# Relazione Progetto

Dependency Analyser Lib

Meco Daniel - Diego Coli - Marco Prandini Costantini - Gabriele DUCEse

April 28, 2025

**Abstract**

In questa relazione presentiamo la libreria `DependencyAnalyserLib`, sviluppata per l'analisi asincrona delle dipendenze in progetti Java. Verranno illustrate le classi principali, il loro funzionamento, le interazioni e la strategia di testing implementata.

## Contents

# 1 Introduzione

La libreria `DependencyAnalyserLib` fornisce un insieme di metodi per analizzare le dipendenze nei sorgenti Java. Il flusso di analisi copre tre livelli differenti:

1. **Classe**: Dato un singolo file Java, individua gli import dichiarati.
2. **Package**: Analizza tutti i file Java di una cartella, aggregando le dipendenze delle classi.
3. **Progetto**: Esamina tutti i package in una struttura di progetto e restituisce un report complessivo.

# 2 Struttura del Progetto

Il progetto è organizzato in modo modulare, con directory dedicate a:

- **parser**: Contiene *JavaParser.js*, responsabile del parsing asincrono dei file Java tramite la libreria `java-parser`.
- **lib**: Contiene *DependecyAnalyserLib.js*, la classe principale con i metodi di analisi.
- **models**: Contiene i modelli `ClassDepsReport`, `PackageDepsReport` e `ProjectDepsReport`, che incapsulano i risultati delle analisi.
- **test**: Include i vari test (*testAnalyzer.js* e *testStructure.js*) per verificare il corretto funzionamento dell'intera libreria.

# 3 Componenti Principali

## 3.1 1. Classe `DependecyAnalyserLib`

`DependecyAnalyserLib` è la classe principale che fornisce i metodi asincroni per l'analisi dei file Java

Le Promise (promesse) sono oggetti in JavaScript che rappresentano il completamento o il fallimento di un'operazione asincrona.

Definizione: Una Promise è un oggetto che rappresenta un valore che potrebbe essere disponibile ora, in futuro o mai.

Stati di una Promise:

- Pending: Stato iniziale, né completata né rifiutata
- Fulfilled: Operazione completata con successo
- Rejected: Operazione fallita

Funzionalità principali di fs.promises

- `fs.promises.readFile`: Legge il contenuto di un file.
- `fs.promises.writeFile` Scrive dati in un file.
- `fs.promises.readdir` Legge il contenuto di una directory.
- `fs.promises.access` Verifica se un file o una directory esiste e se è accessibile.
- `fs.promises.stat` Ottiene informazioni su un file o una directory.

**`async getClassDependencies(classSrcFile)`**

- Verifica l'esistenza del file.
- Legge il contenuto tramite il modulo `fs.promises`.
- Utilizza il parser (`JavaParser`) per estrarre le dipendenze (dichiarazioni `import`).
- Restituisce un `ClassDepsReport` con il nome della classe (estratto dal nome del file) e l'elenco delle dipendenze.

**`async getPackageDependencies(packageSrcFolder)`**

- Ricerca ricorsivamente i file Java all'interno della cartella indicata, tramite il metodo `findJavaFiles`.
- Analizza ogni file raccolto con `getClassDependencies`, in parallelo (tramite `Promise.all`).
- Aggrega i risultati in un `PackageDepsReport`, che contiene i report di ciascuna classe e una lista di dipendenze *deduplicate*.

**`async getProjectDependencies(projectSrcFolder)`**

- Identifica tutti i package del progetto via `findJavaPackages`.
- Per ogni package, invoca `getPackageDependencies`, ancora una volta in parallelo (`Promise.all`).
- Restituisce un `ProjectDepsReport` con il nome del progetto, un elenco dei package analizzati e l'insieme globale delle dipendenze.

**`async findJavaFiles(dir)`**

- Scansiona ricorsivamente il contenuto della directory.
- Raccoglie i file con estensione `.java`.
- Restituisce un array di percorsi completi dei file.

**`async findJavaPackages(projectFolder)`**

- Utilizza `findJavaFiles` per trovare tutti i file Java nel progetto.
- Estrae le directory di ogni file ed aggiunge tali percorsi in un `Set`.
- Restituisce l'elenco dei package (rappresentati dalle cartelle che contengono file Java).

