Find the Dependencies

Con programmazione Asincrona e Reattiva

Autore: Sajmir Buzi

Anno Accademico 2025

Contents

1 Introduzione

`DependencyAnalyser` è stato sviluppato con l'obiettivo di risolvere una problematica comune nel contesto della programmazione Java: l'analisi delle dipendenze tra le varie componenti di un progetto software. Queste dipendenze possono essere tra classi, pacchetti o persino tra l'intero progetto e le librerie esterne. L'analisi di tali dipendenze è cruciale per migliorare la gestione del codice, la manutenzione e l'identificazione di potenziali problematiche di performance o di qualità del software. Una gestione efficace delle dipendenze permette non solo di ottimizzare il codice esistente, ma anche di prevenire conflitti tra le versioni delle librerie e migliorare la modularità del progetto.
La libreria `DependencyAnalyser` è stata progettata con un focus particolare sull'efficienza e sulla scalabilità, utilizzando un'architettura asincrona che consente di gestire anche progetti di grandi dimensioni senza compromettere le performance. Grazie a questo approccio, è possibile analizzare grandi codebase in modo rapido e senza bloccare l'esecuzione dell'applicazione.

`DependencyAnalyser` è stato sviluppato come una libreria riutilizzabile e modulare, facilmente integrabile in qualsiasi progetto Java. È completamente compatibile con ambienti di sviluppo moderni, supportando sia l'analisi di progetti già esistenti che di progetti in fase di sviluppo. La libreria offre un'interfaccia semplice per l'analisi delle dipendenze e può essere utilizzata in vari contesti, dal debugging all'ottimizzazione della struttura di un progetto software.
Il progetto è stato realizzato utilizzando due principali paradigmi di programmazione: *asincrono* e *reattivo*. Inizialmente, viene implementato un approccio asincrono che sfrutta la libreria `Vert.x` per la gestione non bloccante delle operazioni, con l'obiettivo di migliorare le performance, soprattutto in scenari in cui è necessario elaborare un grande numero di dipendenze in parallelo. Successivamente, viene introdotto un approccio reattivo basato su `RxJava`, che consente una gestione ancora più fine delle risorse e delle operazioni asincrone attraverso l'uso di flussi osservabili. L'adozione di un'architettura reattiva permette di migliorare ulteriormente la scalabilità e la gestione del flusso di dati, adattandosi perfettamente alle esigenze di sistemi complessi. Tutte le dipendenze necessarie per i due paradigmi sono specificate all'interno del file `build.gradle`.

In entrambi i casi, l'output dell'analisi delle dipendenze verrà presentato attraverso un'interfaccia grafica, che permette di visualizzare facilmente i risultati sotto forma di grafici interattivi. Questa visualizzazione aiuta a comprendere le relazioni tra le classi e i pacchetti in modo intuitivo e immediato, facilitando l'interpretazione dei risultati e la pianificazione di interventi di manutenzione.
Il progetto, quindi, non solo fornisce un potente strumento per l'analisi delle dipendenze, ma offre anche una soluzione visuale per esplorare i dati, rendendo il processo di sviluppo più efficiente e accessibile. Nel corso dello sviluppo, l'integrazione di una GUI rappresenta un passo fondamentale per migliorare l'usabilità e offrire agli sviluppatori un modo pratico e interattivo per interagire con il sistema e ottenere feedback in tempo reale.

2 Obiettivi del Progetto

Gli obiettivi principali del progetto sono molteplici e coprono vari aspetti dell'analisi delle dipendenze:

- **Analizzare le dipendenze tra le classi**: La libreria è in grado di identificare tutte le classi di cui una determinata classe dipende, sia direttamente che indirettamente. Questo consente di avere una visione chiara delle relazioni tra i vari componenti di un progetto.

- **Analizzare le dipendenze tra i pacchetti**: Oltre alle dipendenze a livello di classe, il sistema è in grado di identificare anche le relazioni tra i pacchetti, fornendo una panoramica completa delle interconnessioni all'interno del progetto.

