Assignment02

Giacomo Foschi
Matricola: 0001179137
Email: giacomo.foschi3@studio.unibo.it

Giovanni Pisoni
Matricola: 0001189814
Email: giovanni.pisoni@studio.unibo.it

Giovanni Rinchiuso
Matricola: 0001195145
Email: giovanni.rinchiuso@studio.unibo.it

Gioele Santi
Matricola: 0001189403
Email: gioele.santi2@studio.unibo.it

Capitolo 1 - Analisi del problema

L'obiettivo generale di questo assignment è analizzare le dipendenze presenti all'interno di un progetto Java, ovvero identificare quali classi, interfacce e package sono utilizzati o acceduti da ciascuna classe, interfaccia o pacchetto del progetto.
L'analisi si basa sul parsing dei file sorgente Java e sull'ispezione dell'Abstract Syntax Tree (AST) costruito per ciascun file, in modo da tracciare l'utilizzo dei tipi definiti e importati.

In particolare, è richiesto lo sviluppo di due soluzioni differenti:

- Una libreria per l'analisi asincrona delle dipendenze i cui report vengono stampati sul terminale
- Un programma con interfaccia grafica per analizzare e visualizzare in tempo reale le dipendenze come un grafo, sfruttando la programmazione reattiva

Programmazione asincrona

Il problema da risolvere consiste nello sviluppo di una libreria asincrona (DependencyAnalyserLib) per analizzare le dipendenze in un progetto Java. L'obiettivo è identificare le classi, interfacce e pacchetti utilizzati da ogni classe, interfaccia o pacchetto del progetto. La libreria deve fornire metodi asincroni per:

- Analisi delle dipendenze di una classe (ClassDepsReport): restituisce un report con le dipendenze di una singola classe o interfaccia.
- Analisi delle dipendenze di un pacchetto (PackageDepsReport): restituisce un report con le dipendenze di tutte le classi e interfacce in un pacchetto.
- Analisi delle dipendenze di un progetto (ProjectDepsReport): restituisce un report con le dipendenze di tutte le classi e interfacce in tutti i pacchetti del progetto.

La libreria deve essere basata su programmazione asincrona utilizzando il framework Vert.x, basato su event-loop. Inoltre, per il parsing dei file sorgente Java, si utilizza la libreria JavaParser, che costruisce un AST (Abstract Syntax Tree) per analizzare il codice.
Le principali sfide da affrontare nello svolgimento di questo progetto sono state:

- Risoluzione dei tipi con JavaParser: JavaParser necessita di risolvere correttamente i tipi per identificare le dipendenze, ma questo può essere complesso in presenza di classi annidate, import con wildcard o tipi generici.
- Gestione delle operazioni asincrone annidate: L'analisi di un progetto richiede operazioni asincrone annidate: il progetto contiene pacchetti, che contengono classi, ognuna da analizzare in modo asincrono.
- Gestione degli errori nelle operazioni sui file: Le operazioni di I/O sui file possono fallire per vari motivi (file non trovati, permessi insufficienti, ecc.).
- Filtro delle dipendenze rilevanti: Identificare quali tipi sono effettivamente rilevanti per l'analisi, escludendo dipendenze non significative come tipi primitivi o classi standard di Java.
- Prestazioni su progetti di grandi dimensioni: L'analisi di progetti con molti file può richiedere tempo e risorse considerevoli.

Programmazione reattiva

L'obiettivo del secondo punto dell'assignment era la progettazione e lo sviluppo di un'applicazione GUI, denominata DependencyAnalyser, in grado di analizzare e visualizzare dinamicamente le dipendenze tra le classi e le interfacce di un progetto Java.
Il progetto richiede l'adozione di un approccio di programmazione reattiva per gestire l'intero processo di analisi e visualizzazione.

L'applicazione doveva offrire all'utente le seguenti funzionalità:

- Selezione della cartella sorgente del progetto Java da analizzare.
- Avvio del processo di analisi, con l'estrazione delle dipendenze tra classi e interfacce.
- Visualizzazione incrementale dei risultati sotto forma di grafo, rappresentando le classi, le interfacce e i pacchetti, nonché le loro dipendenze.
- Visualizzazione di statistiche in tempo reale tramite due riquadri, che riportano rispettivamente il numero di classi/interfacce analizzate ed il numero di dipendenze trovate.

