University of Calgary
SENG 300 - Introduction to Software Engineering

Group Project Iteration 2

Finding Declarations and References

March 26, 2018

Esther Chung 10080030, Tyler Chow 30024575, Josh Schijns 10017168, Irene Chan 10103807, Patrick Gharib 10137116

1 Structural Diagram

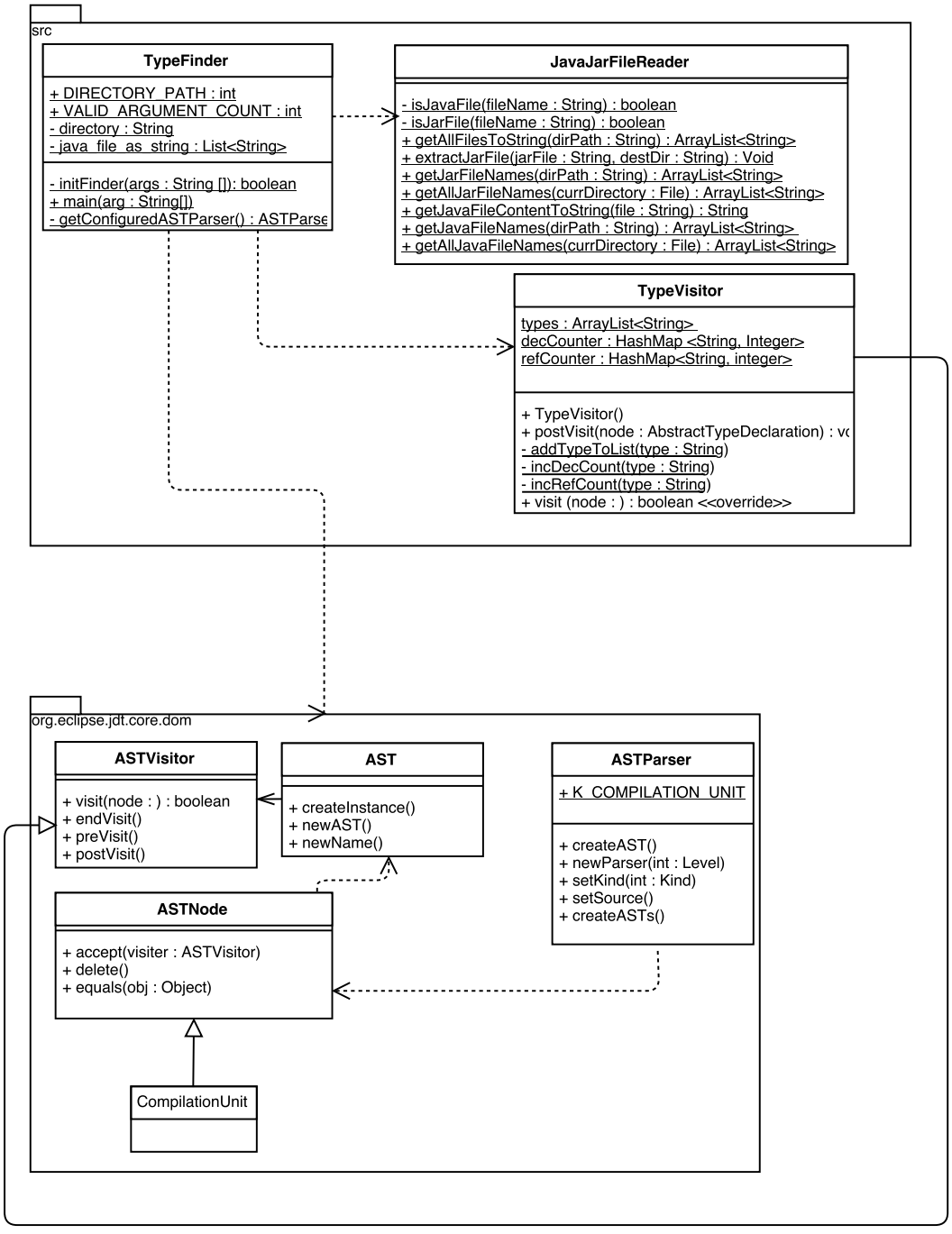

Figure 1: The relationship between our main package and relevant classes provided by org.eclipse.jdt.core.dom

