PCD Assignment 02 - Find the Dependencies

A cura di

Alessandra Versari - alessandra.versari2@studio.unibo.it

Lorenzo Rigoni - lorenzo.rigoni2@studio.unibo.it

Riccardo Moretti - riccardo.moretti6@studio.unibo.it

Analisi del problema

In questo assignment è stato richiesto di implementare un sistema in grado di analizzare le dipendenze di un progetto Java.

L'elaborato si compone di due fasi distinte:

1. **Fase asincrona - Implementazione della libreria DependencyAnalyserLib**
   In questa fase è stato richiesto di realizzare una libreria per l'analisi delle dipendenze tra classi, interfacce e pacchetti di un progetto Java, utilizzando la programmazione asincrona.
   La libreria fornisce tre metodi principali:
   - *getClassDependencies(classSrcFile)*: restituisce, in modo asincrono, l'elenco dei tipi (classi o interfacce) utilizzati o accessibili dalla classe specificata;
   - *getPackageDependencies(packageSrcFolder)*: restituisce l'elenco dei tipi utilizzati da qualunque classe o interfaccia all'interno del pacchetto;
   - *getProjectDependencies(projectSrcFolder)*: restituisce l'elenco dei tipi utilizzati da qualunque classe o interfaccia nel progetto.

   Per la programmazione asincrona è stato utilizzato il framework *Vertx* di Java, mentre per il parsing dei file Java è stata adottata la libreria *JavaParser*.

2. **Fase reattiva - Sviluppo dell'applicazione GUI DependencyAnalyser**
   La seconda parte dell'elaborato prevede la realizzazione di un'applicazione dotata di interfaccia grafica, sviluppata utilizzando la programmazione reattiva.
   La GUI consente all'utente di:
   - selezionare la cartella contenente il codice sorgente;
   - avviare l'analisi delle dipendenze;
   - visualizzare il grafo delle dipendenze tra classi/interfacce;
   - monitorare in tempo reale il numero di classi/interfacce analizzate e il numero di dipendenze rilevate.

   In questa fase è stato adottato *RxJava* per gestire lo stream di eventi e aggiornare dinamicamente l'interfaccia utente durante l'analisi.

L'intero sistema, pur affrontando lo stesso problema, è stato progettato e implementato due volte sfruttando due paradigmi distinti (asincrono e reattivo), come richiesto dalla consegna.

Design, strategia e architettura

Programmazione asincrona

Nella parte di programmazione asincrona, i metodi richiesti dalla consegna sono stati implementati nella classe *DependencyAnalyserLib*, la quale utilizza la libreria *JavaParser* per trovare tutte le dipendenze tra classi e interfacce. Per rappresentare i risultati delle analisi, sono state create tre classi:

- *ClassDepsReport* (report sulle classi/interfacce) il quale contiene il nome della classe/interfaccia, il nome del package in cui è contenuta e l'elenco delle dipendenze;
- *PackageDepsReport* (report sui package) il quale contiene il nome del package e l'elenco dei report delle classi/interfacce appartenenti ad esso;
- *ProjectDepsReport* (report sui progetti) il quale contiene il nome del progetto e l'elenco dei report dei package appartenenti ad esso.

Per utilizzare al meglio i meccanismi della programmazione asincrona, tutti e tre i metodi restituiscono il risultato dentro ad un oggetto *Future*. Inoltre, per garantire che in esecuzione venga utilizzato solo l'event-loop di *Vertx*, il test di questi metodi è stato svolto nella classe *DependencyAnalyserVerticle* la quale eredita dalla classe *AbstractVerticle*.

