Laurea Magistrale in informatica
Università di Salerno

Report

Team member
Saverio Napolitano - 0522501400
Gerardo Festa - 0522501452
Alessandra Parziale - 0522501422

https://github.com/GerardoFesta/infozilla

Sommario

1. Descrizione del tool

InfoZilla è una libreria e un tool che si occupa dell'estrazione di dati strutturati da fonti di dati non strutturati come e-mail, discussioni e segnalazioni di bug.

InfoZilla è in grado di identificare i seguenti elementi:

- **Java Source Code Regions:**
  Codice sorgente Java, di piccole o medie dimensione, che è utilizzato per illustrare un problema, descriverne il contesto di programmazione in cui si è verificato o rappresentare un possibile esempio di soluzione.
- **Patches:**
  Le patch rappresentano una piccola porzione di software progettata per aggiornare problemi in un programma. Il formato impiegato per le patch è il diff unificato (utilizzato per illustrare le modifiche apportate ad un file rispetto alla sua versione precedente).
- **Enumerations and Itemizations:**
  Vengono utilizzate per elencare elementi o presentare una sequenza di azioni che lo sviluppatore deve intraprendere al fine di risolvere un problema. Esse danno una struttura alla descrizione di un problema e agevolano la comprensione.
- **Java Stacktraces:**
  Elencano i frame dello stack attivi nella pila delle chiamate durante l'esecuzione di un programma. Vengono utilizzati per assistere la correzione di errori fornendo delle indicazioni sull'origine di un problema.
- **Talkback Traces (opzione non abilitata):**
  Dettagliano i contesti di crash e le informazioni sull'ambiente quando viene rilevato un problema.

1.1. Utilizzo del tool

Il comando **infozilla** è utilizzato per elaborare una serie di file specificati con l'opzione di estrarre le varie informazioni (**Java Source Code Regions, Patches, Enumerations and Itemizations, Java Stacktraces**), come riportato in **[Figura 1]**. L'utente può personalizzare ogni opzione per scegliere quali tipi di informazioni verranno elaborate ed estratte.

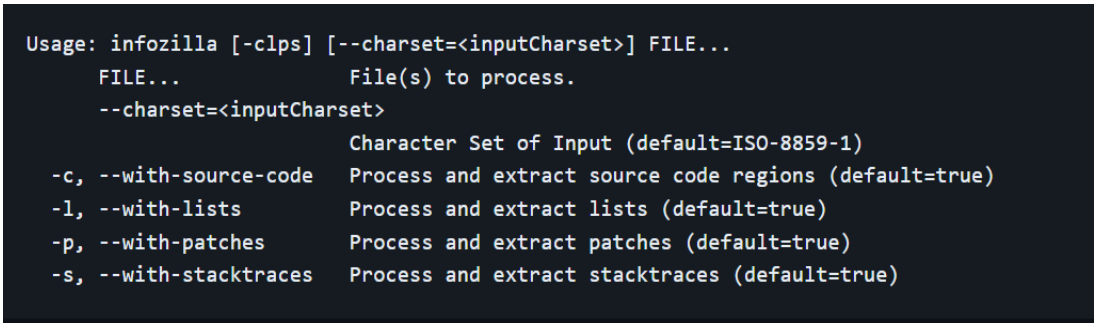

Figura 1: Opzioni per scegliere tipi di info da estrarre

1.2. Output

Come risultato finale verranno prodotti due file:
<nomefile>.txt.cleaned: contiene il testo in linguaggio naturale con tutti gli elementi che non sono stati identificati dai vari filtri.
<nomefile>.txt.result.xml: contiene tutti gli elementi strutturati in xml trovati nell'input.

Infine, la console di output visualizzerà, come riportato in **[Figura 2]**, il numero di tutti gli elementi trovati.

```
gradle run --args="demo/demo0001.txt"

Task :run
Extracted Structural Elements from demo/demo0001.txt
0        Patches
2        Stack Traces
4        Source Code Fragments
1        Enumerations
Writing Cleaned Output
Writing XML Output
```

Figura 2: Console di output

2. Comprensione del codice

In questa fase abbiamo eseguito una comprensione del codice sorgente ed è emerso il ruolo principale, visibile nel Sequence Diagram riportato in **[Figura 3],** dei packages ***cli***, ***helpers, filters, resources, bugreports e datasources.***

