Assignment #02 - Find the Dependencies

Buda Francesco, Sanchi Emanuele, Severi Tommaso

Indice

Analisi del problema

Il problema proposto consiste nell'analisi delle dipendenze di un progetto java che sfrutti sia la programmazione asincrona, per realizzare una libreria di metodi che permettano di analizzare rispettivamente una classe, un package e un intero progetto; sia la programmazione reattiva, per realizzare un'interfaccia grafica che costruisca in maniera dinamica l'albero delle dipendenze di un progetto.
Il nucleo del problema è garantire un'analisi efficiente e scalabile delle dipendenze all'interno di un progetto, gestendo in parallelo operazioni di lettura file e parsing del contenuto. Nell'approccio asincrono, si tratta di spezzare il flusso di lavoro in task indipendenti: lettura di un file, analisi sintattica ed estrazione delle dipendenze, orchestrando la catena di passi attraverso notifiche di completamento. Invece, nell'ottica reattiva, l'intero processo si concepisce come un flusso continuo di dati, con meccanismi intrinseci per mantenere una visione incrementale e sempre coerente dell'albero delle dipendenze mostrato all'utente.

Design, architettura e strategia

Nonostante le implementazioni differiscano per il paradigma e la logica utilizzati, rimane al cuore dell'implementazione la libreria Java Parser e le sue funzionalità, modificate al fine di ottenere il risultato desiderato. Per questo motivo esistono due classi comuni alle due implementazioni:

- DependencyCollector:
  estende il VoidVisitorAdapater di Java Parser per ampliare l'implementazione della visita dei nodi interessati
- DependencyInfo:
  racchiude al suo interno le informazioni raccolte durante la visita dell'AST

Nella Fig.1 è riportata l'architettura delle due classi.

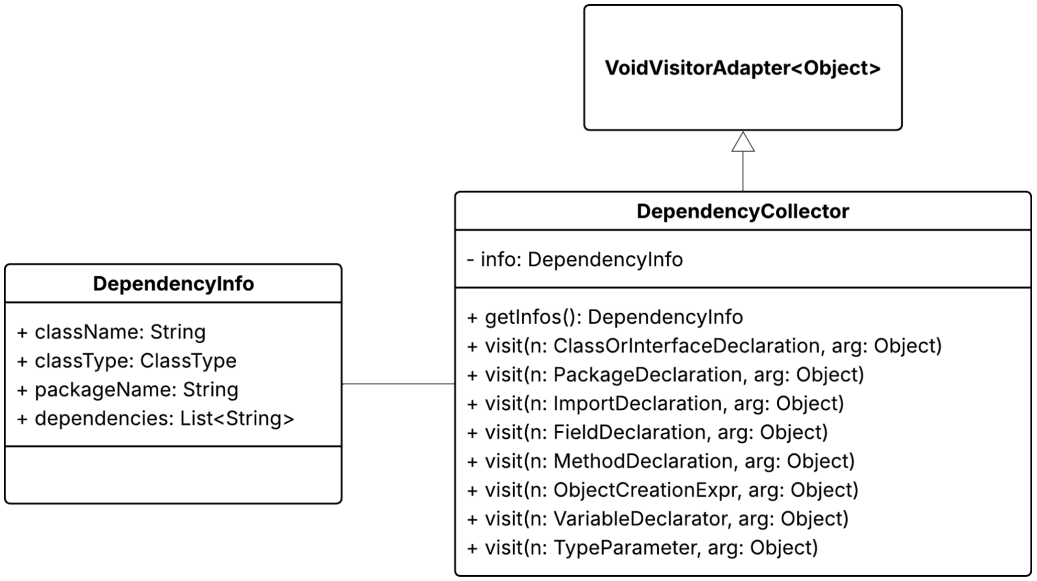

Fig.1 UML classi comuni

Asynchronous

Il design della libreria asincrona per l'estrazione delle dipendenze si basa sul concetto di Promise. Ciascun metodo costruisce la promessa del risultato atteso sfruttando API asincrone.
Nella Fig.2 viene mostrata l'architettura della libreria che si compone di due oggetti che incapsulano rispettivamente le logiche di: esplorazione asincrona del file system del progetto e estrazione delle dipendenze tramite parsing di file Java.

- ProjectFileSystem:
  sfrutta il framework vert.x per navigare i package del progetto, in particolare è in grado di estrarre sia i file java contenuti all'interno di una directory sia di ottenere ricorsivamente il percorso di tutti i file java di un progetto

- ProjectParser:
  sfrutta la libreria JavaParser per realizzare i metodi utili all'estrazione di dipendenze da file java. Il parsing dei file viene reso asincrono grazie al thread apposito messo a disposizione da vert.x per l'esecuzione di codice bloccante.

Infine vengono utilizzate delle classi di report le quali immagazzinano le dipendenze di Classi/Package/Progetti e permettono di stamparle su terminale in modo comprensibile.

