RELAZIONE PCD ASSIGNMENT #02 – LORENZO CAMPANELLI

Assignment svolto singolarmente da Lorenzo Campanelli (matr. N. 0001047818)

Analisi del problema

L'assignment richiede di analizzare ricorsivamente, a partire da una directory radice, dei file sorgenti contandone il numero di righe di codice e di tenere traccia della distribuzione del numero di linee di codice nei file analizzati nonché di un numero N di file col maggior numero di righe di codice.

E' richiesto di implementare strategie risolutive che sfruttino rispettivamente:

1) Approccio asincrono a task
2) Approccio basato su Java Virtual Threads
3) Approccio asincrono ad eventi (event-loop)
4) Approccio basato su programmazione reattiva

Approcci risolutivi

Per tutte e 4 le strategie risolutive si è scelto di adottare Java nella versione 20 abilitando il flag "—enable-preview" che consente l'utilizzo dei Java Virtual Threads.

Nello specifico si è scelto di adottare le seguenti tecnologie (framework):

1) Approccio a Task: Java Executor Framework
2) Approccio con Virtual Threads: Java Virtual Threads
3) Approccio asincrono ad eventi: Eclipse Vert.x
4) Approccio basato su programmazione reattiva: ReactiveX nella versione RxJava

Come richiesto sono state definite (ed implementate) delle interfacce SourceAnalyserXX per i vari approcci che definiscono i metodi getReport per ottenere un report con le informazioni richieste e analyseSources che consente la produzione incrementale dei risultati nonché l'interruzione dell'analisi.

Al fine di ottenere una produzione incrementale dei risultati è stata introdotta l'interfaccia AnalysisUpdateListener (nel package *common*) con i seguenti metodi, che viene implementata dalle View dei vari approcci:

```
void statsUpdated(AnalysisStatsSnapshot snapshot);
```

L'agente che si occupa dell'agornamento della View ad intervalli regolari (in base al periodo fissato) scatta una "snapshot" delle statistiche e poi invoca questo metodo per "notificare" l'aggiornamento alla View.

```
void analysisCompleted(boolean wasStopped, AnalysisReport report);
```

Questo metodo viene invocato per notificare a coloro che implementano l'interfaccia AnalysisUpdateListener (cioè in questo caso alle View) che l'analisi è terminata ed accetta in input il report dell'analisi eseguita (anche in caso di interruzione) e un flag booleano che indica se l'analisi era stata interrotta o era stata completata.

Suddivisione in package

Per ogni approccio risolutivo adottato è stato creato un package che raccoglie tutti i sorgenti ad esso riconducibili. Vale a dire:

1) Approccio a task: package ***executors***
2) Approccio con Virtual Threads: package **virtualthreads**
3) Approccio con event loop: package **eventloop**
4) Approccio con programmazione reattiva: package **reactive**

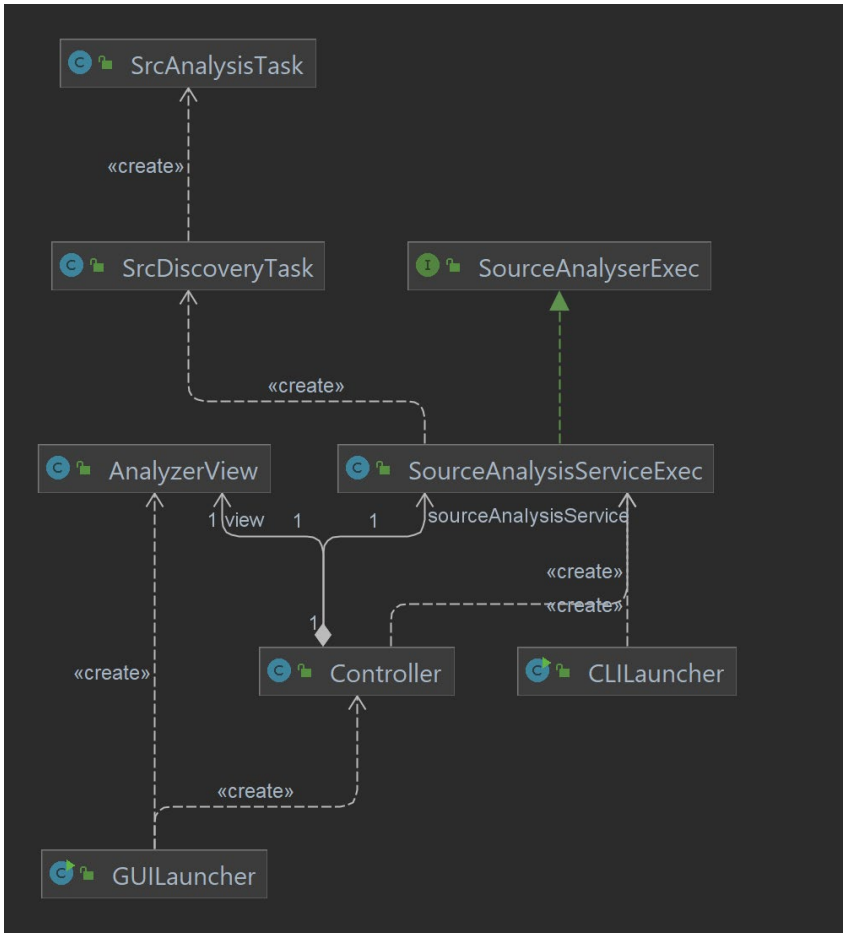
Gli elementi comuni a tutti gli approcci sono presenti nel package **common**.