- **Generare report strutturati e visualizzazioni**: L'output principale del sistema sono dei report dettagliati che possono essere visualizzati come grafici o esportati in formati compatibili con strumenti di analisi e documentazione, come PDF o HTML. Nel nostro caso nella prima implementazione asincrona il report viene offerto mediante stampe a terminale mentre nella seconda implementazione ovvero quella reattiva il report delle dipendenze verrà offerto mediante una interfaccia grafica.

- **Ottimizzare l'analisi per progetti di grandi dimensioni**: Utilizzando `Vert.x` e un modello di esecuzione asincrono, la libreria è in grado di gestire progetti

con milioni di righe di codice senza compromettere le performance o la reattività dell'applicazione.

3 Analisi del Problema

Il problema affrontato riguarda la necessità di analizzare le dipendenze tra i vari componenti di un progetto software complesso. Con l'evoluzione dei progetti, le dipendenze tra le classi e i pacchetti diventano sempre più intricate, creando difficoltà nella gestione e manutenzione del codice. Un altro problema significativo riguarda l'analisi dei progetti di grandi dimensioni, dove la gestione di milioni di righe di codice può causare rallentamenti nelle operazioni di analisi.
La soluzione proposta, come rischiesto dall'assignment, riguarda una con la programmazione asincrona e poi successivamente secondo la programmazione reattiva. Nella sezione seguente andremo a descrivere in dettaglio il design e l'architettura delle due soluzione proposte.

4 Descrizione del Design e dell'Architettura

4.1 Parte Asincrona

La libreria DependecyAnalyserLib è progettata per fornire metodi asincroni che analizzano le dipendenze a livello di classe, pacchetto e progetto. Questi metodi devono essere basati su un approccio asincrono, utilizzando un framework come Vert.x per gestire l'event-loop.
La libreria `DependencyAnalyserLib` è il cuore dell'approccio asincrono del nostro progetto. Essa è stata progettata con l'obiettivo di offrire metodi per l'analisi delle dipendenze tra le classi, i pacchetti e l'intero progetto. L'architettura della libreria si basa sull'utilizzo di un framework come Vert.x, che gestisce l'esecuzione asincrona tramite un event-loop. In pratica, Vert.x ci consente di eseguire operazioni in parallelo senza bloccare il flusso principale dell'applicazione, garantendo una maggiore efficienza, soprattutto quando si tratta di analizzare progetti di grandi dimensioni.
La libreria si articola in tre metodi principali, ognuno dei quali ha una responsabilità specifica. Il primo, `getClassDependencies(Path)`, si occupa di analizzare un singolo file sorgente Java, estraendo tutte le dipendenze che la classe definita all'interno del file ha

con altre classi o pacchetti. Il secondo metodo, `getPackageDependencies(Path)`, è pensato per analizzare ricorsivamente tutti i file `.java` presenti in una determinata cartella, restituendo un report che aggrega tutte le dipendenze a livello di pacchetto.
Infine, `getProjectDependencies(Path)` estende questa logica a tutto un progetto, esaminando anche le sottocartelle contenenti file sorgente e generando un report a livello di progetto. Questi metodi sono alla base del nostro strumento, e ci permettono di raccogliere informazioni cruciali per l'analisi delle dipendenze nel progetto.
Una delle caratteristiche principali della libreria è la gestione asincrona delle operazioni di lettura e analisi semantica. Utilizzando il costrutto `CompositeFuture` di Vert.x, siamo in grado di orchestrare più operazioni in parallelo, migliorando notevolmente le prestazioni rispetto a un approccio sincrono. Ogni operazione di lettura file e analisi semantica è trattata come una `Promise`, che può essere risolta o fallita. Se si verifica un errore durante il parsing di un file o la lettura di un contenuto, l'errore viene propagato attraverso la `Promise.fail()`, informando il sistema dell'errore e permettendo una gestione adeguata.