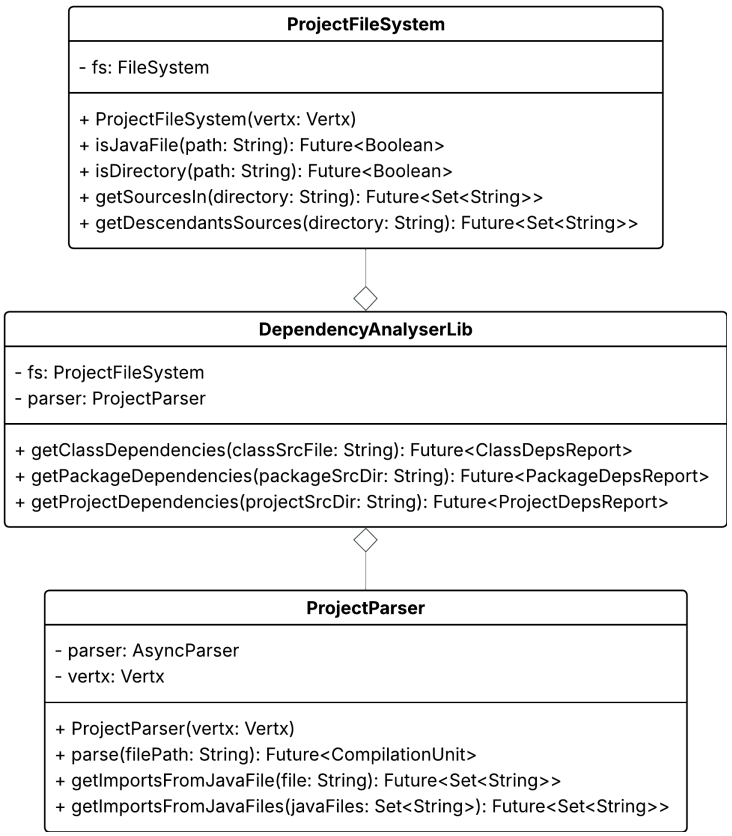

Fig.2 UML classi asincrone

Reactive

La struttura del sistema reattivo è apparentemente semplice in quanto necessita solo della classe in Fig.3 per rappresentare il sistema.

- ReactiveParser:
  permette di analizzare le dipendenze di un progetto, sfruttando le classi per il parsing comuni, ma, invece di ritornare un risultato, richiede una funzione consumatore che possa essere lanciata ogni qual volta la visita di ogni singolo AST viene completata, rendendo l'aggiornamento, potenzialmente, dinamico.

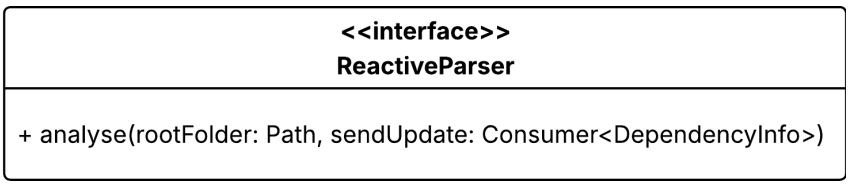

Fig.3 UML classi reattive

L'interfaccia grafica sfrutta l'architettura del parser per passargli come consumatore un emitter di un Flowable della libreria RxJava, ovvero una funzione in grado di catturare gli aggiornamenti, in questo caso, composti dalle dipendenze di ogni singolo file da cui sono state raccolte.
Infatti, il comportamento del sistema è pensato come una sequenza di dati, che inizia nel momento in cui viene premuto il bottone, alla quale l'interfaccia risponde nel momento in cui questi vengono "pubblicati".
Come mostrato in Fig.4 viene creato uno stream di tipo cold, lanciato solo dopo aver definito il comportamento dell'osservatore con i parametri corretti.

1. Viene adottata una strategia di buffer per gestire la backpressure nel caso ci sia un eccessivo numero di file.

2. Per l'esecuzione del codice della sorgente viene scelto uno scheduler ottimizzato per operazioni di I/O, in quanto verranno letti dei file.

3. Per l'esecuzione del codice di aggiornamento dell'interfaccia viene delegato alle funzionalità di swing, assicurandosi che che vengano eseguite in modo sicuro nell'EDT.

```
1  source ← createFlowable(emitter → parser.analyse(root, emitter::onNext))
2  source
3    .onBackpressureBuffer(100, () → exception("Buffer overflow"))
4    .subscribeOn(Schedulers.io())
5    .observeOn(Schedulers.from(SwingUtilities::invokeLater))
6    .subscribe(info → {...})
```