Descrizione dettagliata dei vari approcci

Di seguito vengono descritti i vari approcci in maniera più dettagliata:

Approccio a task con Executors Framework

Di seguito viene mostrato un UML Class Diagram di massima contenente le classi/interfacce coinvolte:



Discussione dell'interfaccia e dell'implementazione

```
Future<AnalysisReport> getReport(Path rootDir, String[]
extensions, int maxSourcesToTrack, int nBands, int maxLoC);
```

Questo metodo restituisce una Future che una volta completata conterrà il report con le informazioni richieste. Sulla Future restituita il chiamante potrà mettersi (eventualmente) in attesa con una get() per ottenere il risultato richiesto.

```
void analyseSources(Path rootDir, String[] extensions, int
maxSourcesToTrack, int nBands, int maxLoC);
```

Questo metodo consente l'analisi incrementale della cartella selezionata ed è interrompibile.

Per quanto riguarda l'implementazione l'idea è stata quella di creare un "Service" che svolga le operazioni richieste. Questo Service internamente è implementato da un ExecutorService di tipo fixedThreadPool che esegue i task che gli vengono passati.

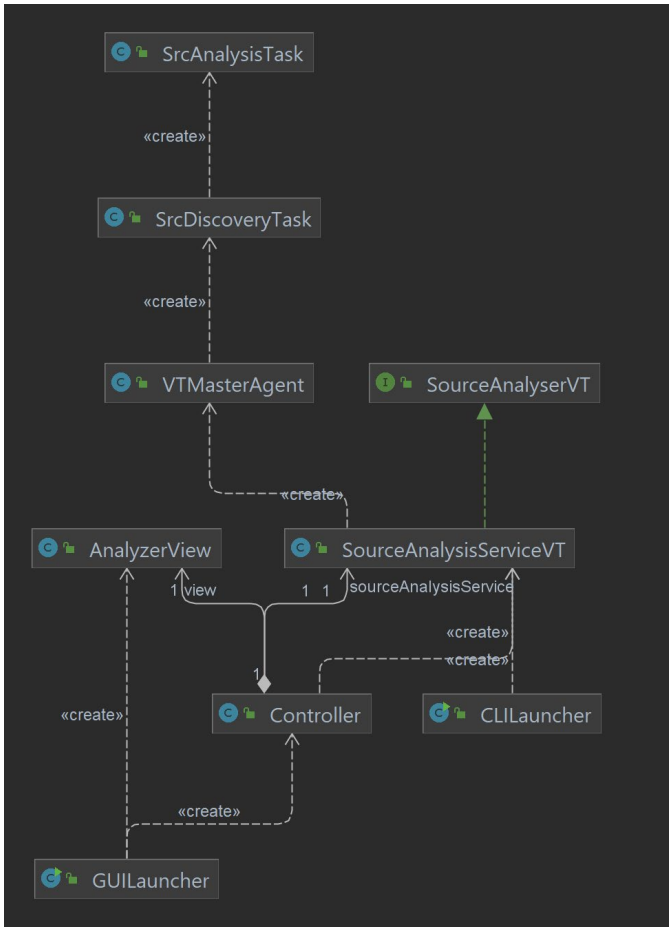
Esistono due tipi di task: il primo denominato SrcDiscoveryTask esegue la visita dell'albero delle cartelle a partire dalla directory radice, visitando ricorsivamente tutte le sottocartelle. Per ogni file sorgente incontrato accoda nell'executor un nuovo task di tipo SrcAnalysisTask che si occupa di contare il numero di linee di codice nel file, aggiornando di conseguenza le statistiche.

