PCD-Assignment02

Alessandro Agosta

May 11, 2025

Contents

1 Introduction

In this report, I will analyze the process of designing, implementing and testing a dependency analyzer software for Java-based projects.
This software is structure in two parts with different technologies: asynchronous programming and reactive programming, applied in two project: a library based on asynchronism and a gui application based on reactivity.

2 Analysis

A "software dependency" is when code requires a specific functionality or feature from an external code component. This requirement creates a dependency or reliance on code outside the main software logic.
In the context of this report, a Java's dependency will refer to its individual import, as a more in-depth analysis would divert time and resources from the aim on the task at hand, being the implementation of this dependency analyzer using asynchronous and reactive programming.

3 Design

In this section, I will explore the choices for each implementation, their differences and similarities.
Since both projects need to analyze and condense every found dependency, I've decided to utilize a common data structure for both projects to rely upon; in the form of a graph.
This graph is composed by edges and arcs, each edge symbolizes a package or subpackage where a certain dependency is found, while arcs connect a package to its subpackages or to the actual Java class or interface in the dependencies.

3.1 Asynchronous

Asynchronous programming can be implemented in many ways:

- Event-driven: where there's an event loop based control flow
- Async Functions & CPS: with the use of event handlers and promises
- Asyncawait mechanisms: mimicking asynchronously while operating synchronously
- Coroutine concurrency: using lightweight threads

Each way of operating asynchronously has its pros and cons, for the purpose of this project I've relied on a Java library called "Vert.x".
Vert.x processes computation using an event-loop, where event can almost be anything, albeit that they can be processed in a fixed time window; such that the event-loop handling is not blocked waiting.

Vert.x also introduces a Future mechanism, different from the JDK futures', with the added feature of being composable, joinable and queried in a non-blocking fashion; this feature has been largely used in the making of the asynchronous part. In particular future's composition was extensively used in designing a flow from raw to processed data.
While Vert.x offers an abstraction of its asynchronous features in the form of AbstractVerticles, I've decided to challenge myself and rely entirely on function calls and future compositions.

3.2 Reactive

Reactive programming's aim is to resolve the CPS and promise's inability to manage streams of data/events; reactive programming is oriented around data flows and the propagation of change, implementing an Observer pattern mixed with event driven programming.
In this project reactive programming was introduced with the Java library "RxJava", a "Reactive Extension" for the JVM; it extends the observer pattern introducing "Reactive Streams" and their listeners. A Stream represents an unbounded number of elements to process fed from producer to consumer; RxJava introduces also many utilities to interact, modify and merge between streams.
This feature was used to create a reactive application with a producer and consumer configurated in a push-based model, where new data from the producer is only consumed by the consumer.

4 Behavior

In this section I will go more in-depth into the behavior of each project, providing also graphical examples in the form of Petri Nets.
As mentioned before a main behavioral structure is shared, mainly consisting in:

1. Reading all contents of a given directory
2. Filtering ahead files while reading the contents of any sub-directory found
3. Analyzing each found file and collecting its dependencies
4. Delivering dependencies to consumers

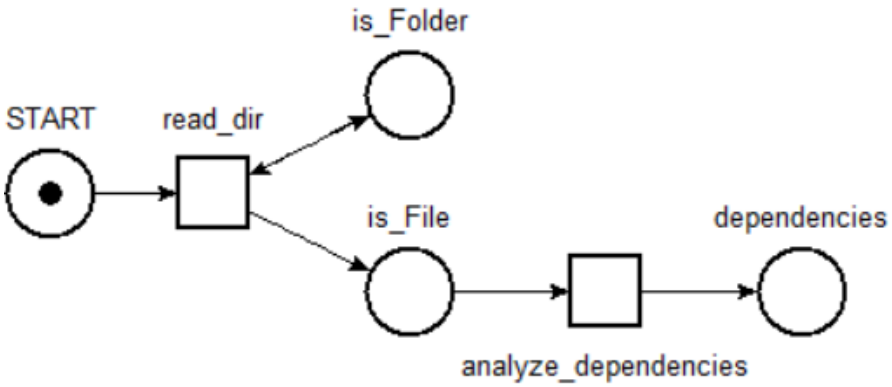

Figure 1: Common shared behavior

In figure 1 I've designed, using Petri Nets, how the implementation interact and operate, closely following the afore-mentioned structural steps.

