Pcd-Assigment02 Report

Find the Dependencies

Bedeschi Federica
0001175655 federica.bedeschi4@studio.unibo.it

Pracucci Filippo
0001183555 filippo.pracucci@studio.unibo.it

Anno accademico 2024-2025

Indice

Analisi

1.1 Descrizione del problema

1.1.1 Programmazione asincrona

Il problema richiedeva lo sviluppo di una liberia asincrona `DependencyAnalyserLib` per effettuare l'analisi delle dipendenze di un progetto Java, tenendo traccia dei tipi (interfacce/classi) e dei package utilizzati da ogni interfaccia/classe/package del progetto. La libreria comprende lo sviluppo di tre metodi asincroni:

- `getClassDependencies(classSrcFile)`: produce come risultato la lista dei tipi usati dalla classe;
- `getPackageDependencies(packageSrcFolder)`: produce come risultato la lista dei tipi usati da tutti i sorgenti del package;
- `getProjectDependencies(projectSrcFolder)`: produce come risultato la lista dei tipi usati da tutti i sorgenti del progetto.

Abbiamo deciso di implementare un metodo per escludere alcune dipendenze. Nel nostro caso l'abbiamo utilizzato per escludere le dipendenze del package java.lang, in quanto contiene i tipi standard del linguaggio.

1.1.2 Programmazione reattiva

Il problema richiedeva lo sviluppo di un programma con interfaccia grafica chiamato `DependencyAnalyser`, usando un approccio basato sulla programmazione reattiva. Questo fornisce la possibilità di analizzare e mostrare dinamicamente/incrementalmente un grafo delle dipendenze trovate nelle interfacce/classi del progetto selezionato dall'utente, raggruppandole per interfaccia/classe e per package.

1.2 Visione concorrente

In entrambi i casi abbiamo adottato il linguaggio Java; per fare il parsing dei sorgenti e generare gli AST abbiamo utilizzato la libreria JavaParser.
Per quanto riguarda la programmazione asincrona abbiamo utilizzato il framework Vert.x. L'implementazione dei tre metodi asincroni produce come valore di ritorno una `Future` del relativo report delle dipendenze.
Per quanto riguarda la programmazione reattiva abbiamo utilizzato il framework RxJava, tramite il quale creiamo uno stream delle dipendenze di ogni singolo sorgente, reagendo alla produzione di ogni elemento con l'aggiornamento della GUI, così da ottenere un comportamento incrementale.

Design e architettura

2.1 Programmazione asincrona

Il metodo `getClassDependencies` ha il compito di analizzare un singolo sorgente e trovarne le dipendenze, mentre gli altri due si occupano di capire la struttura del file system partendo dalla directory sorgente per poi richiamarsi a cascata fino ad arrivare a `getClassDependencies` in modo da ottenere le dipendenze dei singoli sorgenti. Ogni metodo produce una diversa tipologia di report delle dipendenze, anch'essi composti nella modalità sopracitata. Abbiamo inserito una classe di test per mostrare un comportamento tipico di utilizzo della libreria.

2.1.1 Rete di Petri

La rete di Petri in Figura 2.1 descrive il comportamento del sistema, mostrando la composizione dei metodi come descritto sopra. Il sistema parte con un singolo token ad effettuare la transizione `getProjectDependencies`, per poi generare tanti token quanti sono i package. Successivamente viene eseguita la transizione `getPackageDependencies`, generando tanti token quante sono le classi/interfacce, per poi eseguire la transizione `getClassDependencies`.

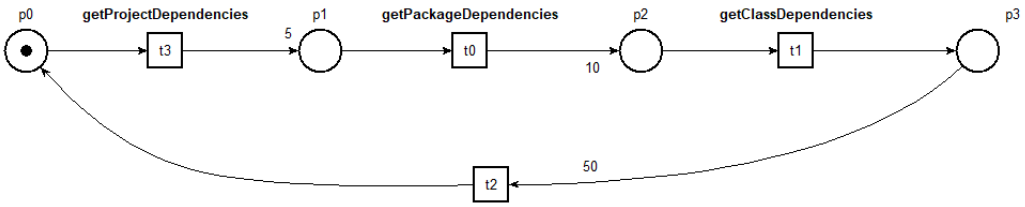

Figura 2.1: Rete di Petri - programmazione asincrona
Il 5 presente in figura rappresenta in realtà il numero di package del progetto, mentre il 10 rappresenta un valore simbolico del numero di classi di ogni package.