## 3.2 2. Classe `JavaParser`

`JavaParser` (contenuta nel file `JavaParser.js`) si basa sulla libreria `java-parser`. Il metodo principale `extractDependencies(content)`:

- Converte il contenuto di un file Java in un AST. Un AST (Abstract Syntax Tree) è una rappresentazione strutturata e gerarchica del codice sorgente. Ogni nodo dell'AST rappresenta un costrutto sintattico del linguaggio, come dichiarazioni, espressioni, operatori, ecc. È utilizzato dai compilatori, interpreti e strumenti di analisi per comprendere e manipolare il codice sorgente.
- Il metodo parse(content) della libreria java-parser prende il contenuto di un file Java (come stringa) e lo converte in un AST. Questo AST può essere navigato per analizzare il codice sorgente.
- Naviga nella sotto-struttura `ordinaryCompilationUnit`, cercando le `importDeclaration`.

- Ricava i nomi delle dipendenze tramite le posizioni `startOffset` ed `endOffset` o interrogando i nodi dell'AST.
- Restituisce un array di stringhe contenente i nomi dei package/classi importati.

### 3.3 3. Modelli di Report

`ClassDepsReport` Contiene le dipendenze (*imports*) di un singolo file Java.

- `className`: Nome della classe analizzata.
- `dependencies`: Array di stringhe con i percorsi delle dipendenze.

`PackageDepsReport` Aggrega i `ClassDepsReport` di un package.

- `packageName`: Nome (o cartella) del package.
- `classReports`: Elenco di `ClassDepsReport`.
- `dependencies`: Collezione univoca di tutte le dipendenze estratte dalle classi.

`ProjectDepsReport` Aggrega i `PackageDepsReport` di un intero progetto.

- `projectName`: Nome (o cartella root) del progetto.
- `packageReports`: Elenco di `PackageDepsReport`.
- `dependencies`: Collezione univoca di tutte le dipendenze, rimuovendo i duplicati.

## 4 Meccanismo di Deduplicazione

I report `PackageDepsReport` e `ProjectDepsReport` contengono funzioni interne di aggregazione che rimuovono i duplicati. Ogni volta che si uniscono liste di dipendenze, viene utilizzato un `Set` per creare l'elenco finale unico.

## 5 Gestione dell'Asincronia

Tutta la logica di I/O col filesystem e di parsing viene gestita con `async/await` e `Promise.all`:

- **Parallelismo**: Più file o package possono essere analizzati in parallelo, migliorando le prestazioni su sistemi multi-core.

- **Error Handling**: Ogni blocco `try/catch` cattura le eccezioni, gestendole in modo sicuro e loggandole sulla console.

# 6 Sistema di Test

## 6.1 Scopo

Il sistema di test verifica il corretto funzionamento della libreria su diversi scenari. `testAnalyzer.js` contiene:

- `testClassDependencies()`: Analizza una singola classe di esempio e verifica le dipendenze estratte.
- `testPackageDependencies()`: Verifica che venga analizzato correttamente un package completo, controllando l'aggregazione delle dipendenze.
- `testProjectDependencies()`: Simula un progetto con più package, testando l'analisi complessiva e la deduplicazione.

## 6.2 Struttura di Test Complessa

`testStructure.js` crea dinamicamente una gerarchia di cartelle e file Java, simulando *cross-dependencies* con `import` incrociati tra package diversi. Questo consente di validare:

- La scansione ricorsiva di più cartelle (`findJavaFiles`).
- La corretta associazione file-package (`findJavaPackages`).
- L'esatto riconoscimento delle dipendenze *incrociate* tra package diversi.

## 6.3 Esecuzione

Basta lanciare:

```
node src/test/testAnalyzer.js
```

si ottiene l'esecuzione di tutti i test in sequenza e la stampa dei risultati in console.

# 7 Ottimizzazioni e Considerazioni

- **Esecuzione parallela**: `Promise.all` velocizza l'analisi di più file o package.
- **Gestione errori**: Ogni operazione I/O e di parsing è protetta da `try/catch`.
- **Logging avanzato**: Output dettagliato per file letti, AST generati e `import` trovati.
- **Estensibilità**: L'architettura è modulare e semplifica l'aggiunta di funzioni (es. trattare wildcard, analizzare dipendenze non dichiarate).
- **Deduplicazione**: Nei report aggregati, le dipendenze duplicate vengono rimosse.

# 8 Limitazioni

1. Non rileva dipendenze *non* dichiarate negli `import`.
2. Gestione wildcard incompleta (`import java.util.*`).
3. Non distingue classi, interfacce e tipi annidati.

# 9 Conclusioni

La libreria `DependencyAnalyserLib` soddisfa i requisiti dell'Assignment #02 (parte 1), fornendo:

- Analisi asincrona a livello di file, package e intero progetto.
- Report completi, con deduplicazione delle dipendenze.
- Un sistema di test *end-to-end* per validare l'intero flusso.

Il progetto può essere esteso per supportare import wildcard, rilevamento di dipendenze non dichiarate e integrazioni con sistemi reattivi o UI (richiesto nella parte 2).