4.1 Asynchronous Futures

This project aims to make a library for analyzing dependencies of either a Java class, package or project; given the path to each analyzable item.
As mentioned before in this project, I've exclusively utilized Future's composition techniques provided by Vert.x, to be specific I've linked many "compositions" that translate to a series of functions that map the input (the path of either a file, package or project source folder), with many transformations, to a report of its dependencies, in the form of a graph.

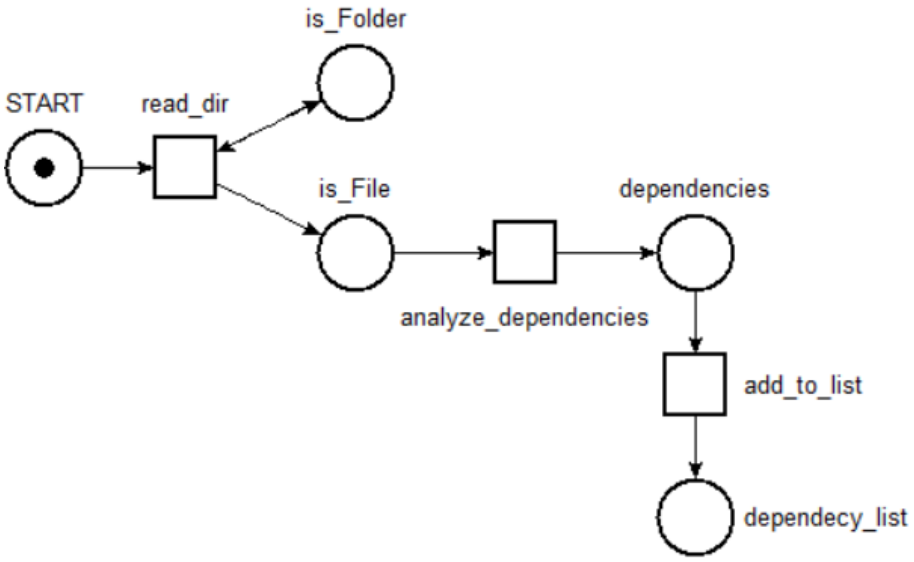

Figure 2: Common shared behavior

Figure 2 shared similarities with Figure 1, although it add every dependency found to a list that is then outputted in the form of a "Future".

4.2 Reactive

The reactive project is structured with a single producer of class dependencies and a single consumer that shows the found dependencies in the form of a directed graph. The producer emits each class dependency found while also exploring recursively each subfolder found and the files within.
The consumer receives and organizes each dependency in the form of a drawable graph, assigning to each edge a pair of coordinates.

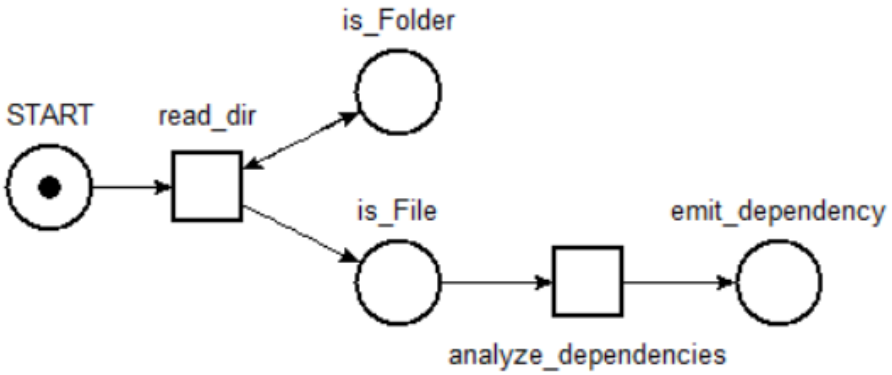

Figure 3: Reactive Behavior

This behavior is show in figure 3. While in the Asynchronous project, there was no need to specify who and where was tasked with computation, reactive programming heavily benefits from delegating computation to specific schedulers; in this project access to the filesystem was delegated to I/O specific schedulers, while large computation was meant for computation specific schedulers. Since a GUI part is also present, there was the need to delegate all visual updates to GUI specific handler.

5 Testing

Both implementation feature moderate testing to ensure quality and accuracy; tests consist in JUnit-style testing for minor software parts and a log-style testing for every other code integration.