- Il package **cli**, contiene solo la classe Main che interagisce con il package ***filters***, presenta delle annotazioni che sono utilizzate per scegliere di abilitare o disabilitare l'opzione di processare ed estrarre i vari filtri. Questa classe rappresenta dunque quella che riceve l'input da linea di comando.
- Il package ***filters*** contiene quattro filtri utilizzati (***FilterPatches***, ***FilterStackTraceJAVA***, ***FilterSourceCodeJAVA***, ***FilterEnumeration***) ed uno non utilizzato (***FilterTalkBack***). Questo package si occupa di filtrare il testo dato in input e rilevare i vari elementi.
- Il package ***helpers*****,** precisamente la classe ***DataExportUtility*****,** viene utilizzato dal package ***cli*** al fine di creare elementi xml che rappresentano i vari filtri.
  Inoltre i vari filtri utilizzano il package ***helpers*****,** precisamente la classe ***RegExHelper***, per fornire alcune utility per lavorare con le espressioni regolari.
- Il package ***resources*****,** che contiene due file di testo**,** è utilizzato dal package **filters**, precisamente da ***FilterSourceCodeJAVA***, per individuare parti del codice java che corrispondono a dei modelli specifici (come import, package, singlecomment ecc.) attraverso l'ausilio di espressioni regolari.
- Il package ***elements*** viene utilizzato dai package ***helpers*** e ***filters.*** Nel primo caso, viene utilizzato dalla classe ***DataExportUtility*** come ausilio alla scrittura in output dei file xml, mentre nel secondo serve a creare le istanze degli elementi di cui il tool fa l'estrazione. Infatti, nel package *elements* sono

presenti tanti sottopackage quanti sono gli elementi di cui viene effettuato il parsing, con classi che servono solo a rappresentare questi elementi (bean).

- Il package ***datasources*** utilizza il package ***databaseConnector*** per stabilire una connessione al DataBase BugZilla. Attraverso l'utilizzo del package ***bugreports*** è possibile creare, leggere e aggiornare le istanze di bugreports che contengono informazioni sul bug come: lo stato, le discussioni o la risoluzione.
  Tuttavia questi due packages non sono stati considerati poiché non utilizzati.

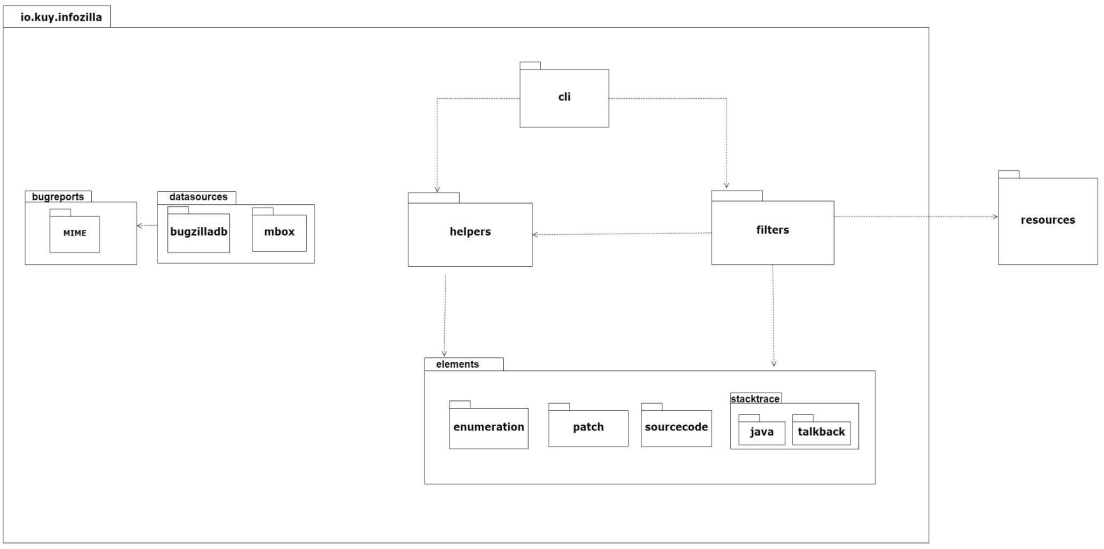

Figura 3: ***Package Diagram***

2.1. Interazione tra Main e FilterChainEclipse

La classe *Main,*come visibile nel Sequence Diagram (sezionato) riportato in **Figura 3,** chiama la classe ***FilterChainEclipse*** passando i parametri (data, withPatches, WithStachTraces, withStackTraces, withCode, withLists). Il parametro "data" è il contenuto del file letto, invece i parametri "withPatches", "withStackTraces", " withCode", "withLists" sono tutti parametri booleani specificati dell'utente e consentono di abilitare o disabilitare l'utilizzo dei vari filtri.