L'utilizzo del framework ReactiveX (con RxJava per l'ambiente Java) era suggerito per implementare i flussi di dati reattivi. Inoltre, era consentito l'utilizzo della libreria JavaParser per effettuare il parsing dei file Java e costruire l'AST necessario per l'estrazione delle dipendenze.

Nell'analisi del problema sono emerse diverse sfide progettuali e implementative:

- **Gestione di grandi volumi di dati**: Un progetto Java può contenere centinaia o migliaia di file. Elaborarli tutti in modo sincrono bloccherebbe l'interfaccia utente, rendendo l'applicazione non reattiva. È necessario un approccio che permetta di processare i file in modo asincrono, aggiornando l'interfaccia man mano che i risultati diventano disponibili.
- **Visualizzazione dinamica delle dipendenze**: Le dipendenze tra classi devono essere visualizzate in tempo reale sotto forma di grafo, con nodi che rappresentano le classi e archi che rappresentano le relazioni. Questo richiede una gestione efficiente del grafo e un meccanismo per aggiornare la visualizzazione senza bloccare l'interfaccia utente.
- **Esclusione delle dipendenze irrilevanti**: È importante filtrare le dipendenze verso classi appartenenti a package standard di Java (come java.lang o java.util), poiché queste non forniscono informazioni utili sull'architettura del progetto analizzato.
- **Gestione degli errori**: Durante l'analisi, possono verificarsi errori (ad esempio, file corrotti o inaccessibili). Il sistema deve essere in grado di gestire questi errori senza interrompere l'intero processo, segnalandoli all'utente in modo chiaro.
- **Reattività dell'Interfaccia**: L'interfaccia utente deve rimanere reattiva durante l'analisi, permettendo all'utente di interagire con il sistema (ad esempio, zoomare o spostarsi nel grafo) anche mentre i dati vengono elaborati.

Capitolo 2 - Strategia risolutiva e architettura

Programmazione asincrona

Il sistema è stato progettato secondo una gerarchia a tre livelli, che consente di aggregare progressivamente le informazioni sulle dipendenze tra i componenti di un progetto Java. Il livello più basso è rappresentato dalla singola classe: per ciascuna classe o interfaccia, viene generato un report (ClassDepsReport) che raccoglie tutte le dipendenze di tipo utilizzate al suo interno. Queste dipendenze comprendono, ad esempio, i tipi dei campi, i parametri dei metodi, le classi estese o implementate e le istanze create. I report relativi alle singole classi vengono successivamente combinati per ottenere un report a livello di pacchetto (PackageDepsReport), che aggrega le informazioni per tutte le classi contenute in uno stesso package. Infine, i report di tutti i pacchetti vengono raccolti in un report complessivo a livello di progetto (ProjectDepsReport). Questa struttura gerarchica garantisce una rappresentazione chiara, modulare e scalabile della rete di dipendenze, favorendo l'analisi architetturale a diversi livelli di granularità.

Programmazione asincrona e gestione delle operazioni I/O

Per garantire efficienza e reattività anche in presenza di operazioni I/O-intensive, il sistema adotta il framework Vert.x, che consente di sviluppare applicazioni non bloccanti e altamente concorrenti.
Tutti i metodi che eseguono l'analisi delle dipendenze, sia per singole classi, pacchetti o l'intero progetto, sono progettati per restituire oggetti di tipo Future (io.vertx.core.Future). Questo approccio permette di eseguire le operazioni in maniera asincrona, evitando di bloccare il thread principale.

L'implementazione di ciascun metodo prevede la creazione e il completamento esplicito di una Promise (io.vertx.core.Promise), da cui viene derivato e restituito il relativo Future. In questo modo, il risultato dell'operazione asincrona può essere completato (con successo o con errore) una volta terminata l'elaborazione.

Le operazioni più costose dal punto di vista computazionale (come la lettura dei file e il parsing del codice sorgente) sono eseguite utilizzando il costrutto executeBlocking di Vert.x. Questo consente di spostare le operazioni bloccanti su un pool di thread dedicato, evitando di sovraccaricare l'event loop principale, che rimane libero per gestire altre richieste e mantenere alta la reattività del sistema.