Per quanto riguarda l'analisi semantica, la libreria sfrutta la potente libreria `JavaParser` per visitare l'albero di sintassi astratta (AST) di ciascuna classe. La classe `DependencyVisitor` si occupa di identificare e raccogliere tutte le dipendenze presenti nel codice sorgente. Le dipendenze vengono classificate in diverse categorie: le importazioni di classi esterne, le relazioni di ereditarietà e implementazione, i riferimenti a classi nei campi, parametri e tipi di ritorno, nonché le istanziazioni di oggetti tramite la keyword `new`. Questi dati vengono raccolti e rappresentati in oggetti di tipo `AsyncUtils`, che contengono informazioni dettagliate sulle dipendenze, come il tipo di dipendenza, la riga di origine nel codice e una breve descrizione.
L'esecuzione dell'analisi avviene attraverso la classe `DependencyAnalyserVerticle`, che funge da punto di ingresso per l'intero processo. Una volta avviata, questa classe effettua l'analisi incrementale, generando e stampando i report delle dipendenze a livello di classe, pacchetto e progetto direttamente sulla console. Questo processo dimostra come sia semplice, con l'ausilio di Vert.x, orchestrare analisi complesse come quella semantica, pur mantenendo un elevato grado di efficienza e reattività.
L'approccio asincrono porta con sé numerosi benefici. In primo luogo, consente di eseguire l'analisi di più file e pacchetti contemporaneamente, sfruttando appieno le risorse

della macchina senza bloccare l'esecuzione del programma. Inoltre, l'utilizzo di un modello asincrono garantisce una maggiore reattività dell'applicazione, riducendo al minimo i tempi di attesa per l'utente finale. Infine, l'approccio asincrono è particolarmente vantaggioso quando si lavora con progetti di grandi dimensioni, dove la scalabilità è un requisito fondamentale. La capacità di gestire il carico di lavoro in modo efficiente senza compromettere le prestazioni è uno degli aspetti chiave che differenzia questa soluzione rispetto a implementazioni sincrone.

4.1.1 Struttura generale

Il componente principale è la classe `DependencyAnalyserLib`, la quale fornisce tre metodi pubblici:

- `getClassDependencies(Path)`: analizza un singolo file sorgente Java e restituisce le dipendenze della classe contenuta.
- `getPackageDependencies(Path)`: analizza ricorsivamente tutti i file `.java` in una cartella, aggregando le dipendenze in un `PackageDepsReport`.
- `getProjectDependencies(Path)`: estende il concetto di analisi a tutte le sottocartelle contenenti file sorgente, generando un `ProjectDepsReport`.

4.1.2 Meccanismo asincrono

Tutte le operazioni di lettura file e analisi semantica vengono gestite in modo asincrono, componendo i risultati con `CompositeFuture`. In caso di errori durante il parsing o la lettura dei file, viene propagato un errore al chiamante tramite `Promise.fail()`.

4.1.3 Analisi semantica con JavaParser

La classe `DependencyVisitor` sfrutta la libreria `JavaParser` per visitare l'albero di sintassi astratta (AST) di ciascuna classe. Le dipendenze rilevate includono:

- **Importazioni**: classi importate esplicitamente.
- **Ereditarietà e implementazione**: classi o interfacce estese/implementate.
- **Campi, parametri, tipi di ritorno**: riferimenti ad altre classi nei membri della classe.

- **Istanziazioni**: utilizzo di `new` per creare oggetti.

Ogni dipendenza rilevata viene rappresentata tramite la classe `AsyncUtils`, contenente informazioni quali tipo di dipendenza, riga di origine e descrizione.

4.1.4 Esecuzione dell'analisi

La classe `DependencyAnalyserVerticle` funge da punto d'ingresso per l'analisi. Una volta avviata tramite Vert.x, la classe effettua un'analisi incrementale, producendo l'output dei tre livelli (classe, pacchetto, progetto) direttamente sulla console. Ciò dimostra la facilità di orchestrazione asincrona di Vert.x anche per compiti strutturati come l'analisi semantica del codice.

4.1.5 Benefici dell'approccio asincrono

L'uso della programmazione asincrona consente:

- Analisi concorrente di più file e pacchetti.
- Maggiore reattività dell'applicazione rispetto a soluzioni sincrone.
- Miglioramento della scalabilità in progetti di grandi dimensioni.