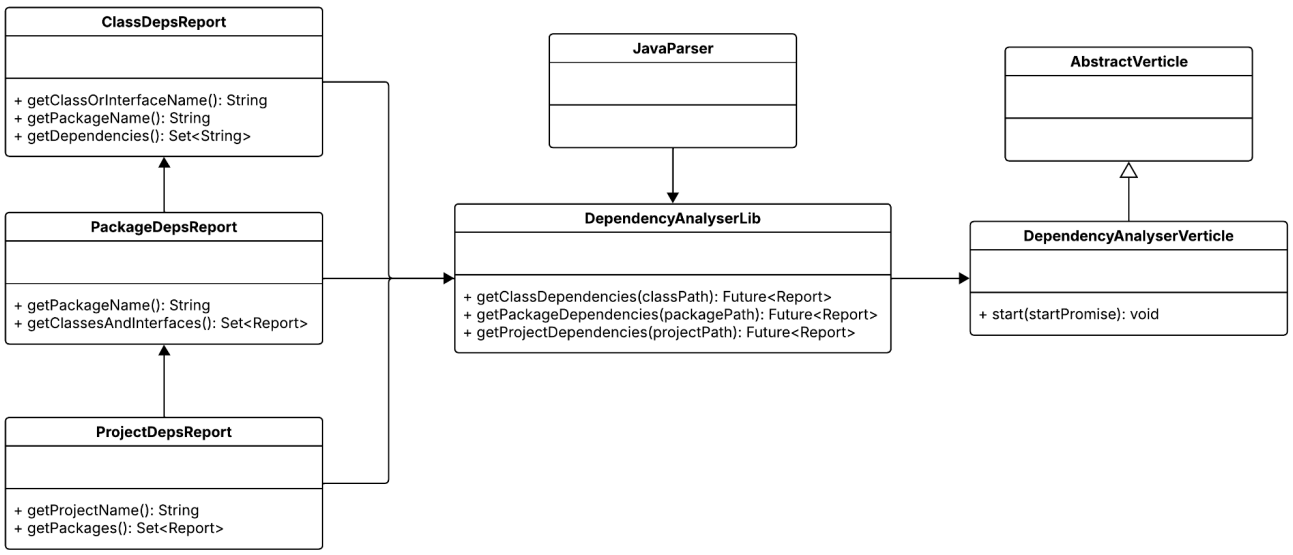


Diagramma della parte asincrona del progetto

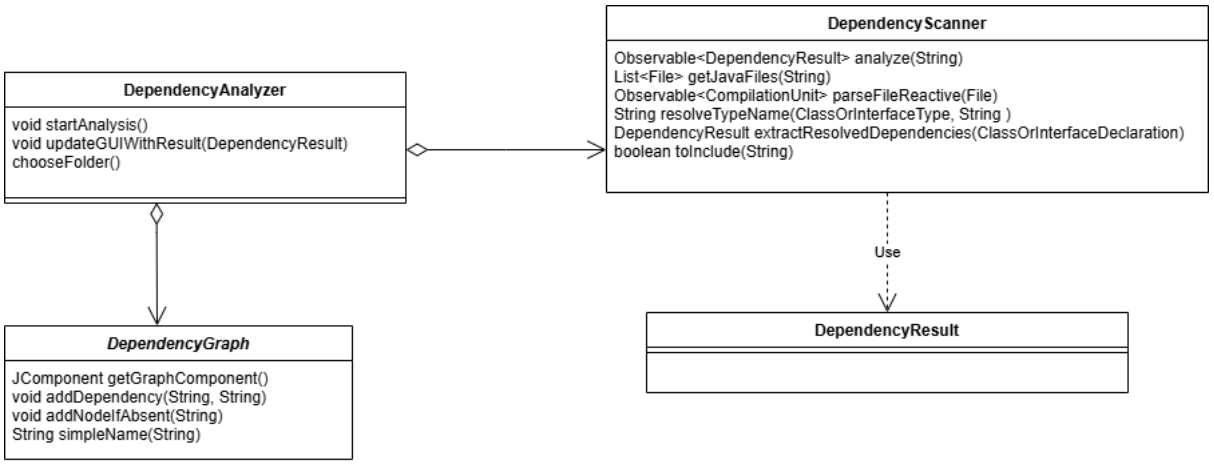
Programmazione reattiva

Per la seconda parte dell'assignment è stata sviluppata un'applicazione desktop reattiva in Java, dotata di un'interfaccia grafica realizzata con *Swing*, che consente di analizzare le dipendenze tra classi Java. Per l'analisi del codice sorgente viene utilizzata la libreria *JavaParser*, mentre l'approccio reattivo è implementato attraverso *RxJava*. I risultati dell'analisi vengono visualizzati graficamente tramite *GraphStream*, la libreria scelta per la rappresentazione del grafo delle dipendenze.

L'architettura del progetto è suddivisa in quattro componenti principali:

- *DependencyAnalyzer*: gestisce l'interfaccia utente e il controllo dell'applicazione.
- *DependencyScanner*: si occupa del parsing, dell'analisi e della risoluzione delle dipendenze tra classi.
- *DependencyGraph*: gestisce la visualizzazione grafica delle dipendenze identificate.
- *DependencyResult*: rappresenta l'output dell'analisi, contenente le informazioni sulle classi e le relative dipendenze.

