PCD - Assignment 2

Author 1
nicolas.amadori@studio.unibo.it
Author 2
riccardo.mazzi@studio.unibo.it

May 2025

Contents

1 Analysis

The first goal of this assignment is to develop an asynchronous Java library designed to analyze type dependencies within a project. Its main purpose is to identify which classes, interfaces, and packages are used or referenced throughout a codebase.

The library provides asynchronous methods to analyze dependencies at the class, package, and project levels. It returns custom reports objects showing which types are accessed in each scope, enabling efficient dependency tracking and supporting tasks like refactoring and architectural analysis. The main aspect to take into consideration was that every package and class dependencies needed to be calculated independently from each other, so it was crucial to split up the work.

The second part aims to create a graphical application that leverages a reactive programming approach to provide a dynamic, interactive view of code dependencies. The application should allow users to visualize the relationships between classes and packages in a Java project as a live, incrementally updated graph, with classes organized by packages.

The user interface should include a component to select the project's source root directory, a button to initiate the analysis, and a panel to display real-time results. This

panel should also feature summary boxes showing the number of classes and interfaces processed, as well as the total number of dependencies detected.

2 Design, Strategy and Architecture

2.1 Library structure

We designed a static library class with three methods:

`getClassDependencies(classSrcFile)` returns a `ClassDepsReport`, composed of the unique class name (complete with the package, e.g. "java.util.List") and the set of strings representing the class dependencies.

`getPackageDependencies(packageSrcFolder)` returns a `PackageDepsReport` object, which includes the name of the package and a set of `ClassDepsReport` objects representing the dependencies of each class within that package. This structure allows dependencies to be grouped and managed at the package level.

`getProjectDependencies(projectSrcFolder)` returns a `ProjectDepsReport` object, which contains the project's root path along with all associated `PackageDepsReport` instances. The design follows the same principle as in `PackageDepsReport`: to provide a structured representation of dependencies, this time at the project level.

2.2 Implementation details

All three types of reports extend the same interface `DepsReport`, easing the use of the library.

We use the JavaParser library `com.github.javaparser` to parse class source files and extract their explicit import dependencies. To identify implicit dependencies, we use a `CombinedTypeSolver` configured with the project's root directory. This setup includes a `JavaParserTypeSolver` to resolve dependencies within the same package, and a `ReflectionTypeSolver` to handle types from `java.lang`, which we filter out to avoid counting standard library references.

In the asynchronous implementation, each report set (of type `String`, `ClassDepsReport`, and `PackageDepsReport`) is stored in a `List`, as data is accessed only after the computation completes (Figure 1).

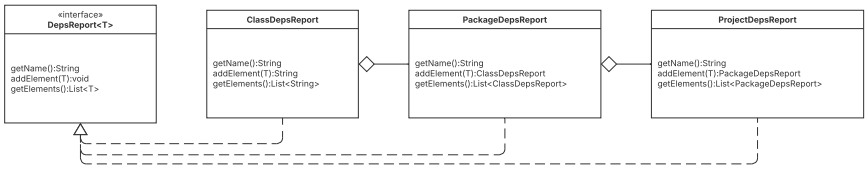

Figure 1: Asyncronous implementation details

In the reactive implementation, as shown in Figure 3, the `List` is replaced with an `Observable`, allowing the GUI to consume the data stream incrementally during computation, without waiting for the full analysis.

For the reactive functionality, we built a GUI that lets users start by selecting the source directory of their Java project. Once the analysis begins with a button click, the interface dynamically updates to show elements as they are processed. Classes are visually grouped within bordered areas representing their packages, and connections between elements illustrate dependencies either between classes or from classes to packages (when an entire package is imported).