Per coordinare l'esecuzione di analisi asincrone annidate è utilizzato il costrutto CompositeFuture.all(). Questo meccanismo permette di aggregare e sincronizzare più Future in parallelo, garantendo che il livello superiore attenda il completamento di tutte le operazioni dei livelli inferiori prima di procedere. In caso di fallimento di uno dei task, l'intero CompositeFuture viene marcato come fallito, e il fallimento viene propagato correttamente per consentire una gestione robusta degli errori.

Parsing del codice sorgente e filtraggio delle dipendenze

L'analisi sintattica del codice sorgente è stata realizzata utilizzando la libreria JavaParser, che consente di ottenere l'albero sintattico astratto (AST) a partire dai file sorgente Java. L'AST viene visitato tramite un visitor personalizzato, denominato DependencyVisitor, progettato per estrarre le dipendenze tra classi e interfacce rilevanti dal punto di vista applicativo.

Il visitor analizza in modo sistematico diversi elementi: estensioni di classi, implementazioni di interfacce, dichiarazioni di campi di tipo oggetto, tipi restituiti dai metodi, parametri dei metodi e istanze create con l'operatore new.
Per garantire una risoluzione robusta dei tipi anche in presenza di classi annidate, import con wildcard o tipi generici, è stato configurato un parser ad hoc nella classe ad ParserConfigurator. Questo integra un ReflectionTypeSolver per risolvere le classi standard di Java, un ClassLoaderTypeSolver per le dipendenze importate tramite gradle o maven, e un JavaParserTypeSolver per risolvere i tipi definiti nel progetto analizzato.
Inoltre, per garantire che i report mettano in evidenza solo le dipendenze significative, il parser applica un filtro: vengono escluse automaticamente le dipendenze verso tipi primitivi, array e classi appartenenti alla libreria standard Java. Questo filtro si basa sia su un'analisi sintattica sia su una risoluzione semantica ottenuta grazie a JavaSymbolSolver. Solo le dipendenze che superano questi criteri vengono registrate nei report, corredate da informazioni dettagliate quali il tipo di dipendenza, il codice sorgente rilevante e il numero di riga.
Grazie a questa strategia, i report prodotti offrono una visione mirata e priva di rumore delle relazioni tra le classi applicative, facilitando la comprensione della struttura e delle interazioni del progetto.

Gestione degli errori

Il sistema è stato progettato per gestire in modo robusto i casi critici più frequenti, evitando che errori locali compromettano l'intero processo di analisi.
Per gestire le operazioni di I/O in modo sicuro, sono stati utilizzati i meccanismi Promise.fail() e onFailure() di Vert.x: eventuali errori (come file non trovati o permessi insufficienti) vengono catturati e segnalati in modo esplicito senza interrompere l'intero processo. Se durante il parsing si verificano eccezioni (ad esempio ParseProblemException), le righe problematiche vengono loggate e il file viene ignorato senza bloccare l'elaborazione complessiva.
Nel caso in cui un file sia assente o non leggibile, viene restituito un errore esplicito tramite una Promise fallita, accompagnata da un messaggio informativo. Infine, qualora una directory risultasse vuota, il sistema restituisce semplicemente un report vuoto, senza sollevare eccezioni.

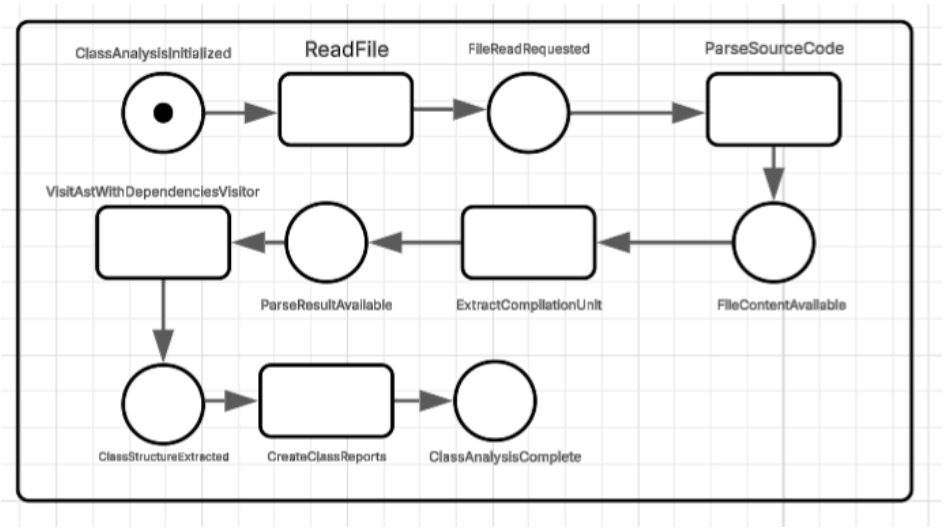
Collaudo end-to-end

Per verificare la correttezza e la solidità del sistema in un contesto realistico, è stato sviluppato un collaudo end-to-end tramite la classe SimulationAnalyser. Questo componente avvia un Verticle dedicato (DependencyAnalyserVerticle), il quale inietta percorsi di test a livello di classe, pacchetto e progetto. Vengono poi eseguiti i tre metodi asincroni di analisi e i report generati vengono stampati in console. Questo test dimostra la correttezza dei meccanismi di aggregazione, la validità della struttura gerarchica dei report e la tenuta del modello asincrono durante l'elaborazione di un progetto reale.

Petri’s net

Nella Petri’s net sottostante è rappresentato il sistema generale. In particolare si può vedere la sua struttura asincrona, infatti l’analisi della classe, del package e del progetto vengono eseguite in modo asincrono e prima di proseguire alla stampa dei report si aspetta che tutte e tre abbiano completato.

La Petri’s net di seguito riportata rappresenta un focus su quello che accade fra ClassAnalysisInitialized e ClassAnalysisComplete.



Programmazione reattiva

Architettura del sistema

Il DependencyAnalyser è stato progettato seguendo il pattern architetturale Model-View-Controller (MVC), che ci ha permesso di separare chiaramente la logica di analisi delle dipendenze (Model), l'interfaccia utente per la visualizzazione dei risultati (View) e la logica di coordinamento tra questi due componenti (Controller).

- **Model**: rappresentato principalmente dalla classe ReactiveDependencyAnalyser, che incapsula la logica di analisi delle dipendenze tra classi e interfacce. Le sue responsabilità principali sono:
    - Individuare ricorsivamente tutti i file sorgente Java all'interno di una directory di progetto.
    - Eseguire il parsing di ciascun file Java per estrarne l'Abstract Syntax Tree (AST)
    - Analizzare l'AST per identificare le dipendenze tra classi e interfacce
    - Filtrare le dipendenze non rilevanti (come le classi appartenenti a package standard di Java)
    - Emettere i risultati dell'analisi sotto forma di oggetti ClassDependency

  Questa componente è stata progettata in modo da essere completamente reattiva. Anziché restituire risultati sincroni, emette stream di dati che possono essere osservati e manipolati dal Controller.

- **View**: Il componente View è costituito principalmente dalle classi AnalysisView e GraphView:
    - AnalysisView: gestisce l'interfaccia utente principale, inclusi i controlli per la selezione della cartella di progetto, l'avvio dell'analisi, la visualizzazione dei log e delle statistiche
    - GraphView: si occupa specificamente della visualizzazione del grafo delle dipendenze, incapsulando la logica di rendering e interazione con la libreria GraphStream

  Le View sono state progettate per essere aggiornate in modo incrementale e reattivo, riflettendo in tempo reale i cambiamenti nei dati senza bloccare l'interfaccia utente.

- **Controller**: La classe AnalysisController funge da mediatore tra Model e View, orchestrando il flusso di dati e le interazioni. Le sue responsabilità principali sono: gestire gli eventi dell'interfaccia utente (come clic su pulsanti), configurare e avviare l'analisi delle dipendenze, processare i risultati emessi dal Model, aggiornare la View in base ai risultati elaborati e, infine, gestire errori e situazioni eccezionali

  Il Controller fa ampio uso di costrutti reattivi per gestire in modo efficiente e non bloccante il flusso di dati tra Model e View.

Flussi reattivi per l'analisi delle dipendenze

Il cuore dell'implementazione risiede nell'utilizzo dei flussi reattivi di RxJava per gestire l'intero processo di analisi delle dipendenze. In particolare abbiamo scelto di utilizzare la classe Flowable anziché Observable per beneficiare della gestione della backpressure, fondamentale quando si processano progetti di grandi dimensioni con potenzialmente migliaia di file. La pipeline reattiva inizia con la scansione ricorsiva della directory del progetto, trasformata in un flusso di oggetti Path rappresentanti i file Java. Ogni file Java nel flusso viene quindi trasformato in un oggetto ClassDependency attraverso l'operatore map. Utilizziamo Schedulers.io() per eseguire le operazioni I/O-intensive su un pool di thread dedicato, mantenendo l'event loop principale reattivo e non bloccato. L'utilizzo di observeOn(Schedulers.single()) garantisce che l'elaborazione dei risultati avvenga in modo sequenziale, evitando condizioni di race tra gli aggiornamenti della UI.

Gestione delle operazioni I/O e parsing del codice

Una delle sfide principali nell'analisi del codice sorgente è la gestione efficiente delle operazioni di I/O. Per superare questa sfida abbiamo incapsulato le operazioni di lettura dei file all'interno di un metodo getJavaFiles che restituisce un Flowable<Path>. Questo metodo utilizza Files.walk() per attraversare ricorsivamente la directory del progetto, ma lo fa all'interno di un Flowable.defer() per assicurare che l'operazione non venga eseguita finché un subscriber non si abbona al flusso.

Il parsing di ciascun file Java è stato implementato come una trasformazione reattiva attraverso l'operatore map, che converte ogni Path in un ClassDependency.

All'interno del metodo parseClassDependencies, utilizziamo JavaParser per analizzare il file sorgente e costruire l'AST. L'uso di subscribeOn(Schedulers.io()) garantisce che queste operazioni potenzialmente bloccanti vengano eseguite su thread dedicati.
Per evitare che errori durante il parsing di un singolo file compromettano l'intero processo, abbiamo implementato una robusta gestione degli errori, permettendo al flusso di continuare anche in presenza di file malformati o inaccessibili.
Questo approccio reattivo alla gestione I/O ha consentito all'applicazione di rimanere responsiva anche durante l'analisi di progetti di grandi dimensioni.

Visualizzazione dinamica del grafo delle dipendenze

La visualizzazione del grafo delle dipendenze è stata implementata utilizzando la libreria GraphStream, integrata con la nostra architettura reattiva. Per mantenere il grafo aggiornato in tempo reale, abbiamo sottoscritto il controller al flusso di oggetti ClassDependency e, per ogni nuovo oggetto emesso, abbiamo aggiornato il grafo.

Gli aggiornamenti dell'interfaccia grafica sono stati incapsulati all'interno di Platform.runLater() per garantire che vengano eseguiti sul thread dell'UI di JavaFX, mantenendo l'interfaccia responsiva.
Abbiamo implementato il metodo updateGraph() per aggiungere incrementalmente nodi e archi al grafo, riflettendo le dipendenze tra classi man mano che vengono scoperte.
Per migliorare l'usabilità, abbiamo aggiunto controlli per lo zoom e il panning del grafo, rendendo la visualizzazione più interattiva e facile da esplorare.

Questa integrazione tra RxJava e GraphStream ha permesso di visualizzare in tempo reale l'evoluzione del grafo delle dipendenze, offrendo un feedback immediato all'utente durante l'analisi.

Strategie di backpressure per la gestione dei dati

Una delle sfide nella programmazione reattiva è gestire situazioni in cui i dati vengono prodotti più velocemente di quanto possano essere consumati. Per affrontare questo problema abbiamo applicato la strategia di backpressure (onBackpressureBuffer) con un buffer di dimensione fissa. Questo buffer agisce come un tampone che assorbe i picchi nella produzione di dati, evitando che il sistema si sovraccarichi quando molti file vengono analizzati rapidamente. In caso di overflow del buffer, abbiamo configurato la strategia per emettere un errore (BackpressureOverflowStrategy.ERROR), che viene poi gestito appropriatamente nel handler degli errori.

Inoltre, l'uso di observeOn(Schedulers.single()) garantisce che il processing dei risultati avvenga in modo sequenziale, riducendo ulteriormente il rischio di sovraccarico.

Filtraggio delle dipendenze

Per rendere il grafo delle dipendenze più leggibile e focalizzato sulle relazioni significative tra le classi del progetto, anche in questo modulo abbiamo utilizzato il sistema di filtraggio comune alla parte asincrona di progetto, che tramite un parser personalizzato esclude automaticamente le dipendenze verso classi standard di Java, definite in un set di package da escludere.
Durante il parsing di ciascuna classe, verifichiamo che le dipendenze trovate non appartengano ai package esclusi, inoltre escludiamo i tipi primitivi e i tipi array.
Questo filtraggio ha notevolmente migliorato la qualità del grafo risultante, mettendo in evidenza le relazioni architetturali significative piuttosto che le dipendenze di libreria comuni.

Gestione reattiva degli errori

In un sistema distribuito e basato su flussi come il nostro, la gestione degli errori è cruciale. Gli errori che si verificano durante l'analisi dei file (come file non trovati o errori di parsing) vengono catturati all'interno dei flussi reattivi e gestiti nel handler degli errori. Questo approccio permette di gestire gli errori in modo non bloccante, fornendo feedback all'utente senza interrompere l'analisi degli altri file.
Per errori critici come l'overflow del buffer di backpressure, il sistema fornisce messaggi specifici che aiutano l'utente a comprendere la natura del problema. In ogni caso, anche in presenza di errori, l'interfaccia utente rimane responsiva e i controlli vengono riattivati, permettendo all'utente di intraprendere azioni correttive.
Questa gestione reattiva degli errori contribuisce alla robustezza complessiva del sistema, rendendolo resiliente a varie condizioni di errore che possono verificarsi durante l'analisi di progetti complessi.

Disposizione delle risorse e gestione del ciclo di vita

Una corretta gestione delle risorse è essenziale in applicazioni reattive per evitare memory leak. Nel nostro sistema utilizziamo CompositeDisposable per tenere traccia di tutte le sottoscrizioni ai flussi reattivi. Quando l'applicazione viene chiusa, chiamiamo il metodo dispose() su questo oggetto per annullare tutte le sottoscrizioni attive. Implementiamo anche una corretta pulizia delle risorse di GraphStream tramite il metodo closeViewer().
Infine, utilizziamo Schedulers.shutdown() per un corretto spegnimento dei thread pool utilizzati da RxJava.
Questa attenzione alla gestione delle risorse garantisce che l'applicazione funzioni correttamente anche dopo lunghe sessioni di utilizzo e che si chiuda pulitamente senza lasciare thread orfani o risorse non rilasciate.

Implementazione dell'interfaccia utente reattiva

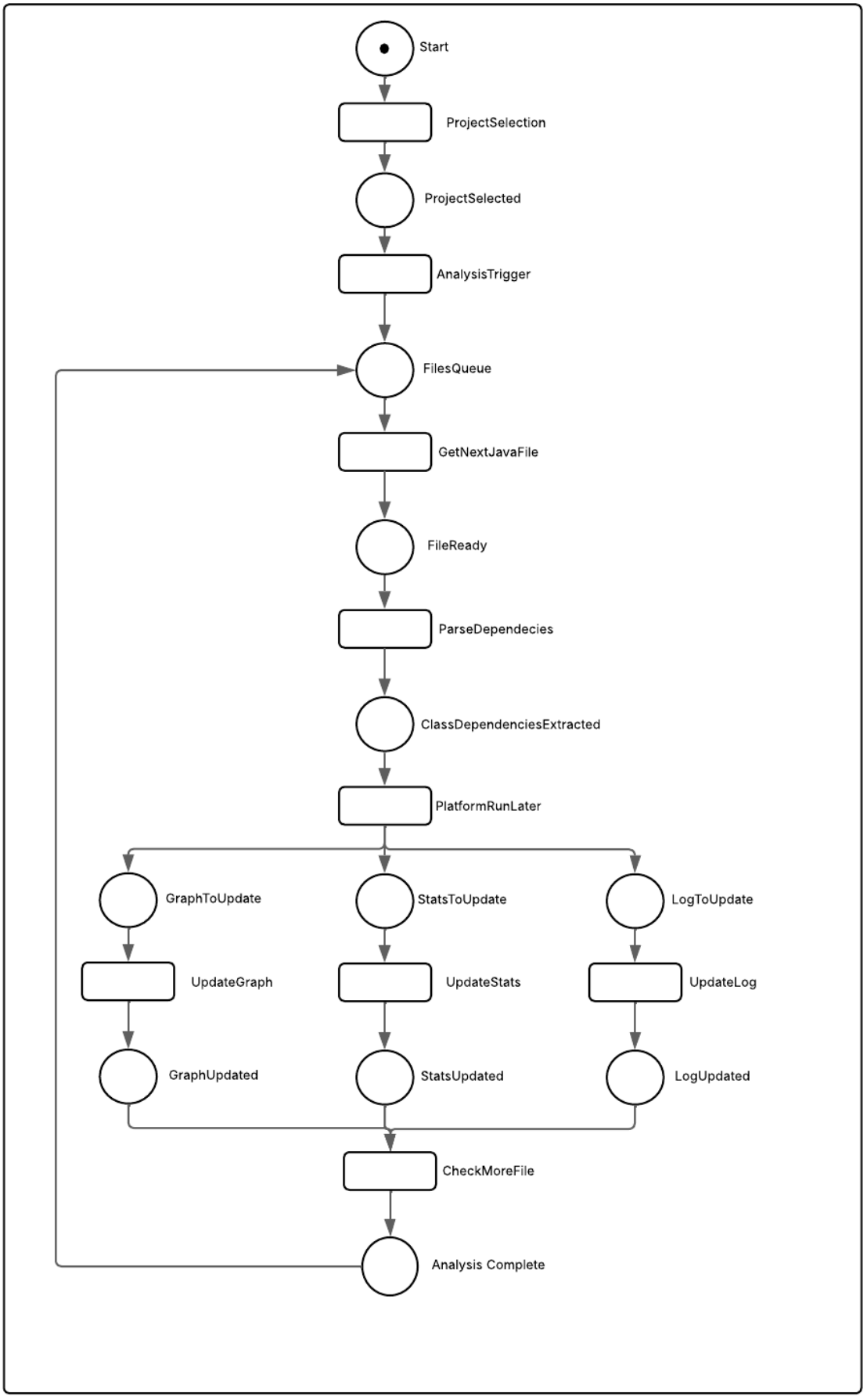
L'interfaccia utente è stata progettata per essere non solo visivamente attraente, ma anche reattiva agli eventi e ai flussi di dati, abbiamo utilizzato controlli JavaFX come Button, TextArea, Slider e ScrollPane per creare un'interfaccia intuitiva e funzionale.
Per garantire che l'interfaccia rimanga responsiva durante l'analisi, abbiamo disabilitato strategicamente i controlli che potrebbero interferire con un'analisi in corso. Le statistiche in tempo reale (numero di classi e dipendenze) vengono aggiornate in modo reattivo tramite contatori atomici.

Il controllo dello zoom del grafo è stato implementato in modo reattivo utilizzando i listener di JavaFX. Questa implementazione reattiva dell'UI migliora significativamente l'esperienza dell'utente, fornendo feedback immediato e mantenendo l'applicazione responsiva anche durante operazioni intensive.

Petri’s net

Questa rete di Petri illustra il flusso di esecuzione reattivo del nostro sistema, dimostrando chiaramente la sua natura reattiva, evidenziando come i vari componenti interagiscano in risposta agli eventi generati durante l'analisi.

main


Start
ProjectSelection
ProjectSelected
AnalysisTrigger
FilesQueue
GetNextJavaFile
FileReady
ParseDependecies
ClassDependenciesExtracted
PlatformRunLater
GraphToUpdate
StatsToUpdate
LogToUpdate
UpdateGraph
UpdateStats
UpdateLog
GraphUpdated
StatsUpdated
LogUpdated
CheckMoreFile
Analysis Complete