Poiché non è noto a priori il numero di file da analizzare, il SrcDiscoveryTask accoda la future restituitagli dal metodo submit dell'executor in una lista di future. Al termine della visita di tutto l'albero delle directory il SrcAnalysisTask si mette in attesa sulla lista di future facendo la get di tutte le future per poi completare la Future ritornata dal metodo col report richiesto dal chiamante.

Il metodo analyseSources permette invece l'analisi incrementale per mezzo di uno ScheduledExecutorService che esegue periodicamente il task *UpdateTask* che si occupa dell'aggiornamento della GUI.

Approccio con Virtual Threads

Di seguito viene mostrato un UML Class Diagram di massima delle classi/interfacce coinvolte:



Discussione interfaccia/implementazione

L'interfaccia SourceAnalyserVT presentata per l'approccio con Virtual Threads non presenta differenze rispetto a quella dell'approccio a Task con Executors.

Per quanto concerne l'implementazione (SourceAnalysisServiceVT) anche questo approccio prevede l'implementazione di un "Service" con funzionalità analoghe a quelle descritte per l'approccio a task.

La differenza rispetto all'approccio a task risiede nel fatto che, mentre nell'approccio a task veniva utilizzato un pool di thread di dimensione fissata; nell'uso dei virtual threads, viene sfruttata la natura più "logica" dei virtual threads per cui un solo carrier platform thread può "trasportare"/eseguire più virtual threads, che vengono opportunamente ed automaticamente "swappati" nel caso in cui eseguano chiamate bloccanti. In considerazione di questo viene creato un nuovo virtual thread per ogni task.

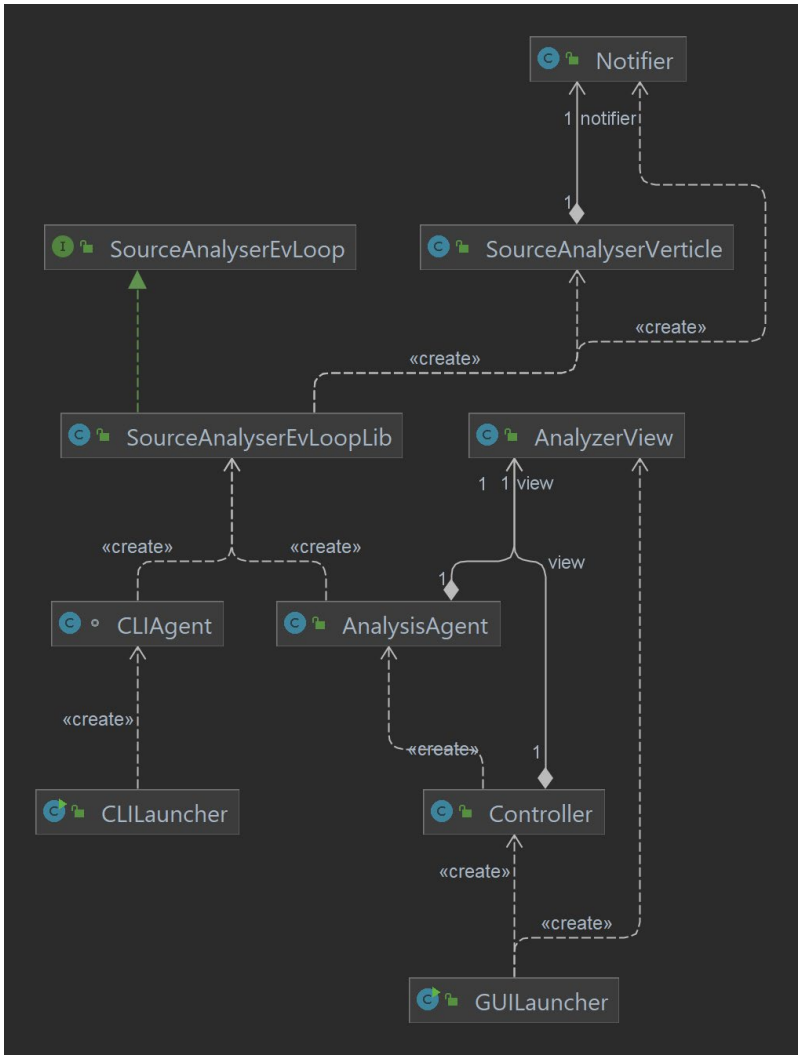
Tutti i virtual threads creati vengono poi aggiunti in una coda di threads. Su tale coda si mette in attesa un thread "fisico" VTMasterAgent, che dà avvio all'analisi creando ed

avviando il primo virtual thread di discovery della cartella radice e ponendosi successivamente in attesa (mediante join) di tutti i virtual threads presenti nella coda.