Per applicare efficacemente i principi della programmazione reattiva affrontati a lezione, la classe *DependencyScanner* utilizza *Observable* per creare un flusso reattivo in cui ogni *file.java* è trattato come un evento indipendente. Grazie all'uso combinato di operatori come *flatMap*, *map*, e alla gestione asincrona dei *thread* , il processo di analisi è completamente non bloccante. I risultati vengono propagati man mano che vengono prodotti, consentendo un aggiornamento incrementale del grafo. Questo approccio garantisce una UI responsiva, anche durante l'elaborazione di grandi progetti.



Comportamento del sistema

Programmazione asincrona

Nella parte asincrona del progetto, i tre metodi di analisi svolgono i propri calcoli tramite la concatenazione di oggetti *Future* grazie all'uso del metodo *compose,* il quale permette di appiattire la piramide delle concatenazioni.

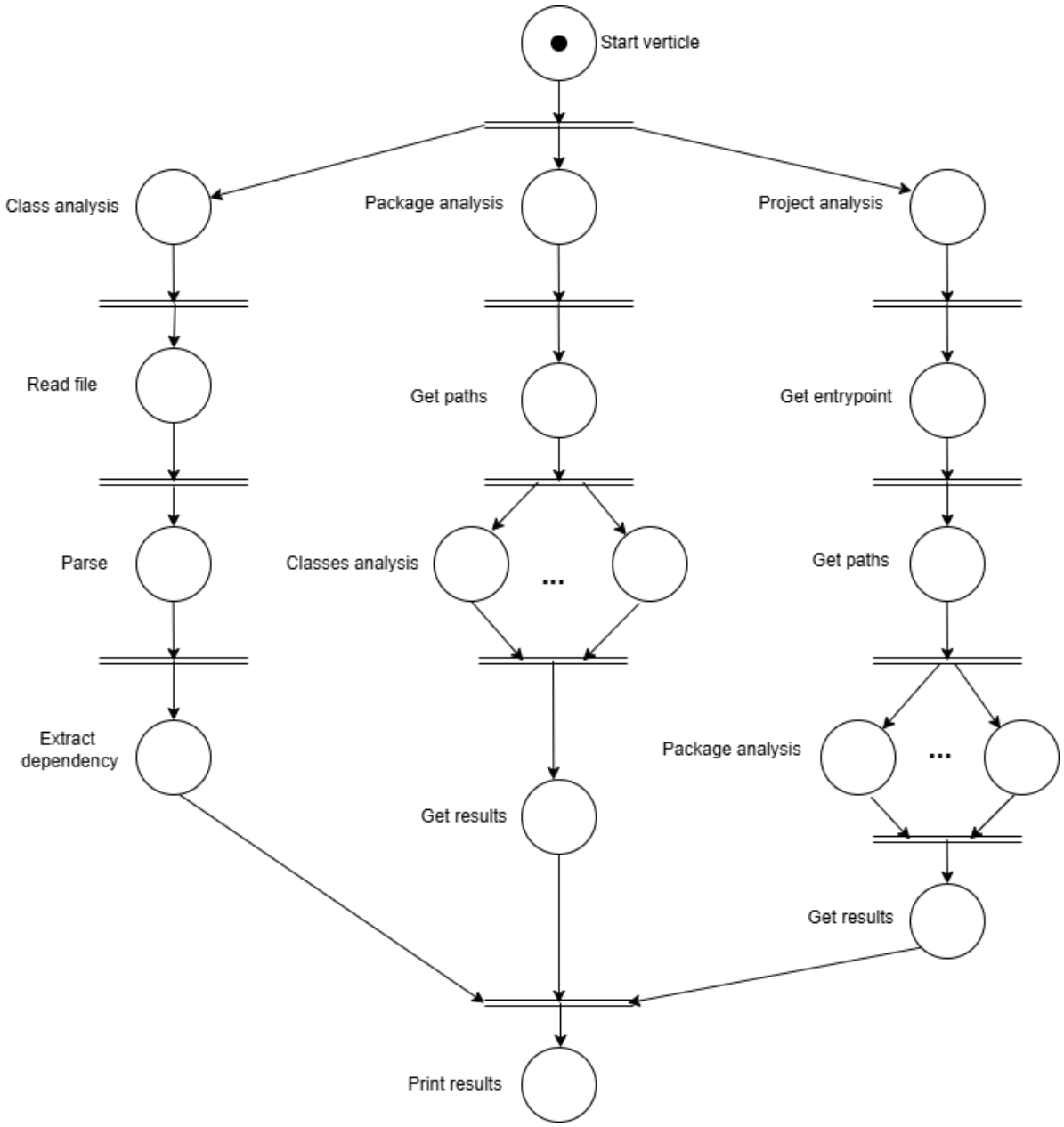
Innanzitutto, nel main viene fatto il deploy del *Verticle*, il quale contiene le tre chiamate asincrone e i metodi di stampa una volta ricevuti i risultati.