4.2 Parte Reattiva

La parte reattiva del progetto, denominata **Reactive Dependency Analyser**, rappresenta una delle componenti più interessanti e innovative. L'obiettivo di questa parte è quello di offrire un'analisi in tempo reale delle dipendenze tra le classi e i pacchetti di un progetto Java. Per raggiungere questo scopo, l'applicazione sfrutta le potenzialità di **RxJava**, una libreria per la programmazione reattiva, e **Java Swing** per la realizzazione dell'interfaccia grafica utente (GUI). In sostanza, l'applicazione permette all'utente di visualizzare graficamente le dipendenze tra i file sorgente di un progetto in modo dinamico, aggiornando in tempo reale la vista ogni volta che vengono apportate modifiche ai file sorgente.
L'obiettivo principale della parte reattiva è quello di creare uno strumento semplice, ma potente, che permetta di identificare e visualizzare le dipendenze tra le varie entità di un

progetto Java. In particolare, l'applicazione deve essere in grado di identificare le dipendenze tra classi, interfacce e pacchetti e rappresentarle graficamente in un grafo. Inoltre, la visualizzazione deve essere aggiornata automaticamente ogni volta che un file sorgente viene modificato, offrendo all'utente una rappresentazione sempre aggiornata dello stato del progetto. In questo modo, il sistema diventa uno strumento utile per la comprensione delle interazioni tra i vari componenti di un progetto Java e per la gestione delle sue dipendenze.
Il sistema è composto da tre componenti principali;

- Il primo è la classe `ReactiveDependencyAnalyser`, che si occupa dell'analisi dei file Java. Utilizzando i flussi osservabili (observable streams) offerti da **RxJava**, questa classe è in grado di eseguire l'analisi in modo asincrono e reattivo. In particolare, la classe osserva le modifiche ai file sorgente e, quando vengono rilevate modifiche, esegue nuovamente l'analisi delle dipendenze e aggiorna la visualizzazione.

- Il secondo componente è `GraphPanel`, un pannello Swing che si occupa della visualizzazione grafica delle dipendenze. Ogni nodo del grafo rappresenta una classe o un pacchetto, mentre gli archi tra i nodi indicano le relazioni di dipendenza.

- Il terzo componente è la classe `MainGUI`, che gestisce l'interfaccia utente e coordina l'interazione tra l'analisi delle dipendenze e la visualizzazione grafica.

Il flusso dei dati all'interno del sistema è basato su **Observable** di RxJava, che consente un'elaborazione reattiva e asincrona delle modifiche ai file. Ogni volta che un file viene modificato, l'analisi viene eseguita nuovamente e il grafo viene aggiornato di conseguenza. Questo approccio permette di gestire efficacemente le modifiche in tempo reale, senza la necessità di ricaricare l'intero progetto o di eseguire manualmente l'analisi.

Nel sistema vengono utilizzate diverse tecnologie. Il linguaggio principale utilizzato è **Java 17**, che fornisce le funzionalità necessarie per la gestione delle dipendenze e l'interfaccia grafica. Per la gestione dei flussi reattivi, si fa ampio uso di **RxJava**, che permette di creare flussi di dati osservabili e di gestire le modifiche in modo asincrono. La GUI è realizzata con **Java Swing**, una libreria ben consolidata per la creazione di interfacce utente in Java. In aggiunta, potrebbe essere utilizzata **JGraphT** per la modellazione del grafo, anche se non è stato implementato direttamente nel codice attuale.

Il funzionamento dell'applicazione è semplice ma efficace. L'utente può selezionare una cartella contenente i file sorgente Java, e il sistema procederà ad analizzarli per estrarre le informazioni rilevanti: nome del pacchetto, classi e interfacce definite, e dipendenze tramite le dichiarazioni `import`. Queste informazioni vengono quindi passate alla componente grafica, che visualizza il grafo delle dipendenze. La visualizzazione è dinamica e viene aggiornata ogni volta che vengono apportate modifiche ai file sorgente.
Per quanto riguarda il parsing e la gestione delle dipendenze, vengono utilizzate espressioni regolari per identificare i pacchetti, le classi e le interfacce all'interno dei file. Le dipendenze vengono rilevate tramite la keyword `import`, che indica i riferimenti ad altre classi o pacchetti. Una volta raccolte, queste informazioni vengono inserite in una mappa delle dipendenze, che rappresenta le relazioni tra i nodi del grafo.
Per concludere, la visualizzazione grafica delle dipendenze avviene all'interno del pannello `GraphPanel`. Ogni nodo del grafo rappresenta una classe o un pacchetto, mentre gli archi rappresentano le relazioni di dipendenza tra di essi. I nodi dei pacchetti vengono evidenziati con una colorazione differente per distinguerli dalle classi. Il layout del grafo è calcolato automaticamente, distribuendo i nodi in modo circolare lungo una griglia, il che rende la visualizzazione chiara e facilmente comprensibile.