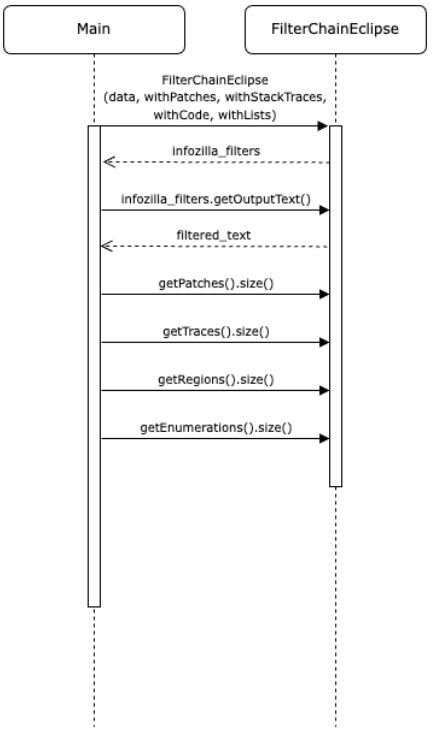

Figura 4: ***Sequence Diagram Main***

2.2. FilterChainEclipse

La classe ***FilterChainEclipse*****,** chiama i quattro filtri ***FilterPatches****,* ***FilterStackTraceJAVA****,* ***FilterSourceCodeJAVA****,* ***FilterEnumeration***, come visibile nel Sequence Diagram (sezionato) riportato in **[Figura 5]**.

Ogni filtro, chiamato attraverso il metodo ***runFilter*** passando i vari **OutputText**, restituisce una lista di elementi individuati (ad esempio, il metodo *runFilter* di *FilterPatches* restituirà una lista di oggetti istanze della classe *Patch*).
Il *Main* successivamente richiamerà, per ogni filtro, ***getOutputText*** per poter stampare il numero di elementi.
Il metodo ***getOutputText*** di ogni filtro utilizza il metodo ***doDelete*** di ***TextRemover*** per eliminare le porzioni di testo, marcate precedentemente, corrispondenti alle sezioni in cui sono stati trovati gli elementi di interesse del filtro. Il metodo restituisce dunque il testo dato in input al filtro, eliminando però gli elementi individuati.

Come visibile nel sequence diagram, l'outputText di un filtro diventa il testo dato in input al filtro successivo.

