Assignment 02 - Asynchronous programming

Assignment svolto in gruppo da Federico Pirazzoli (matr. 0001079378), Gian Luca Nediani (matr. 0001075900)

Analisi del problema

L'assignment in questione richiede di risolvere lo stesso problema del precedente utilizzando diverse tecniche di programmazione asincrona, programmazione basata su task, programmazione con virtual thread, programmazione basata su eventi e programmazione reattiva. Avendo la possibilità di utilizzare dei framework con le relative astrazioni di alto livello, sarà importante realizzare soluzioni che sfruttano al massimo i concetti specifici di ciascun approccio.

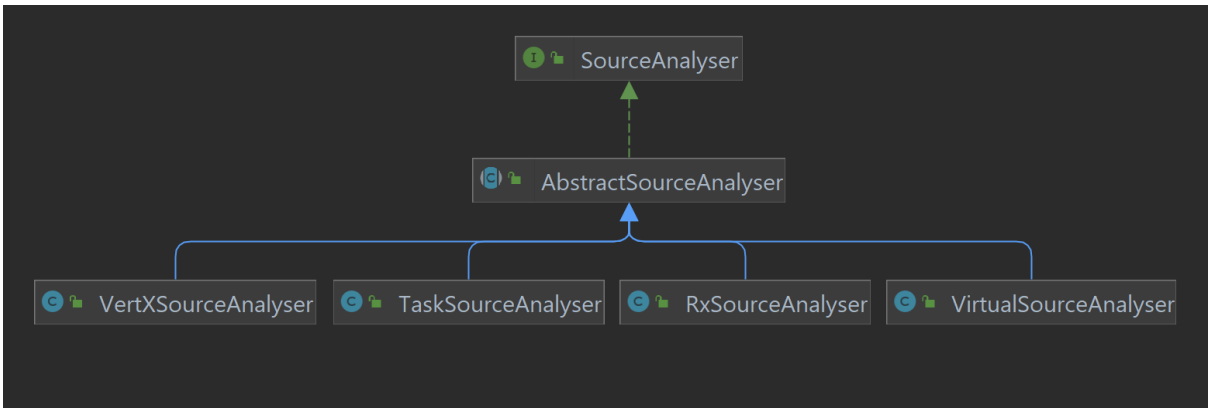
Strategia risolutiva

Per tutte e 4 le soluzioni si utilizza Java, nella versione 19 per poter utilizzare i virtual thread e con i seguenti framework/astrazioni:

- Soluzione basata su task -> Java Executor
- Soluzione basata su virtual thread -> Java Executor
- Soluzione basata su eventi -> Vert.x
- Soluzione reattiva -> RxJava

L'interfaccia ***SourceAnalyser*** con i due metodi per esplorazione di directory richiesta nell'assignment viene implementata da una classe astratta ***AbstractSourceAnalyser*** che implementa direttamente tutti i concetti comuni alle 4 soluzioni come la GUI, e svariati metodi di utility per le stampe e gli aggiornamenti dei risultati o il conteggio del numero di righe di un file. Questo viene fatto per semplificare l'architettura e limitare le ripetizioni di codice.
AbstractSourceAnalyser ha poi alcuni metodi astratti che verranno implementati secondo la logica di ciascuna soluzione: oltre ai due metodi dell'interfaccia, anche i metodi per avviare e fermare la GUI tramite i tasti "start" e "stop" sono astratti in quanto ogni soluzione avrà una differente logica per l'avvio e l'arresto delle computazioni asincrone.

Viene mostrato un diagramma UML di alto livello di quanto appena spiegato:



Soluzione basata su task (Executor Java)

Nella soluzione basata su task si utilizza un Executor Service con una pool di thread fissato al numero massimo di core disponibili, al *service* vengono passati i task *Runnable* che restituiscono dei future, utili per tenere traccia della terminazione delle operazioni. Il primo task è quello relativo alla directory in input, dopodiché viene svolta una ricerca ricorsiva delle sottodirectory dove ogni sottodirectory trovata genera un nuovo task di esplorazione mentre una volta trovato un file .java viene inviato al service un task di computazione che conteggia il numero di righe ed aggiorna le strutture dati dei file col maggior numero di righe e degli intervalli. Nella variante con GUI all'interno del task di computazione di un file, una volta terminato quest'ultimo, viene anche invocato l'aggiornamento della GUI.
Siccome non è possibile sapere a priori il numero di task da lanciare, viene simulato un comportamento bloccante aspettando la terminazione di tutti i future così che il programma non termini prima che tutti i task siano stati forniti all'executor e poi completati.

