**Assignment-02**

**Strategia comune**

In tutti e quattro gli approcci si è fatto uso dell’implementazione di una struttura chiamata **Result**. Essa fornisce semplicemente la possibilità di aggiungere nuovi file analizzati in mutua esclusione e ad ogni aggiunta internamente posiziona il nuovo file **ordinandolo correttamente** rispetto a quelli presenti. Di conseguenza negli approcci seguenti la parallelizzazione è principalmente concentrata sulla ricerca dei file, la loro analisi e l’aggiunta al risultato temporaneo e finale. (In questo modo le tempistiche per l’ordinamento vengono ridotte notevolmente, avendo un risultato temporaneo costantemente ordinato).

**1) Approccio asincrono a Task – Framework Executors**

**Architettura e strategia risolutiva**

A livello architetturale in questo approccio si è pensato di realizzare la struttura di base implementando il pattern **MVC** per facilitare l’interazione del sistema con l’interfaccia grafica a linea di comando e con la GUI.

Raffinando il livello di dettaglio dell’architettura, si elencano gli elementi principali delle varie sezioni:

**Model:**

* DirectoryScanner
* ScanFolderTask
* CountLinesTask

Il Model contiene al suo interno il servizio offerto dal sistema, ovvero il **DirectoryScanner**, che funge da starting-point per l’esecuzione dei Task.

Per quanto riguarda la progettazione dei Task ne sono stati individuati due, lo **ScanFolderTask** si occupa dello scanning di una Folder, mentre il **CountLinesTask** del conteggio delle linee a partire da un file .java.

**Controller:**

* SourceAnalyzer

Il controller implementa l’interfaccia SourceAnalyzer che al suo interno contiene le definizioni di getReport() e analyzeSources().

**View:**

* ConsoleView
* GUIView

**Comportamento del sistema**

**Esecuzione dei Task**

L’esecuzione dei task sopra descritti è stata organizzata utilizzando una strategia ricorsiva, implementando un meccanismo di fork-join. Come executor è stato utilizzato un **ForkJoinPool** a cui è stato registrato uno ScanFolderTask iniziale.

Al suo interno ScanFolderTask esegue una **fork** per ogni subfolder al suo interno creando ulteriori ScanFolderTask, mentre per ogni documento .java effettua una fork facendo eseguire un CountLinesTask.

Al termine di tutte le fork, esegue una **join** per ogni Task figlio attendendo i risultati e restituendoli come risultato della propria terminazione.

Il CountLinesTask si occupa semplicemente del conteggio delle linee di codice.

**Metodo getReport()**

Il metodo getReport riceve in input tutte le informazioni di setup per poter inizializzare l’esecuzione e restituisce in output un **ForkJoinTask** (Che implementa l’interfaccia Future). In questo modo quando la CLI richiama getReport riceve istantaneamente una Future vuota e in modo asincrono, quando l’esecuzione dei Task termina, essa viene riempita con il risultato.

**Metodo analyzeSources ()**

Il metodo analizeSources() riceve anch’esso le informazioni per il setup dell’esecuzione, ma a differenza di getReport(), restituisce il riferimento ad un oggetto di tipo Result che conterrà il risultato. Successivamente tramite il pattern **Observer**, verrà notificato di nuovi risultati disponibili e in questo modo andrà a consultare la variabile corrispondente.

**Prove di performance**

Sono state effettuate delle misurazioni dei **tempi di esecuzione** su tre progetti Java diversi e per ognuna di esse viene riportata la media delle tempistiche in millisecondi.

* Progetto piccolo (111 sorgenti): **124 ms**
* Progetto medio (1569 sorgenti): **360 ms**
* Progetto grande (8023 sorgenti): **860 ms**

**2) Approccio basato su Virtual Threads**

**Architettura e strategia risolutiva**

Analogamente alla strategia precedente, anche in questo caso è stato applicato il pattern architetturale **MVC** per mantenere una separazione semantica tra le varie parti del sistema e favorire la realizzazione di una View a linea di comando sostituibile con la GUI.