E' da notare anche il fatto che in questo caso non viene più eseguita una DFS come nell'approccio con Executor ma nella pratica una sorta di visita di tipo Breadth First dell'albero.

Approccio basato su Event Loop (Vert.x)

Di seguito viene mostrato di diagnamma UML di massima delle classi coinvolte:



Nella soluzione basata su Event Loop si sfruttano le potenzialità del framework Vert.x per poter lavorare in maniera asincrona con un event loop.

Vert.x mette in campo un un'implementazione del Reactor Pattern che prevede la creazione di un certo numero di thread che agiscono da event loop e che di conseguenza non dovranno mai bloccarsi.

Nel caso in questione quindi, è stato sfruttato l'oggetto *FileSystem* e le API non bloccanti da esso offerte per eseguire la lettura asincrona delle cartelle ed il metodo

executeBlocking per delegare le chiamate bloccanti ad un pool di worker esterno all'event loop.

Discussione interfaccia/implementazione

L'interfaccia espone i consueti metodi getReport e analyseSources con le seguenti signature:

```
Future<AnalysisReport> getReport(Path rootDir, String[]
extensions, int maxSourcesToTrack, int nBands, int maxLoC);
```

Questo metodo restituisce una Future (sulla quale il client potrà poi agganciare gli handler da eseguire in caso di completamento con successo o fallimento della promise ad essa associata.

```
Future<Void> analyseSources(Path rootDir, String[] extensions,
String address);
```

Il metodo analyseSources invece, restituisce una Future<Void> su cui il client potrà agganciare gli handler.

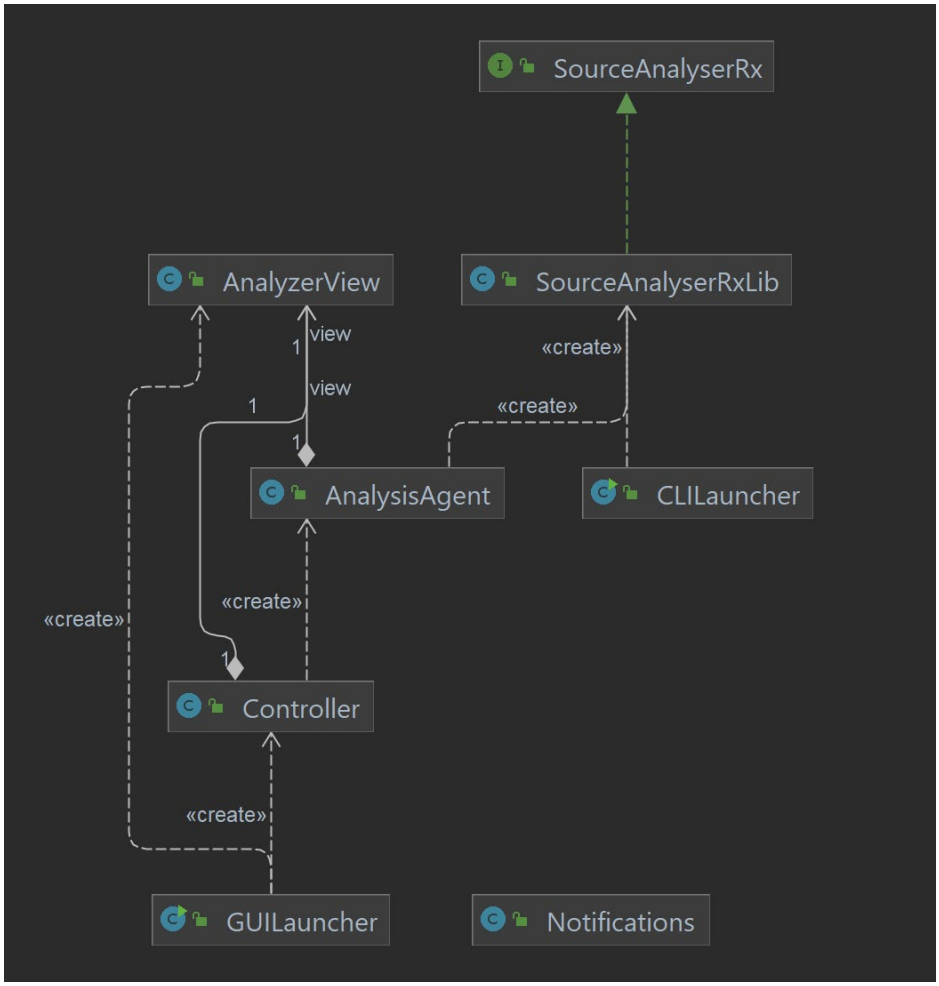
L'implementazione prevede il deploy di un Verticle denominato *SourceAnalyserVerticle* che si occupa di eseguire l'analisi. Mentre esegue l'analis,i per ogni nuova directory o file sorgente analizzato, pubblica un evento col topic indicato nel parametro address del metodo analyseSources, sull'Event Bus dell'istanza di Vert.x sotto forma di JSON.