4.2.1 Obiettivi parte reattiva

L'obiettivo principale 'e realizzare uno strumento semplice e reattivo che, dato un insieme di file sorgente Java, permetta di:

- Identificare le dipendenze tra classi, interfacce e pacchetti.
- Visualizzare le dipendenze in una struttura a grafo.
- Aggiornare dinamicamente il grafo in caso di modifiche ai file.

4.2.2 Architettura del sistema reattiva

Il sistema 'e suddiviso in tre componenti principali:

- `ReactiveDependencyAnalyser`: si occupa di eseguire l'analisi dei file usando flussi osservabili.

- `GraphPanel`: pannello Swing responsabile della visualizzazione grafica delle dipendenze.
- `Main`: gestisce l'interfaccia utente e l'integrazione tra analisi e visualizzazione.

Il flusso dati 'e basato su **Observable** di RxJava, che consente un'elaborazione asincrona e fluida dell'input.

4.2.3 Tecnologie utilizzate

- **Java 17**: linguaggio principale.
- **RxJava**: gestione reattiva dei flussi di dati.
- **Java Swing**: realizzazione della GUI.
- **JGraphT** (opzionale): per la modellazione del grafo (non utilizzato direttamente nel codice attuale).

4.2.4 Funzionamento del sistema

L'applicazione permette all'utente di selezionare una cartella contenente file sorgente Java. Ogni file viene letto e analizzato per estrarre:

- Nome del pacchetto.
- Classi e interfacce definite.
- Dipendenze `import` verso altre classi o pacchetti.

Queste informazioni vengono strutturate e passate al pannello grafico per la visualizzazione.

4.2.5 Parsing e gestione delle dipendenze

Il parsing dei file Java avviene tramite espressioni regolari. In particolare:

- I pacchetti vengono rilevati con pattern su `package`.
- Le classi e interfacce sono identificate tramite keyword `class` o `interface`.
- Le dipendenze sono dedotte dagli `import`.

I dati raccolti sono inseriti in una mappa delle dipendenze tra nodi (classi o pacchetti).

4.2.6 Visualizzazione grafica (GUI)

Il grafo viene visualizzato all'interno del pannello `GraphPanel` tramite primitive grafiche Swing:

- Ogni nodo rappresenta una classe o un pacchetto.
- Gli archi rappresentano le relazioni di dipendenza.
- I nodi dei pacchetti sono evidenziati con una colorazione differente.

Il layout dei nodi 'e calcolato automaticamente distribuendo i vertici lungo una griglia circolare.