Nell'analisi della classe, viene prima di tutto letto il file sorgente. Essendo un'azione bloccante, questa viene svolta tramite la funzione asincrona *readFile* di *Vertx*. Fatto ciò viene poi svolto il parsing del codice tramite la libreria *JavaParser*. Essendo anche questa un'azione bloccante, è stato utilizzato il metodo *executeBlocking*. Ottenuta l'unità compilata, si può procedere alla visita dell'AST per la ricerca delle dipendenze della classe/interfaccia (anche questa operazione fatta nel metodo *executeBlocking*). Infine, se si ottiene un risultato senza errori, questo viene inserito nella classe *ClassDepsReport*.

Per quanto riguarda l'analisi del package, lo svolgimento è abbastanza simile al precedente: vengono prima presi tutti i path delle classi presenti nel package e poi, in una lista di *Future*, vengono inserite le chiamate asincrone al metodo *getClassDepedencies*. Una volta fatto ciò, tramite il metodo *Future.all* si aspetta che tutte le dipendenze siano state calcolate. Se non vengono generate eccezioni, il sistema scompone l'oggetto *CompositeFuture* restituito dal metodo precedente e crea il report per il package.

L'analisi del progetto ha un comportamento analogo a quello per i package ma con l'unica differenza che, invece dei path delle classi, vengono cercati i path dei package.

Quando tutti e tre i metodi hanno finito i loro compiti, nel *Verticle* si può procedere alla stampa dei risultati.