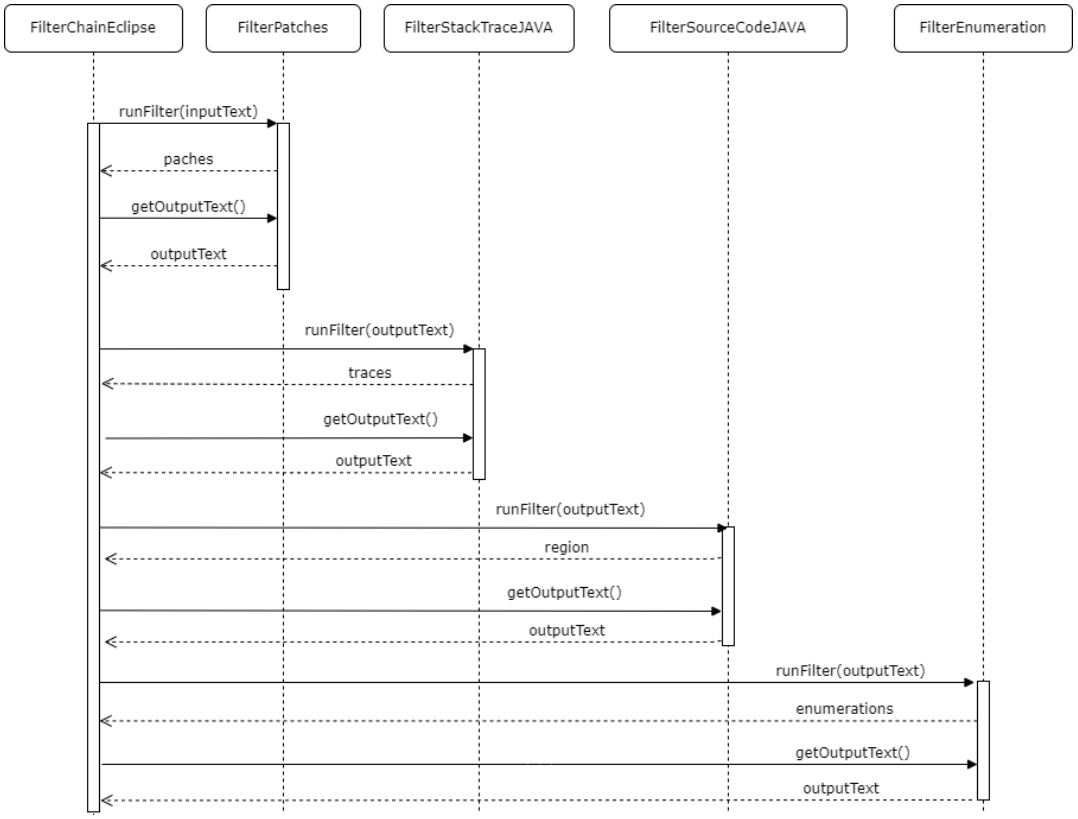

Figura 5: ***Sequence Diagram FilterChainEclipse***

2.3. Approccio generale dei Filter

Tutti i filtri implementano l'interfaccia ***IFilter*** la quale include due metodi, uno per l'esecuzione del filtro (***runFilter***) e l'altro per la generazione dell'output (***getOutputText***).
I vari ***runFilter*** invocati da ***FilterChainEclipse*** chiamano rispettivamente una ***getOutputText*** differente per ogni filtro. Ogni ***getOutputText*** inizializza una ***FilterTextRemover*** che setta un array booleano di dimensione pari al numero di caratteri del testo ricevuto in input con tutti False.
La porzione di testo individuata dai vari filtri viene annotata dal metodo ***MarkForDeletion*** ed eliminata attraverso il ***doDelete*** di ***FilterTextRemover***.

2.3.1. FilterPatches

Il primo filtro ad essere chiamato è ***FilterPatches*:**

Come visibile nel Sequence Diagram riportato in **[Figura 6],** il ***getPatches*** di ***FilterPatches*** richiama ***ParseForPatches*** che verrà utilizzato per individuare le varie patches, che vengono successivamente passate al metodo ***markForDeletion***, il quale imposterà a True le posizioni dell'array booleano corrispondenti alle sezioni di testo in cui le patch sono state individuate.
Inizialmente il metodo ***parseForPatches*** chiama ***partitionByIndex***.

Tale metodo chiama a sua volta ***findNextIndex***, passando il testo in input, per verificare la presenza di “Index:” con “====” e ritornare le aree individuate tra i vari “Index”.
Successivamente ***parseForPatches*** identifica le singole informazioni della patch: **"---"** (che rappresenta il file originale) e **“+++”** (che rappresenta il nuovo file) ed in seguito verifica se ci sono le informazioni di ***Hunk*** e, se sono in un formato valido, le aggiunge alla lista ***Hunk*** della specifica patch.