5 Rappresentazione tramite Reti di Petri

5.1 Parte asincrona

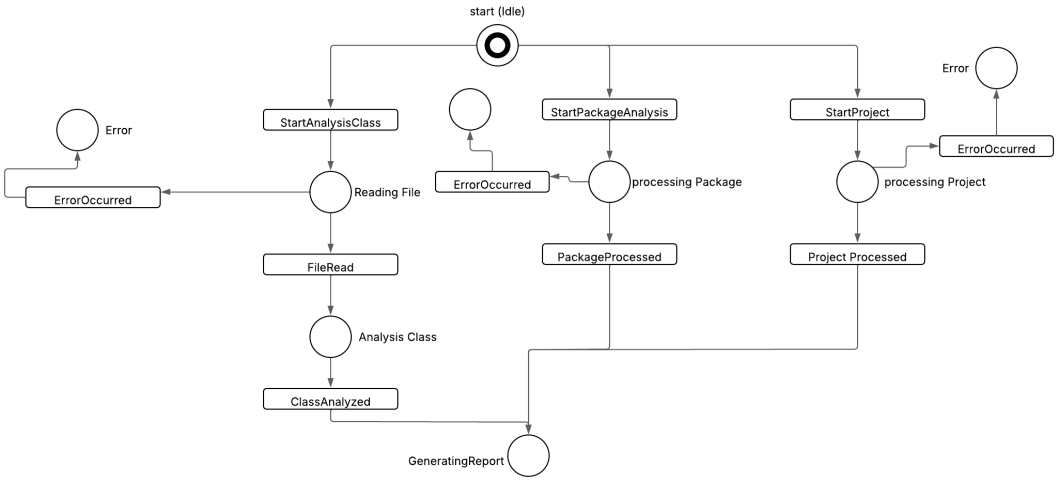

Figure 1: Modello di Rete di Petri per la parte asincrona

Luoghi (Places)

- **Idle**: Stato iniziale, in attesa che venga avviata un'analisi.
- **ReadingFile**: Stato in cui un file Java viene letto.
- **AnalyzingClass**: Stato in cui il codice sorgente di una classe viene analizzato.

- **ProcessingPackage**: Stato in cui i file di un pacchetto vengono elaborati.
- **ProcessingProject**: Stato in cui i pacchetti di un progetto vengono elaborati.
- **GeneratingReport**: Stato in cui viene generato un report (classe, pacchetto o progetto).
- **Error**: Stato in cui si verifica un errore durante l'analisi.

Transizioni

- **StartClassAnalysis**: Avvio dell'analisi di una singola classe.
- **StartPackageAnalysis**: Avvio dell'analisi di un pacchetto.
- **StartProjectAnalysis**: Avvio dell'analisi di un progetto.
- **FileRead**: Completamento della lettura di un file.
- **ClassAnalyzed**: Completamento dell'analisi di una classe.
- **PackageProcessed**: Completamento dell'elaborazione di un pacchetto.
- **ProjectProcessed**: Completamento dell'elaborazione di un progetto.
- **ErrorOccurred**: Si verifica un errore durante una delle operazioni.

Descrizione del Comportamento

- **Analisi di una classe**:
  - La transizione **StartClassAnalysis** sposta il sistema dallo stato *Idle* a *ReadingFile*.
  - Una volta letto il file (**FileRead**), il sistema passa a *AnalyzingClass*.
  - Dopo l'analisi della classe (**ClassAnalyzed**), il sistema genera un report nello stato *GeneratingReport*.
- **Analisi di un pacchetto**:

  - La transizione **StartPackageAnalysis** sposta il sistema dallo stato *Idle* a *ProcessingPackage*.
  - Durante l'elaborazione del pacchetto, vengono analizzati i file Java al suo interno.
  - Una volta completata l'elaborazione del pacchetto (**PackageProcessed**), il sistema genera un report nello stato *GeneratingReport*.

- **Analisi di un progetto**:
  - La transizione **StartProjectAnalysis** sposta il sistema dallo stato *Idle* a *ProcessingProject*.
  - Durante l'elaborazione del progetto, vengono analizzati i pacchetti al suo interno.
  - Una volta completata l'elaborazione del progetto (**ProjectProcessed**), il sistema genera un report nello stato *GeneratingReport*.

- **Gestione degli errori**:
  - Se si verifica un errore durante la lettura di un file, l'analisi di una classe, l'elaborazione di un pacchetto o di un progetto, il sistema passa allo stato *Error* tramite la transizione **ErrorOccurred**.