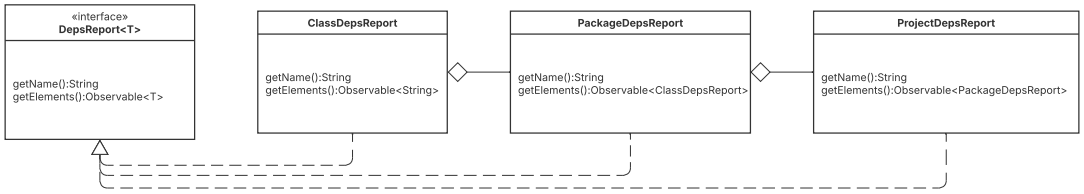

Figure 2: Reactive implementation details

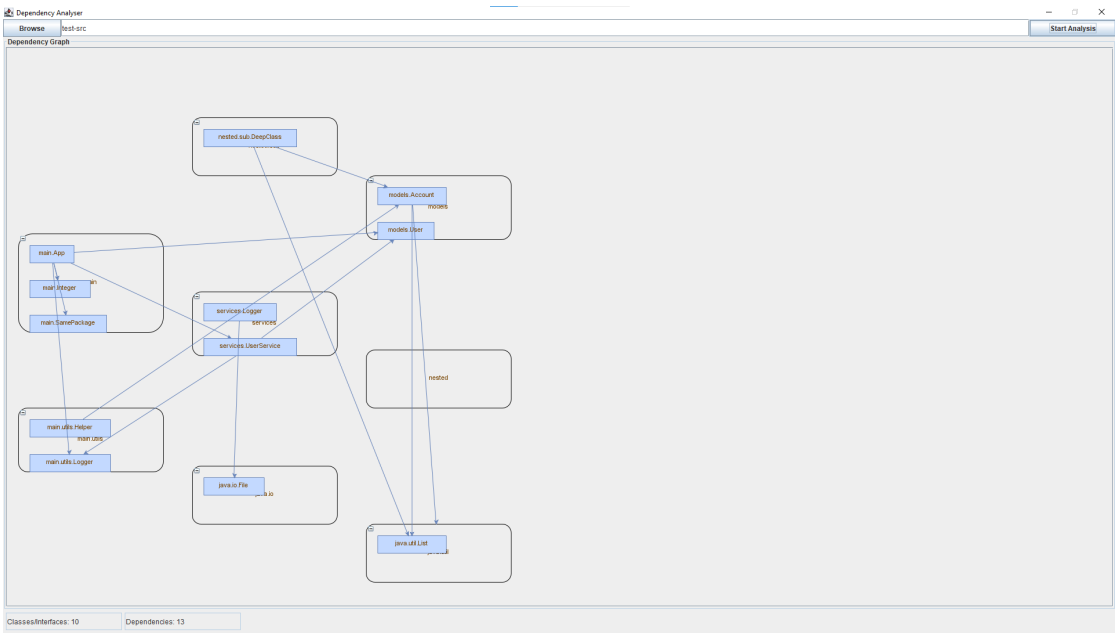

Figure 3: Screenshot from our GUI.

3 Behaviour

As shown in Figure 4, the main thread of the user's application can invoke our library's functions asynchronously, allowing the analysis to run in the background. Each function completes with either a successful result or an error, depending on the outcome of the operation. Future.all is an asynchronous method that combines multiple futures into a single one, which succeeds only if all the individual futures complete successfully but will fail otherwise.

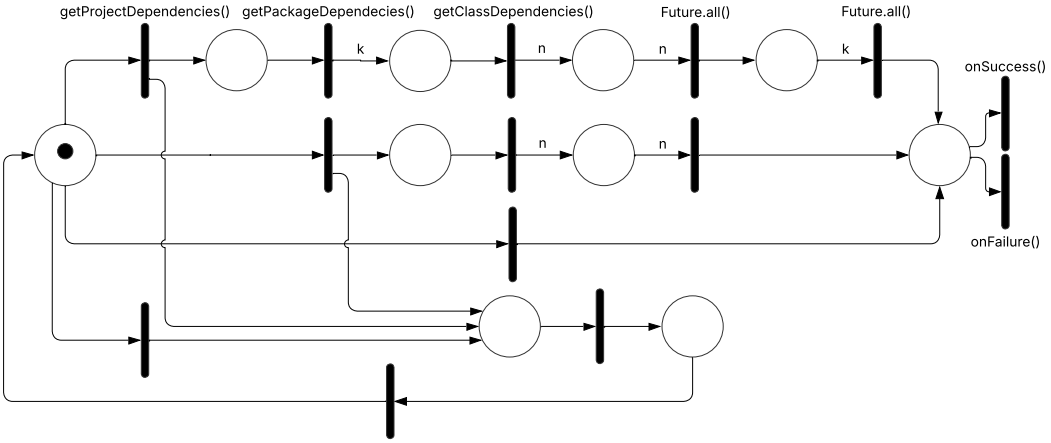

Figure 4: Petri Net representation of the asyncronous behaviour.

In Figure 5, we demonstrate reactive behavior that incrementally updates the view each time a new result becomes available. This is achieved through subscription to Observables, ensuring that the interface reflects progress in real time as the analysis unfolds.

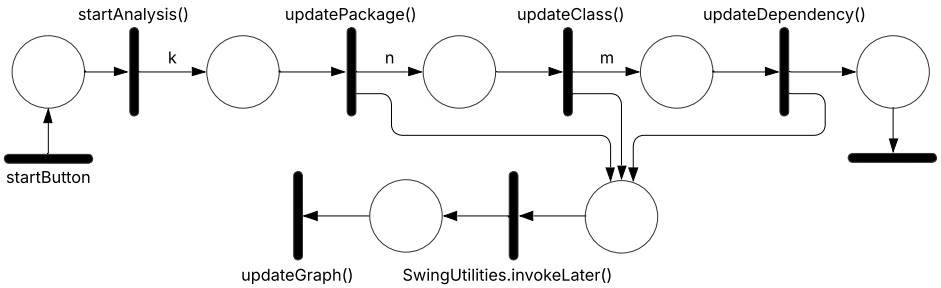

Figure 5: Petri Net representing the reactive implementation.