Per quanto riguarda questo approccio, la struttura architetturale del sistema è simile alla precedente, sono stati mantenuti entrambi i Task: **ScanFolderTask** (per la scansione delle folders) e **CountLinesTask** (per il conteggio delle linee di un file).

In questa strategia le principali differenze riguardano il comportamento nel dettaglio del sistema.

**Comportamento del sistema**

**Esecuzione dei Task**

In questo caso i Task vengono eseguiti da **Virtual Threads**, in particolare l’esecuzione inizia con l’istanziazione di un **rootThread** che esegue il primo ScanFolderTask prendendo in considerazione la root iniziale del progetto dato in input.

Al suo interno lo **ScanFolderTask** si occupa di eseguire un VirtualThread per ogni subfolder e ogni documento al suo interno, assegnando ad ognuno il Task corretto. Ogni ScanFolderTask ha inoltre il compito di effettuare la **join** su tutti i Task figli da esso lanciati.

Il CountLinesTask effettua il conteggio delle righe e l’aggiunta del risultato ottenuto.

**Metodo getReport()**

Il metodo getReport() si occupa di eseguire il rootThread e restituisce un **CompletableFuture** (inizialmente vuoto) che viene completato con il risultato al termine dell’esecuzione del rootThread. Di conseguenza nella View viene registrata la callback corrispondente.

**Metodo analyzeSources ()**

Questo metodo si occupa (come il precedente) di avviare l’esecuzione ma restituendo il riferimento alla variabile che conterrà incrementalmente i risultati. Successivamente tramite il pattern **Observer**, ad ogni nuovo risultato disponibile, la View viene notificata e visualizza i risultati di conseguenza.

**Prove di performance**

Sono state effettuate delle misurazioni dei **tempi di esecuzione** su tre progetti Java diversi e per ognuna di esse viene riportata la media delle tempistiche in millisecondi.

* Progetto piccolo (111 sorgenti): **148 ms**
* Progetto medio (1569 sorgenti): **420 ms**
* Progetto grande (8023 sorgenti): **1041 ms**

**3) Approccio asincrono ad eventi – Framework Vert.x**

**Architettura e strategia risolutiva**

Anche in questo caso l’architettura è stata strutturata seguendo il pattern **MVC**, ma utilizzando il framework Vert.x sono state effettuate piccole variazioni a livello di entità logiche.

Di seguito si presenta nel dettaglio la struttura realizzata:

**Model:**

* ScanFolderAgent
* CountLinesAgent

**Controller:**

* SourceAnalyzer

**View:**

* ConsoleAgent
* GUIAgent

Come si può notare, sono stati introdotti quattro **Verticle** che cooperano tra loro per realizzare le funzionalità richieste. Di seguito si procede a descrivere il loro comportamento.

**Comportamento del sistema**

**ScanFolderAgent**

Questo Verticle ha lo stesso compito che nelle fasi precedenti era incapsulato nello **ScanFolderTask,** all’inizio dell’esecuzione viene effettuato il **deploy** del primo **ScanFolderAgent** sulla root directory passata in input. Successivamente al suo interno si occupa di creare un nuovo ScanFolderAgent per ogni subfolder e un nuovo CountLinesAgent per ogni documento.

**CountLinesAgent**

Analogamente, questo Verticle si occupa di realizzare le operazioni che prima erano incapsulate nel **CountLinesTask**, contando le righe del documento assegnato e aggiungendo il risultato ottenuto.

**ConsoleAgent**

Questo Verticle, una volta eseguito, registra una callback sul metodo **getReport** e al suo completamento visualizza a Console il Report finale dei risultati.

**GUIAgent**

Il GUI Agent si mette in ascolto sull’**event-bus** utilizzato dai Verticle per comunicare, sul topic “*mid-report*”, ogni volta che arriva un messaggio, esso si occupa di aggiornare la visualizzazione dei risultati incrementali.