5.2 parte Reattiva

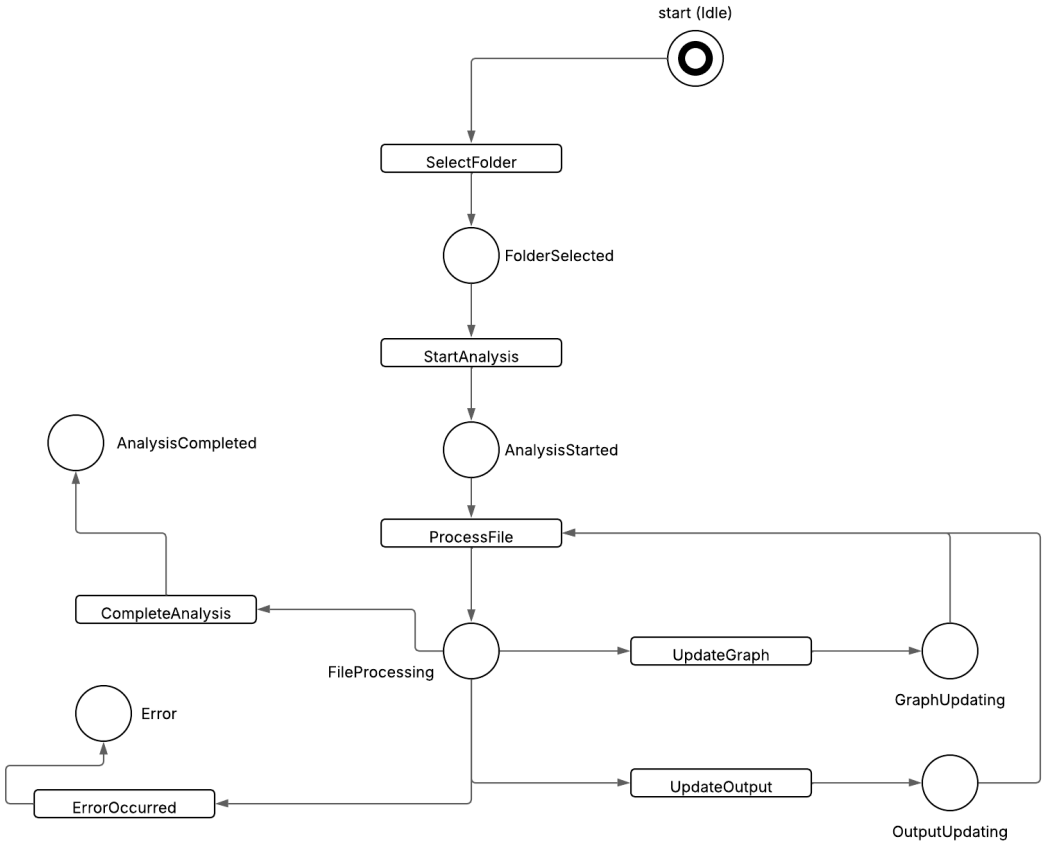

Figure 2: Modello di Rete di Petri per la parte Reattiva

Descrizione della rete di Petri

Luoghi (Stati)

- **Idle**: Stato iniziale, in attesa che l'utente interagisca con l'interfaccia.
- **FolderSelected**: Stato in cui l'utente ha selezionato una cartella sorgente.
- **AnalysisStarted**: Stato in cui l'analisi delle dipendenze è stata avviata.
- **FileProcessing**: Stato in cui i file Java vengono analizzati.
- **GraphUpdating**: Stato in cui il grafo delle dipendenze viene aggiornato.

- **OutputUpdating**: Stato in cui il pannello di output viene aggiornato con i risultati.
- **AnalysisCompleted**: Stato in cui l'analisi è completata.
- **Error**: Stato in cui si verifica un errore durante l'analisi.

Transizioni

- **SelectFolder**: L'utente seleziona una cartella sorgente tramite il *SourceSelector*.
- **StartAnalysis**: L'utente preme il pulsante *"Analyze"* per avviare l'analisi.
- **ProcessFile**: Un file Java viene analizzato.
- **UpdateGraph**: Il grafo delle dipendenze viene aggiornato con i nuovi nodi e archi.
- **UpdateOutput**: Il pannello di output viene aggiornato con i risultati dell'analisi.
- **CompleteAnalysis**: L'analisi è completata.
- **ErrorOccurred**: Si verifica un errore durante una delle operazioni.