2.2 Programmazione reattiva

Abbiamo optato per un'architettura che seguisse il pattern *MVC*, come mostrato in Figura 2.2, quindi con un *Controller* che si occupasse di mettere in comunicazione il *Model* con la *View*. Il Controller ha il compito di ottenere dal `DependencyAnalyser` l'`Observable` dello stream composto dai report delle dipendenze di ogni classe/interfaccia, sul quale effettua la `subscribe` per aggiornare la GUI incrementalmente. La visualizzazione delle dipendenze avviene utilizzando un albero in cui le foglie sono le dipendenze raggruppate per sorgente e per package.

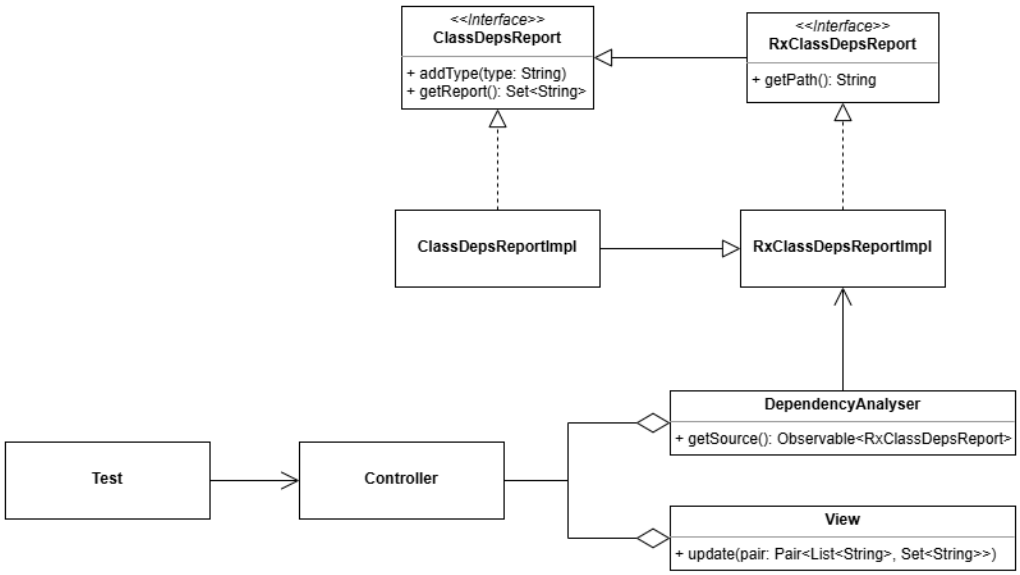

Figura 2.2: Diagramma delle classi - programmazione reattiva

2.2.1 Rete di Petri

La rete di Petri in Figura 2.3 descrive il comportamento del sistema, mostrando la produzione continua di token tramite la transizione `getClassDependencies`, che rappresentano i report prodotti. In seguito alla produzione di un report viene eseguita la transizione `UpdateGUI`. Abbiamo inserito la transizione *t2* per simulare la terminazione dei file sorgenti, che conclude l'esecuzione.

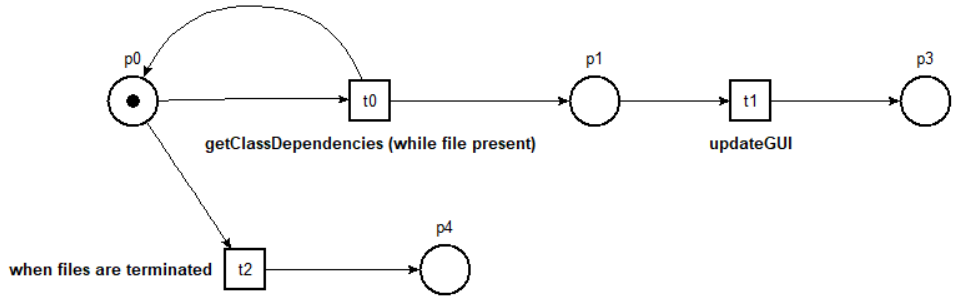

Figura 2.3: Rete di Petri - programmazione reattiva

Conclusioni

La libreria realizzata con un approccio asincrono ci ha permesso di sfruttare le caratteristiche del paradigma, cioè consentendo la continuazione dell'esecuzione senza dover rimanere bloccati in attesa del risultato. Inoltre, trattandosi di una libreria, è possibile utilizzarla per diverse applicazioni, sfruttando i metodi utili nel caso specifico; nel nostro caso è stato realizzato un semplice test per mostrare il suo funzionamento.
Per quanto riguarda la programmazione reattiva abbiamo potuto notare come quest'ultima sia particolarmente adatta in un contesto incrementale, quindi per esempio nel nostro caso è perfetta per ottenere un aggiornamento dinamico dell'interfaccia grafica, grazie alla flessibilità offerta dall'utilizzo degli stream.