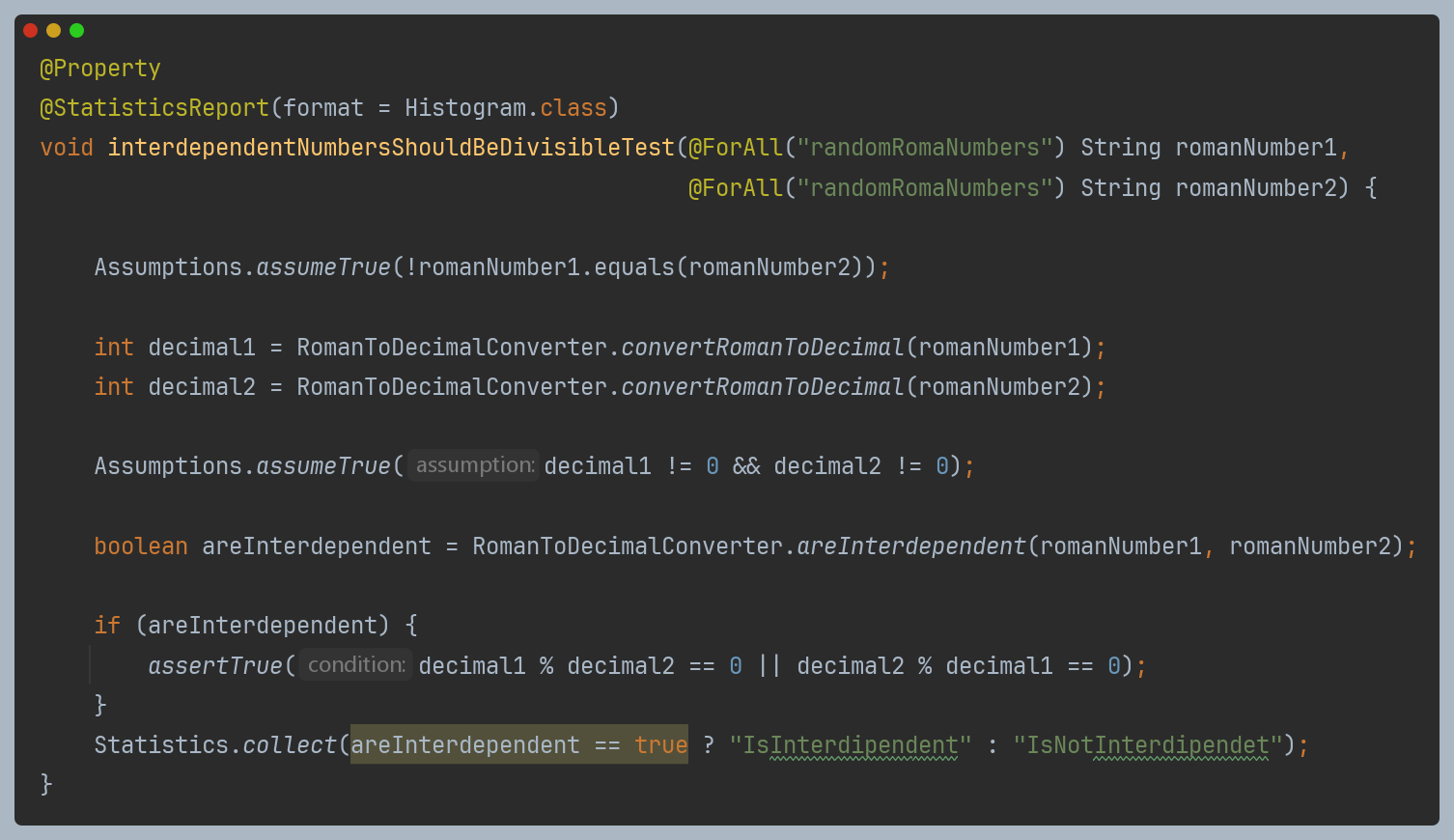
1. ***convertRandomRomansIntoDecimalsTest***: Questo test genera casualmente numeri romani utilizzando il generatore randomRomaNumbers. Per ogni numero romano generato, viene verificato che la conversione in decimale restituisca un numero maggiore o uguale a zero. Inoltre, il test raccoglie statistiche sul numero di numeri romani validi che vengono convertiti in decimali.