Descrizione del comportamento

- **Selezione della cartella:**
  - L'utente seleziona una cartella sorgente tramite il componente *SourceSelector*.
  - Lo stato passa da **Idle** a **FolderSelected**.
- **Avvio dell'analisi:**
  - L'utente preme il pulsante *"Analyze"* nel pannello *DependencyAnalyserPanel*.
  - Lo stato passa da **FolderSelected** a **AnalysisStarted**.
- **Elaborazione dei file:**
  - Ogni file Java nella cartella selezionata viene analizzato.
  - Lo stato passa da **AnalysisStarted** a **FileProcessing**.
- **Aggiornamento del grafo:**

  - Dopo l'analisi di un file, il grafo delle dipendenze viene aggiornato.
  - Lo stato passa da **FileProcessing** a **GraphUpdating**.

- **Aggiornamento dell'output:**
  - I risultati dell'analisi vengono mostrati nel pannello di output.
  - Lo stato passa da **FileProcessing** a **OutputUpdating**.

- **Completamento dell'analisi:**
  - Quando tutti i file sono stati analizzati, lo stato passa da **FileProcessing** a **AnalysisCompleted**.

- **Gestione degli errori:**
  - Se si verifica un errore durante una delle operazioni, lo stato passa a **Error**.

6 Conclusioni

Il progetto **Find the Dependencies** ha rappresentato un'esperienza significativa di integrazione tra due paradigmi fondamentali per lo sviluppo di applicazioni moderne: la programmazione **asincrona** e quella **reattiva**. Entrambe le componenti sono state progettate con l'obiettivo di offrire un sistema capace di analizzare strutturalmente progetti Java e rappresentarne visivamente le dipendenze tra classi e pacchetti, fornendo così uno strumento utile sia a fini didattici che pratici.
Dal lato asincrono, l'implementazione della libreria `DependencyAnalyserLib` ha permesso di delegare le operazioni più onerose — come la lettura e l'analisi dei file sorgente Java — a un meccanismo basato su `Vert.x`, uno dei framework più diffusi nel mondo reactive per la gestione di eventi. La progettazione modulare della libreria, articolata attorno ai metodi `getClassDependencies`, `getPackageDependencies` e `getProjectDependencies`, ha reso possibile un'elaborazione concorrente ed efficiente di interi progetti, anche di grandi dimensioni. La logica asincrona, supportata da `Promise` e `CompositeFuture`, ha garantito una gestione fluida e reattiva degli errori e dei risultati intermedi. L'utilizzo della libreria `JavaParser` ha consentito un'analisi semantica precisa, rilevando tipologie di dipendenze che spaziano dalle importazioni fino all'ereditarietà e all'utilizzo di istanze.

Parallelamente, la parte reattiva del progetto ha introdotto una GUI interattiva sviluppata in `Java Swing` e supportata dalla libreria `RxJava`. Tale componente ha reso il sistema accessibile anche a utenti non tecnici, permettendo di visualizzare in modo intuitivo il grafo delle dipendenze tra i file analizzati. La classe `ReactiveDependencyAnalyser` si occupa di trasformare il flusso di dati in eventi osservabili, che vengono poi consumati e visualizzati nel pannello grafico `GraphPanel`. L'utente può così selezionare una directory contenente codice sorgente Java e osservare in tempo reale la struttura di dipendenza emergente, grazie a un layout automatico dei nodi e a una rappresentazione differenziata tra classi e pacchetti.
Questo progetto ha raggiunto i seguenti obiettivi principali:

- Sviluppare una libreria asincrona robusta e riutilizzabile per l'analisi di dipendenze Java.
- Implementare un'applicazione desktop reattiva che consenta la visualizzazione dinamica delle dipendenze.
- Integrare efficacemente due paradigmi di programmazione moderni in un contesto applicativo reale.

Finendo il discorso inziato in precendenza, **Find the Dependencies** si è dimostrato un progetto completo, in grado di coniugare l'efficienza computazionale della programmazione asincrona con la reattività e l'usabilità di un'interfaccia grafica moderna. La sua architettura modulare, la chiarezza dei flussi di dati e l'efficace separazione delle responsabilità lo rendono un'ottima base per progetti più ambiziosi nel campo dell'analisi del codice e della visualizzazione interattiva.In un progetto futuro l'idea sarebbe combinare le due idee , ovvero reattiva ed asincrona per vedere i risultati che si ottengono e confrontarli con quelli attuali.