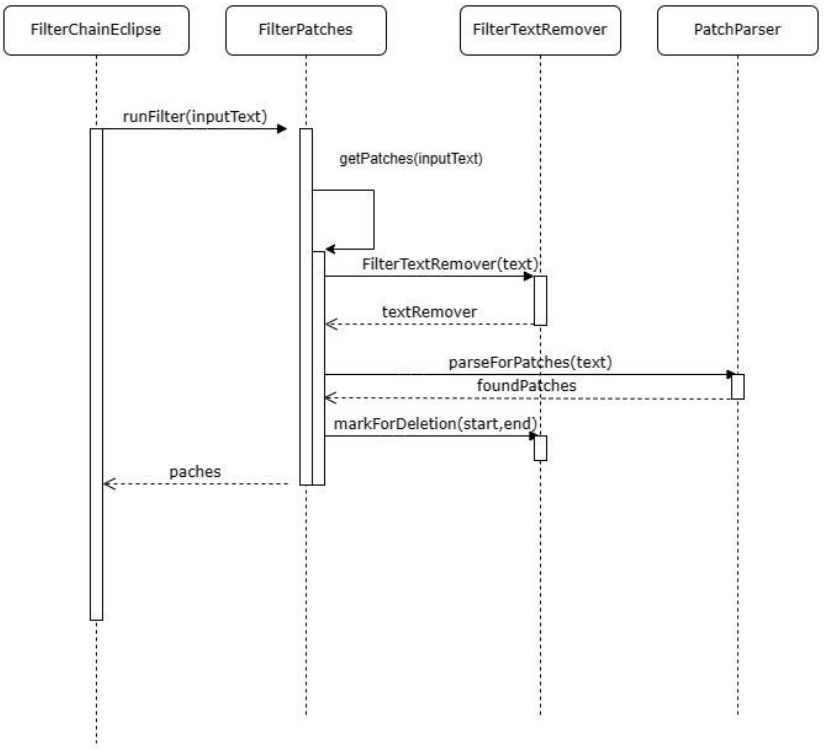

Figura 6: ***Sequence Diagram FilterPatches***

2.3.2. FilterStackTracesJAVA

Il secondo filtro ad essere chiamato è ***FilterStackTracesJAVA*****:**

Come visibile nel Sequence Diagram riportato in **[Figura 7], getStackTraces** chiama inizialmente il metodo ***findException*** il quale si occupa di individuare la lista di indici delle eccezioni attraverso il metodo ***matchResult*****.**
Quest’ultimo si occuperà di verificare se i caratteri dati in input coincidono con i pattern di eccezioni Java conosciute.
In seguito ***getStackTraces*** prende l’array di indici ed identifica e salva la regione, attraverso l’ausilio di ***findStackTraces*** verifica la tipologia di pattern (***StackTrace*** o ***Cause By***, clausola che sono causa dell’eccezione).
Se si tratta di un ***Caused by***, ***findStackTraces*** richiama ***createCause***; se si tratta di una Trace, richiama ***CreateTrace***.

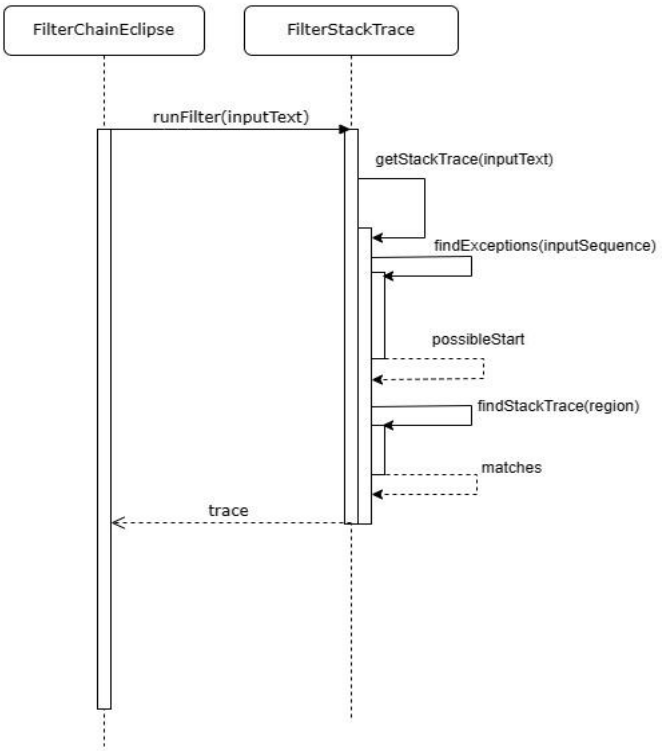

Figura 7: ***Sequence Diagram FilterStackTracesJAVA***

2.3.3. FilterSourceCodeJAVA

Il terzo filtro ad essere chiamato è ***FilterSourceCodeJAVA*:**

Come visibile nel Sequence Diagram riportato in **[Figura 8],** ***getCodeRegions*** scorre una lista di possibili pattern da un file dedicato, memorizza il ***codePatterns*** (che contiene la regex) e il ***CodeOption*** (che contiene la stringa “MATCH” della specifica keyword).
Successivamente se in ***CodeOption*** è presente la stringa “MATCH” calcola l’offset delle due parentesi graffe (apertura e chiusura) attraverso la funzione ***findMatches***. Dichiara e aggiunge una nuova ***CodeRegions***.
Infine ***getCodeRegions*** chiama la funzione ***makeMinimalSet*** che verifica se esistono elementi annidati e in tal caso li considera come un'unica regione.