Sarà poi compito del client della libreria (qui rappresentato dal Verticle AnalysisAgent) porsi in ascolto dei messaggi pubblicati sull'Event Bus, tenere traccia delle statistiche, ed aggiornare periodicamente la View.

Questa scelta, sebbene forse non ottimale da un punto di vista delle performance mi ha dato la possibilità di sperimentare l'Event Bus di Vert.x e, a mio modo di vedere, di creare un metodo analyseSources che si occupa della sola analisi interrompibile dei file sorgenti, lasciando al client della libreria l'onere e la libertà di scegliere cosa fare dei risultati prodotti incrementalmente.

Approccio basato su programmazione reattiva (RxJava)

Di seguito viene mostrato un UML Class DIagram di massima contenente le classi/interfacce coinvolte:



La soluzione con programmazione reattiva fa uso dei concetti propri di RxJava al fine di realizzare un flow che a partire dalla scansione della directory iniziale arriva ad esaminare tutti i file sorgenti pesenti all'interno dell'albero delle directory.

Per rendere lo svolgimento delle operazioni asincrono viene fatto uso degli *Schedulers* di RxJava: infatti, mediante l'uso del metodo subscribeOn(Schedulers.io()) gli observable sono stati fatti lavorare in thread appositamente creati dal framework per operazioni di I/O.

Discussione interfaccia/implementazione

L'interfaccia espone i metodi getReport e analyseSources con le seguenti signature:

```
Observable<AnalysisReport> getReport(Path rootDir, String[]
extensions, int maxSourcesToTrack, int nBands, int maxLoC);
```

Il metodo in questione ritorna un Observable<AnalysisReport> che può essere sottoscritto dal client in qualità di Observer per ottenere il report desiderato.

```
Observable<JsonObject> analyseSources(Path rootDir, String[]
extensions);
```

Per l’analisi incrementale, il metodo analyseSources nella versione reattiva, ritorna un Observable<JsonObject> con l’idea di fornire gradualmente, all’observer che si sottoscrive ad esso, i risultati sotto forma di stringhe JSON che possono essere poi elaborati a seconda delle esigenze (nel caso in questione per ottenere delle statistiche), lasciando alla libreria il solo compito di dover eseguire l’analisi interrompibile dei sorgenti con produzione incrementale dei risultati, senza legarsi allo specifico utilizzo che ne verrà fatto dei risultati prodotti dall’analisi.

Il fatto che sia stata utilizzata la classe JsonObject offerta da Vert.x per la creazione e manipolazione in maniera facile di oggetti JSON ha portato a definire il client della libreria (AnalysisAgent) come Verticle che si occupa di raccogliere le statistiche e di eseguire periodicamente l’aggiornamento della GUI.

Per quanto riguarda l’implementazione (*SourceAnalyserRxLib*) questa si avvale degli operatori RxJava ed in particolare,per la realizzazione del metodo analyseSources, è stato utilizzato l’operatore *takeUntil* per intercettare lo stop dell’analisi e chiudere il flow. Ciò però genera delle eccezioni durante le chiamate bloccanti a ai metodi di java.nio.Files list() per il listing delle directory e lines() per il conteggio delle righe di codice nei vari sorgenti che non si è riusciti perfettamente a gestire.