**Metodo getReport()**

Il metodo getReport() avvia il primo **ScanFolderAgent** sulla root directory, poi restituisce una **Future** del risultato inizialmente vuota. Esso registra inoltre una callback per il completamento dell’esecuzione, nella quale si occupa di completare la Future con il risultato finale. In questo modo viene attivata la callback del **ConsoleAgent** e viene effettuata la visualizzazione.

**Metodo analyzeSources()**

In questo approccio analogamente a quelli precedenti, il metodo avvia l’esecuzione e restituisce il riferimento alla variabile che conterrà i risultati. A differenza delle strategie precedenti però, non viene adottato il pattern Observer, ma viene sfruttato l’**event-bus** di Vert.x. Ogni volta che viene analizzato un nuovo file, il **CountLinesAgent** effettua una **publish** sul topic “*mid-report*” attivando la **consume** del **GUIAgent** in ascolto sul medesimo topic che si occupa dell’aggiornamento dell’interfaccia.

**Prove di performance**

Sono state effettuate delle misurazioni dei **tempi di esecuzione** su tre progetti Java diversi e per ognuna di esse viene riportata la media delle tempistiche in millisecondi.

* Progetto piccolo (111 sorgenti): **164 ms**
* Progetto medio (1569 sorgenti): **585 ms**
* Progetto grande (8023 sorgenti): **1546 ms**

**4) Approccio basato su programmazione reattiva – Framework RxJava**

**Architettura e strategia risolutiva**

Anche in questo ultimo approccio è stato applicato il pattern **MVC**, per mantenere una separazione semantica delle varie parti del sistema.

In questa parte non sono state individuate entità attive che corrispondono esattamente allo scanning delle directory o al conteggio dei documenti (come, ad esempio, lo ScanFolderAgent oppure il CountLinesAgent) ma sfruttando la gestione dei **Flowable** è stato realizzato un sistema che sfrutta la computazione asincrona per risolvere il problema descritto.

**Comportamento del sistema**

**Metodo analizeFolder()**

Questo metodo si occupa di generare un **Flowable** di **AnalizedFile** (Singolo risultato che contiene il nome del file e il suo numero di righe), a partire da una folder (la root del progetto considerato). Al suo interno prima ottiene un **Flowable** di tutte le subfolder, poi ognuna di esse viene mappata nei documenti .java contenuti al suo interno. Infine, ogni documento presente nello stream viene mappato con il risultato corrispondente.

**Metodo getReport()**

Questo metodo utilizza il metodo *analizeFolder()* precedentemente descritto ottenendo un **Flowable** di risultati singoli. Il suo compito principale è raggruppare i singoli file analizzati nel risultato finale e lo fa accumulandoli con una **reduce**all’interno di una collezione Result (Che viene emessa dopo che la reduce ha accumulato tutti i file). Questo metodo è asincrono in quanto restituisce un Flowable su cui lato view viene registrata una **callback**, che ottiene il risultato finale.

**Metodo analyzeSources()**

Anche questo metodo fa uso di *analizeFolder()* ma in questo caso anziché accumulare tutti i file analizzati con **reduce** e poi emettere, essi vengono raggruppati ogni volta che c’è un nuovo risultato disponibile (tramite una **map**), di conseguenza ad ogni nuovo risultato analizzato viene emesso un nuovo risultato parziale, che viene visualizzato nella **callback** corrispondente della GUI.

**ConsoleView e GUIView**

In entrambe le View vengono registrate le callback rispettivamente su getReport() e analyzeSources(), facendo eseguire la computazione delle **subscribe** su Thread diversi, in modo da non bloccare l’esecuzione principale.

**Prove di performance**

Sono state effettuate delle misurazioni dei **tempi di esecuzione** su tre progetti Java diversi e per ognuna di esse viene riportata la media delle tempistiche in millisecondi.

* Progetto piccolo (111 sorgenti): **414 ms**
* Progetto medio (1569 sorgenti): **1707 ms**
* Progetto grande (8023 sorgenti): **2626 ms**