Soluzione basata su virtual thread (Executor Java)

L'architettura della soluzione basata su virtual thread è ad alto livello quasi del tutto analoga a quella basata su task in quanto in entrambi i casi si fa uso degli Executor Java. La differenza sta nel fatto che nella soluzione task based (TaskSourceAnalyser) i task messi in coda sono gestiti da un Executor Service che dispone di una pool fissa di thread (pari al numero massimo di processori della macchina che esegue il programma), mentre nella soluzione con virtual thread l'Executor Service istanzia un nuovo virtual thread per ogni nuovo task che riceve. Questo è possibile grazie all'alta efficienza dei virtual thread Java che hanno un costo molto basso in termini di memoria e una gestione ottimizzata sui "platform thread".

Soluzione basata su eventi (Vert.x)

Nella soluzione basata su eventi, si sfruttano le capacità e costrutti della libreria Vert.x, attraverso l'istanziazione di un'entità *Vert.x*, che detiene diversi event loops, da notare però che nonostante non ci sia priorità nella scelta dell'handler che si occupa di gestire le chiamate delle operazioni all'event loop, non verranno mai eseguite concorrentemente, e generalmente (tranne per alcuni casi, come i worker verticles) verrà sempre utilizzato lo stesso event loop; nel nostro caso avremo l'istanziazione di un oggetto *FileSystem* per

effettuare operazioni su file più facilmente (come la *readDir()* per leggere i contenuti di una cartella, o *fileRead()* per leggere file), e diverse chiamate a *Promise* che effettuano operazioni: o di esplorazione della/e cartella/e, o di lettura di file .java trovati, i quali verranno tutti accodati (come detto prima) ad uno stesso event loop che si occuperà di effettuare queste operazioni, che restituiscono dei *Future* immessi in una *Queue* per tener traccia della terminazione delle operazioni. Nella variante GUI le operazioni sono sempre le stesse, ed è possibile fermare l'esecuzione dell'esplorazione tramite l'utilizzo del tasto "stop"

Soluzione reattiva (RxJava)

Nella soluzione reattiva si sfruttano i costrutti di RxJava per realizzare un flow a partire dalla directory iniziale che grazie a una serie di operatori riesce ad effettuare la ricerca ricorsiva delle sottodirectory e la computazione sui file java. Grazie agli scheduler *IO* e *computation* di RxJava tali operazioni possono essere svolte in multi-threading asincrono, nello scheduler *io* vengono "sottoscritti" gli operatori di ricerca ricorsiva mentre in quello di *computation* le operazioni svolte sui file .java trovati. Nella variante con GUI si fa uso dell'operatore *doOnNext()* per poter avere il side effect di aggiornamento della GUI, e dell'operatore *takeUntil()* per attendere una eventuale interruzione tramite il tasto "stop" della GUI.

Performance

Le performance delle 4 soluzioni vengono testate sulla stessa directory dell'assignment precedente al fine di poter fare un confronto diretto di performance rispetto alla soluzione multi-threaded con monitor da noi implementata. Tutti i tempi di completamento del compito di analisi della directory di seguito riportati sono una media di 10 esecuzioni su due diversi processori utilizzando gli stessi parametri.

1. Multi-threading del primo assignment con un solo thread
2. Multi-threading del primo assignment con numero di thread pari al numero massimo di core
3. Executor Service task based con pool di thread pari al numero di core
4. Executor Service con virtual thread
5. Vert.x con event-loop
6. RxJava con scheduler IO e scheduler computation per la gestione dei thread

Tempo medio completamento:	1. Multi-threading 1 core/ thread	2. Multi-threading Tutti i core, 1 thread per core	3. Executor, tutti i thread, 1 thread per core	4. Executor, 1 virtual thread per task	5. Vert.x	6. RxJava
Intel i5-4400 (4 core)	3135.1 ms	2510.7 ms	430.2 ms	479.6 ms	6305.4 ms	526.2 ms

AMD FX-8320E (8 core)	2481.6 ms	1175.8 ms	435 ms	477.8 ms	7122.2 ms	651.6 ms

Avvio del programma

Per lanciare una demo delle soluzioni sviluppate, effettuare la build Gradle e poi run dell'applicazione. A questo punto verrà richiesto di specificare tramite input su riga di comando quale delle quattro varianti e quale dei due metodi (riga di comando one-shot o GUI interattiva) si vuole testare.

Di seguito un'immagine dimostrativa dell'interfaccia grafica: