PCD – Assignment 02

jiahao Guo matricola 0001173814

12 maggio 2025

Analisi del Problema

L'obiettivo dell'Assignment 02 del corso di Programmazione Concorrente e Distribuita è la realizzazione di due componenti software:

- Una libreria asincrona per l'analisi delle dipendenze tra file sorgente Java.
- Un'applicazione con interfaccia grafica (GUI) che sfrutti la programmazione reattiva per trovare le dipendenze dei file per poi visualizzare i risultati dell'analisi.

Requisiti della libreria asincrona

- Deve fornire tre metodi asincroni per analizzare le dipendenze:
  - di un singolo file,
  - di un intero package,
  - dell'intero progetto.
- Ogni metodo è associato a un test che stampa le dipendenze rilevate in console.

Requisiti della GUI reattiva

- Un selettore per la directory root da analizzare.
- Un pulsante per avviare l'analisi.
- Due contatori (classi e dipendenze).
- Visualizzazione dinamica dei risultati trovati.

I passaggi principali sono:

1. Individuazione dei file da analizzare.
2. Lettura dei file.
3. Parsing con JavaParser.
4. Generazione grafo e jtree delle dipendenze.
5. Visualizzazione del risultato.

Design e Architettura della Soluzione Asincrona

Per la libreria asincrona si è utilizzata `Vert.x` in combinazione con `JavaParser`. Ogni metodo restituisce una `Promise`, ovvero un risultato calcolato asincronamente. Il metodo base per l'analisi di una singola classe è stato riutilizzato per i metodi più complessi.

Design e Architettura della Soluzione Reattiva

Per la realizzazione dell'interfaccia grafica reattiva, si è adottato un approccio basato su diverse librerie tra cui `RxJava`, utilizzata per gestire i flussi di dati in modo reattivo, `JavaParser` per l'analisi sintattica dei file Java, `JGraphT` per la modellazione delle dipendenze sotto forma di grafo, e `JTree` per la visualizzazione gerarchica dei pacchetti e delle classi. Il modello dell'applicazione espone un metodo principale che restituisce un oggetto di tipo `Flowable`, il quale rappresenta un flusso asincrono e osservabile di risultati. Questo flusso viene poi sottoscritto direttamente dalla GUI realizzata in Swing, che reagisce a ogni nuovo dato ricevuto aggiornando dinamicamente l'interfaccia utente. La logica si basa sul codice seguente:

ReactiveAnalyzer

```
public class ReactiveAnalyzer {
  public Flowable<Dependencies> analyzeDependencies(Path srcJava) {
    return Flowable.create(emitter -> {
      try {
        ParserConfiguration configuration = new ParserConfiguration();
        configuration.setLanguageLevel(ParserConfiguration.LanguageLevel.
            JAVA_21);
        StaticJavaParser.setConfiguration(configuration);

        CompilationUnit cu = StaticJavaParser.parse(srcJava);
        String packageName = cu.getPackageDeclaration()
          .map(NodeWithName::getNameAsString)
          .map(pkg -> pkg.substring(pkg.lastIndexOf('.') + 1))
          .orElse(srcJava.getParent().getFileName().toString());

        String className = cu.findFirst(ClassOrInterfaceDeclaration.class)
          .map(ClassOrInterfaceDeclaration::getNameAsString)
          .orElse(srcJava.getFileName().toString().replace(".java", ""));

        Set<String> dep = cu.findAll(ClassOrInterfaceType.class).stream()
          .map(ClassOrInterfaceType::getNameAsString)
          .collect(Collectors.toSet());
        dep.addAll(cu.findAll(ObjectCreationExpr.class).stream()
          .map(ObjectCreationExpr::getTypeAsString)
          .collect(Collectors.toSet()));

        emitter.onNext(new Dependencies(className, packageName, dep,
            srcJava));
        emitter.onComplete();
```

```
      } catch (Exception e) {
        emitter.onError(e);
      }
    }, BackpressureStrategy.BUFFER);
  }
}
```

Gestione GUI tramite Flowable

```
Flowable<Dependencies> dependencyObservable = analyzePath(analyzer, srcPath
    );

dependencyObservable.subscribe(result -> SwingUtilities.invokeLater(() -> {
  String className = result.className();
  String packageName = result.packageName();
  Set<String> dependencies = result.getDependencies();
  Path srcParent = result.srcPath().getParent().getParent();

  model.getGraph().addVertex(className);
  for (String dep : dependencies) {
    model.addVertex(dep);
    model.addEdge(className, dep);
    model.incrementDependenciesCounter();
  }

  model.addVertex(packageName);
  model.addEdge(packageName, className);
  List<String> listPackages = new LinkedList<>();
  listPackages.add(packageName);
  if (srcParent != null && !packageName.equals(model.getProjectRoot())) {
    while (srcParent != null && !packageName.equals(model.getProjectRoot())
        ) {
      String folder = srcParent.getFileName().toString();
      model.addVertex(folder);
      model.addEdge(folder, packageName);
      listPackages.addFirst(folder);
      packageName = folder;
      srcParent = srcParent.getParent();
    }
    view.addPackageToPackage(listPackages, className, dependencies);
  } else {
    view.addNodeToJTree(result.packageName(), className, dependencies);
  }

  view.updateTreeModel();
  view.expandAllNodes();

  model.addVertex(model.getProjectRoot());
  model.addEdge(model.getProjectRoot(), packageName);
```

```
  model.incrementClassCounter();
  view.getClassCountLabel().setText("Classes: " + model.getClassCounter());
  view.getDepCountLabel().setText("Dependencies: " + model.
      getDependenciesCounter());

  updateGraphView(model, view);
}), Throwable::printStackTrace);
```

Codice della Funzione `analyzePath`

Di seguito è riportato il codice della funzione `analyzePath`, che utilizza `Flowable` per analizzare i file Java all'interno di una directory.

```
private Flowable<Dependencies> analyzePath(ReactiveAnalyzer depReactive,
    Path root) {
    return Flowable.using(
      () -> Files.walk(root),
      files -> Flowable.fromStream(files)
        .filter(p -> p.toString().endsWith(".java"))
        .flatMap(path -> depReactive.analyzeDependencies(path)
          .subscribeOn(Schedulers.io())),
      Stream::close
    ).observeOn(Schedulers.computation());
}
```

Questo codice assicura che ogni aggiornamento all'interfaccia grafica sia eseguito nel thread corretto (`EDT`). La struttura dei pacchetti viene rappresentata graficamente, con aggiornamenti dinamici e contatori aggiornati in tempo reale.

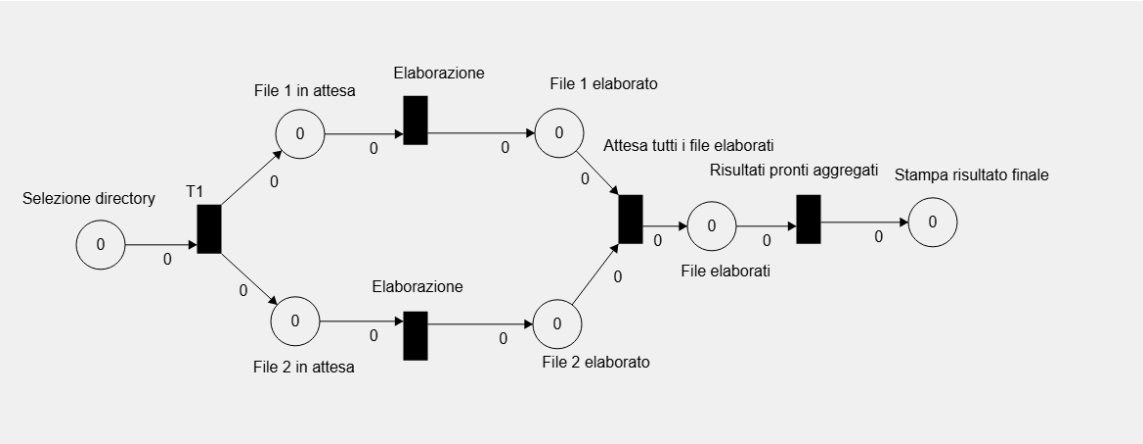

Figura 1: Petri net della soluzione asincrona.

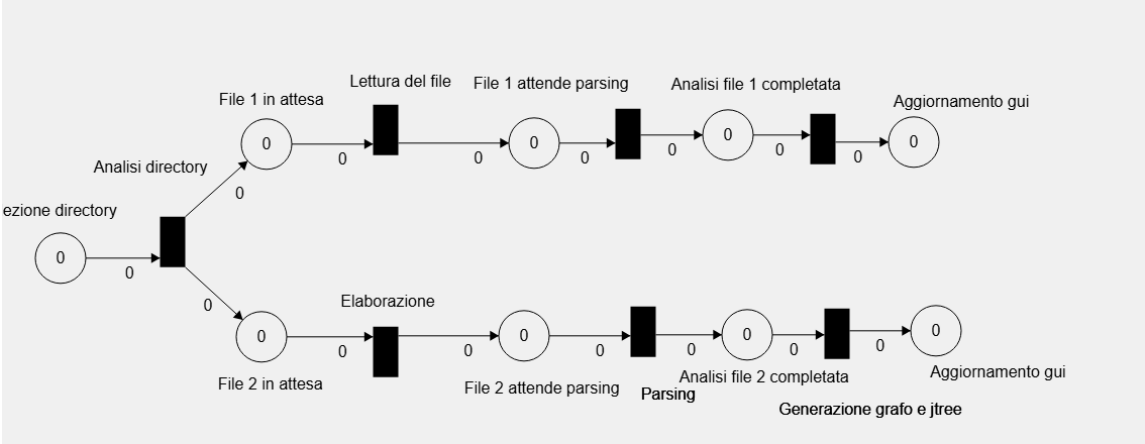

Figura 2: Petri net della soluzione reattiva. Si noti che per ogni file letto viene eseguito il ciclo completo delle operazioni.