Fig.4 Codice reattivo

Comportamento del sistema

Asynchronous

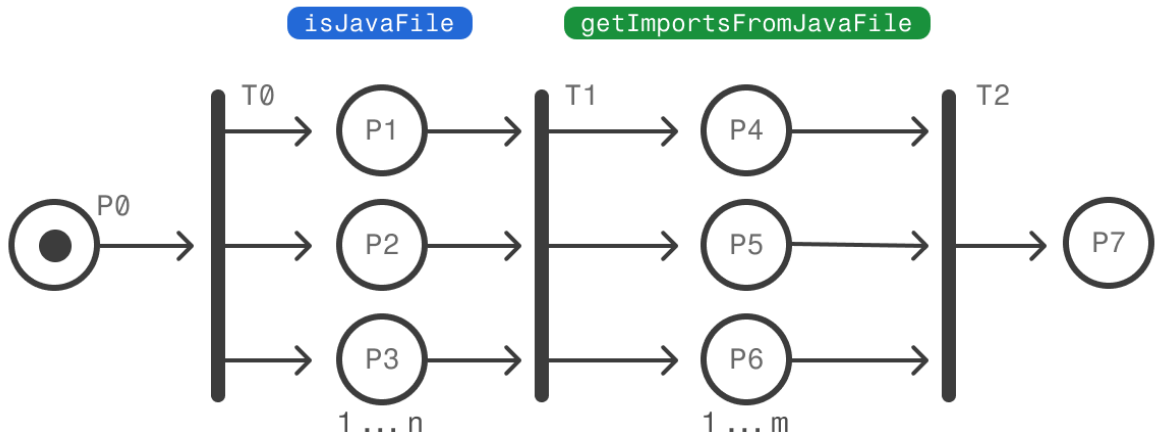

Fig.5 Petri Net asincrona

Mentre il metodo per l'estrazione delle dipendenze da una classe è pensato con una struttura che può essere messa in corrispondenza con una sequenziale, nonostante sia gestito comunque in maniera asincrona, i metodi per l'estrazione delle dipendenze da un package e da un progetto sfruttano il parallelismo intrinseco nella loro definizione. Come mostrato nella rete di petri a Fig.5, infatti, sia il controllo dell'esistenza dei file che il parsing stesso e l'estrazione delle dipendenze vengono svolti parallelamente. A livello di implementazione questo si traduce nella costruzione parallela, all'interno di un ciclo for, di un numero n di promise (Future in Vert.x) che vengono poi "aspettate" e ricongiunte in un'unica promise grazie al metodo "Future.all()" che a livello di design parallelo ha lo stesso ruolo di una barriera per sincronizzare diversi agenti.

Reactive

Nella rete di petri a Fig.6 è mostrato il funzionamento del sistema nel caso dell'implementazione reattiva.
Si può notare come il flusso sia pressoché sequenziale e, a differenza del funzionamento asincrono, non è necessario l'uso di barriere; infatti, la GUI viene aggiornata ogni volta che un file che viene analizzato, non al termine di tutte le operazioni di analisi del progetto.
Anche se il flusso rappresentato dalla rete risulta infinito, in un caso reale il numero di file all'interno di un progetto è finito, quindi il flusso si andrà sempre a concludere.

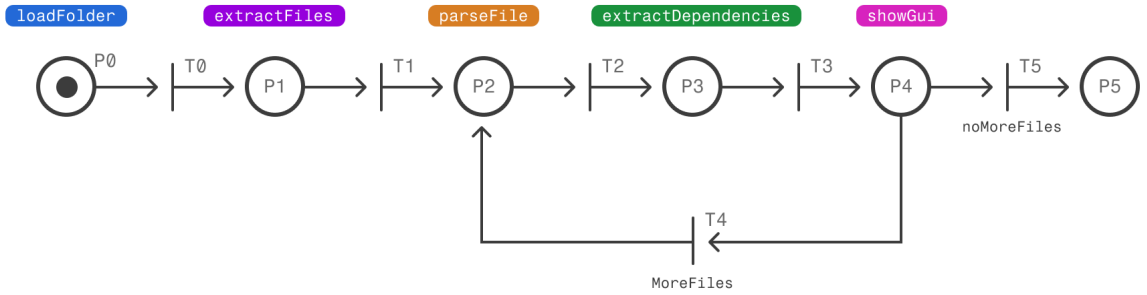

Fig.6 Petri Net reattiva

In alternativa, si può modificare la rete inserendo all'uscita della transizione t1 peso pari al numero dei file da analizzare e senza la doppia transizione t4 t5. Il risultato è mostrato alla Fig.7 in cui sono inseriti 10 token, nel caso reale files, come esempio.

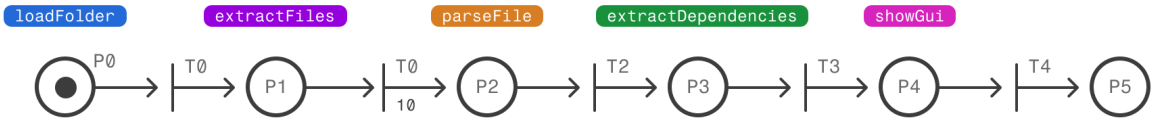

Fig.7 Petri Net reattiva 2.0