1. ***checkConvertedRomanNumberFromRandomIntegersTest***: Questo test genera casualmente numeri decimali utilizzando il generatore Integers. Per ogni numero decimale generato, viene verificato se la conversione in un numero romano è valida. Se la conversione è valida, viene raccolta una statistica. Questo test è progettato per testare la correttezza della conversione da decimale a romano per un ampio spettro di valori decimali.



1. **isPalindromeStatisticsTest**: Questo test genera casualmente numeri romani utilizzando il generatore `palindromicRomanNumbers`. Per ogni numero romano generato, viene verificato se è un palindromo. Viene quindi raccolta una statistica su quanti numeri romani sono palindromi e quanti non lo sono.



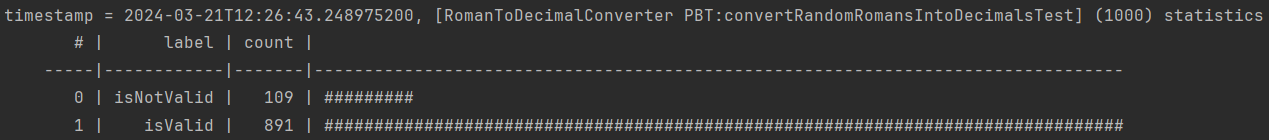
1. interdependentNumbersShouldBeDivisibleTest: Questo test genera casualmente coppie di numeri romani validi utilizzando il generatore `validRomanNumbers`. Per ogni coppia di numeri romani generati, viene verificato se sono interdipendenti utilizzando il metodo `areInterdependent`. Se sono interdipendenti, viene verificato che uno sia divisibile per l'altro. Viene quindi raccolta una statistica sulle coppie di numeri romani che sono interdipendenti e divisibili.



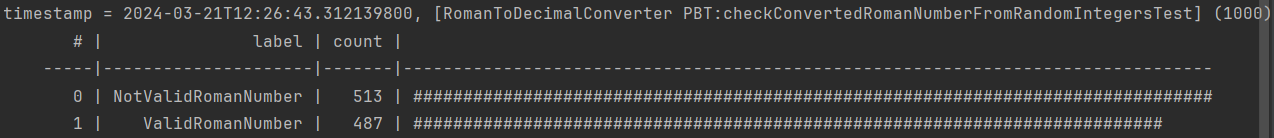
Abbiamo anche definito un generatore chiamato randomRomaNumbers che genera stringhe rappresentanti numeri romani validi. Questo generatore viene quindi utilizzato nel test *converted\_decimal\_should\_be\_greater\_than\_zero* per fornire valori validi di numeri romani. Ogni provide genera i valori appropriati per ogni test altrimenti il framework di jqwik non riesce a generare i valori richiesti per i test.

**STATISTICHE RACCOLTE**

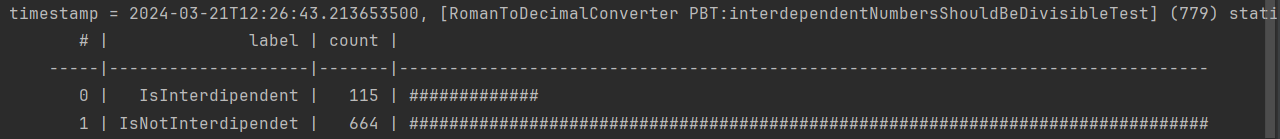
Di seguito riportiamo le statistiche raccolte dai metodi di test, nonostante il codice da noi testato non si presti alla raccolta di statistiche significative.



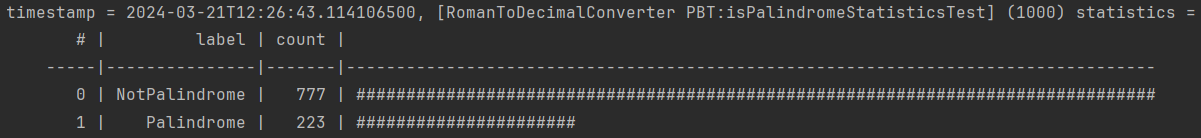
1. Stiamo tenendo traccia di quanti numeri romani validi che sono stati convertiti correttamente in decimali.



1. Stiamo tenendo traccia di quanti numeri romani abbiamo convertito con successo partendo da un numero decimale negativo.



1. Stiamo tenendo traccia di quanti numeri romani sono tra loro interdipendenti, ovvero sono tra loro divisibili.



1. Stiamo tenendo traccia di quali numeri romani sono palindromi.