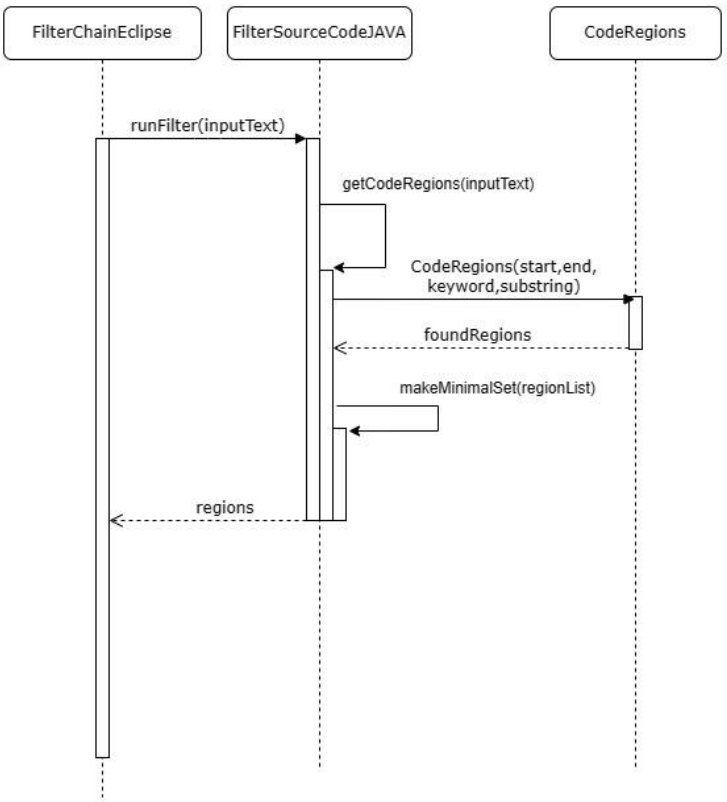

Figura 8: ***Sequence Diagram FilterSourceCodeJAVA***

2.3.4. FilterEnumeration

Il quarto filtro ad essere chiamato è ***FilterEnumeration*****:**

Come visibile nel Sequence Diagram riportato in **[Figura 9],** il ***getEnumerationAndItemization*** prende in input una stringa e dichiara tre liste (char, num, Itemization) per processare tutti i tipi di enumerazioni.
In seguito chiama ***getChar*****,** ***getEnum*** **e** ***getItemization*****.**

- ***getChar*****:** per identificare le enumerazioni che iniziano con le lettere (a. A. A ..)
- ***getEnum*****:** per identificare le enumerazioni che iniziano con i numeri (1. 2. 3. ..)
- ***getItemization*****:** per identificare le enumerazioni che iniziano con i trattini e spazi (- ..).

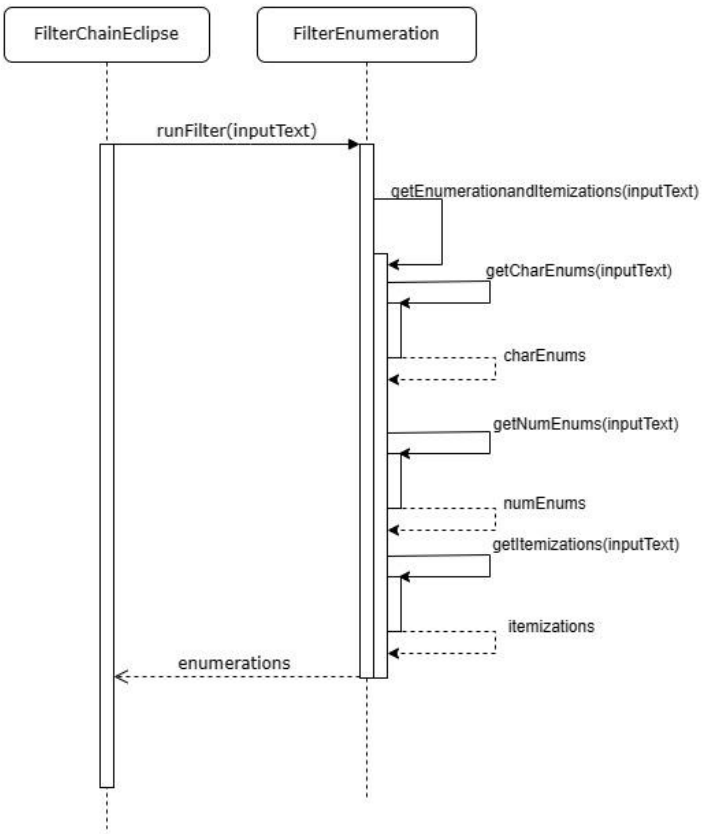

Figura 9: ***Sequence Diagram FilterEnumeration***