Rete di Petri della parte asincrona

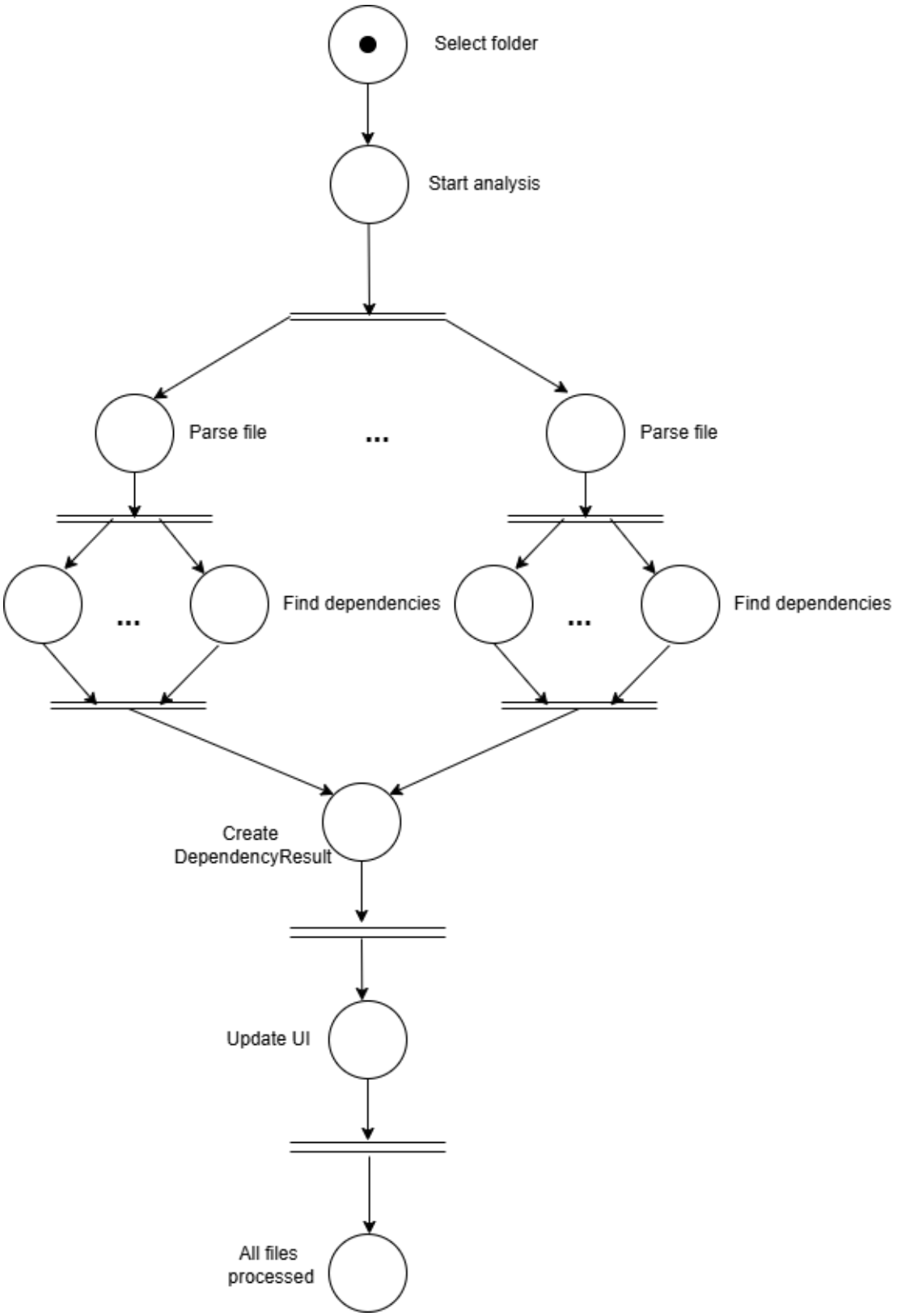
Programmazione reattiva

Una volta avviata l'analisi, la classe *DependencyAnalyzer* inizializza un nuovo *DependencyScanner*, che inizia ad esaminare ricorsivamente la directory selezionata per individuare tutti i file.java. L'elaborazione di ciascun file è gestita da un flusso reattivo (*Observable<File>*), permettendo di processare i file in parallelo su un thread pool I/O (*Schedulers.io()*), evitando così il blocco della *Event Dispatch Thread* di Swing. Ogni file viene quindi parsato tramite *JavaParser* e trasformato in un *CompilationUnit*, che viene ulteriormente analizzato per estrarre classi e le relative dipendenze.

Per mantenere la chiarezza e la rilevanza del grafo generato, il sistema esclude volutamente dall'analisi alcune dipendenze considerate di uso generico o non rilevanti per il progetto. In particolare, vengono filtrati i tipi appartenenti a pacchetti standard come *java.lang, java.util, java.io, java.net, java.time, java.math, java.text, java.nio, javafx e org.graphstream*, nonché i tipi fondamentali come *String, Object, Throwable, Exception, RuntimeException ed Error*. Questa scelta progettuale consente di focalizzare l'analisi sulle classi create dall'utente riducendo il rumore e migliorando la leggibilità del grafo risultante.

La visualizzazione delle dipendenze, pur essendo funzionale, ha sicuramente margini di miglioramento, in particolare per quanto riguarda la gestione dello spazio e la distribuzione dei nodi nel grafo. In alcuni casi, la densità dei collegamenti può rendere difficile l'interpretazione visiva della struttura complessiva. Tuttavia, è stata adottata la libreria GraphStream proprio per delegare la gestione grafica e potersi concentrare maggiormente sull'obiettivo primario del progetto: l'utilizzo dei principi della programmazione reattiva.

Per ogni classe trovata, viene generato un oggetto *DependencyResult*, che rappresenta il risultato dell'analisi e che viene emesso nel flusso principale. Questi risultati vengono osservati nel thread grafico (*Schedulers.single()*), dove *DependencyAnalyzer* aggiorna progressivamente il conteggio delle classi analizzate e delle dipendenze trovate. Contestualmente, il grafo viene modificato in tempo reale dal componente *DependencyGraph*, che si occupa di visualizzare le relazioni tra classi sotto forma di nodi e archi, evitando la duplicazione dei nodi già presenti.



Rete di Petri della parte reattiva