Valutazione delle performance

Tutti gli approcci sono stati testati su un clone della repo linux-master (analisi di sorgenti C) su un PC portatile dotato di CPU Intel Core i7-6700HQ (4 core, 8 threads), 16 GB di RAM ed SSD Crucial MX500 2 TB SATA.

I seguenti tempi sono stati calcolati per mezzo di test da 10 iterazioni ciascuno effettuati con la classe PerformanceTests presente nei sorgenti.

Approccio	**Executors**	**Virtual Threads**	**Event Loop (Vert.x)**	**ReactiveX (RxJava)**
Tempo medio	3278.7	3017.2	16751.8	10363.5 (CLI) ~ 4000 (GUI)

Per l’approccio basato su ReactiveX ho riportato anche un tempo stimato medio di esecuzione nella versione con GUI poiché pare offrire prestazioni migliori rispetto alla versione con CLI ma non ho compreso bene il perché.

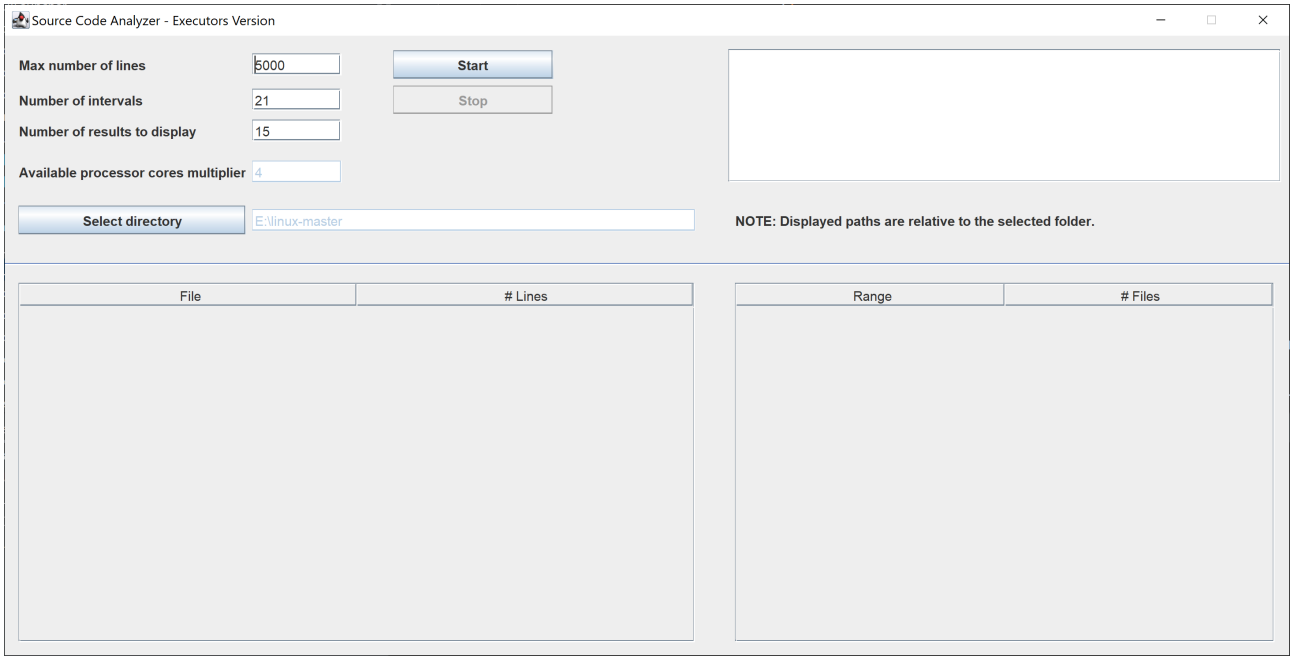
Avvio del programma

Per ogni tipo di approccio sono presenti due classi **CLILauncher** e **GUILauncher** che consentono di lanciare rispettivamente la versione a riga di comando (a cui bisogna passare in input il percorso della root directory da cui iniziare l’analisi, il numero massimo

di righe di codice per delimitare la distribuzione, il numero di intervalli della distribuzione e il numero di sorgenti con più righe di codice che vogliamo visualizzare) e la versione con interfaccia grafica.

Di seguito viene mostrata l'interfaccia grafica con cui è possibile interagire col programma.

E' possibile impostare i parametri richiesti (cambiando quelli di default) e selezionare una directory premendo il pulsante "Select directory".



Nota Bene: Ignorare il campo "Available processor cores multiplier" che avevo introdotto per debugging.