2 Sequence Diagrams

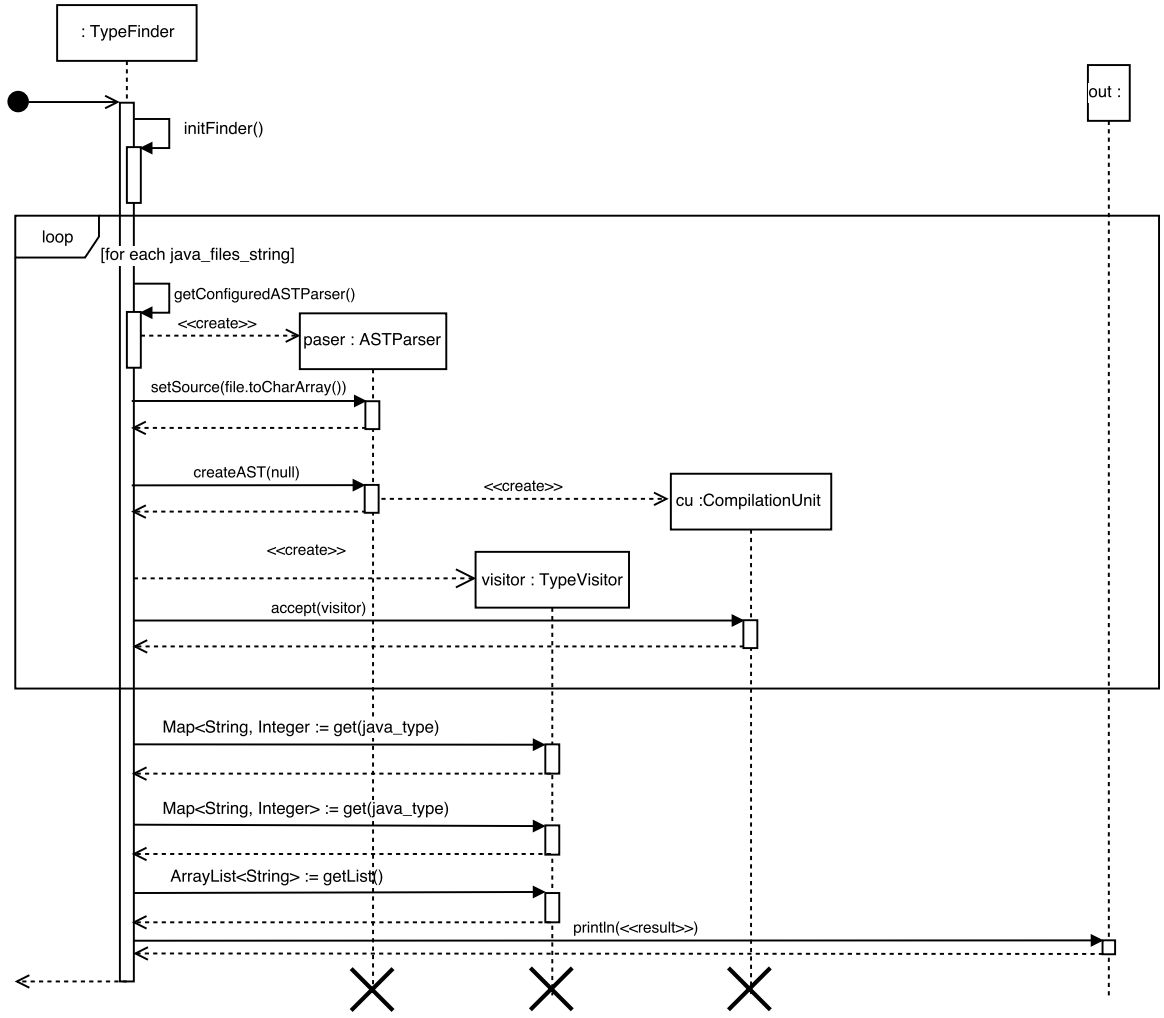

Figure 2: Sequence of TypeFinder program intialization and completion

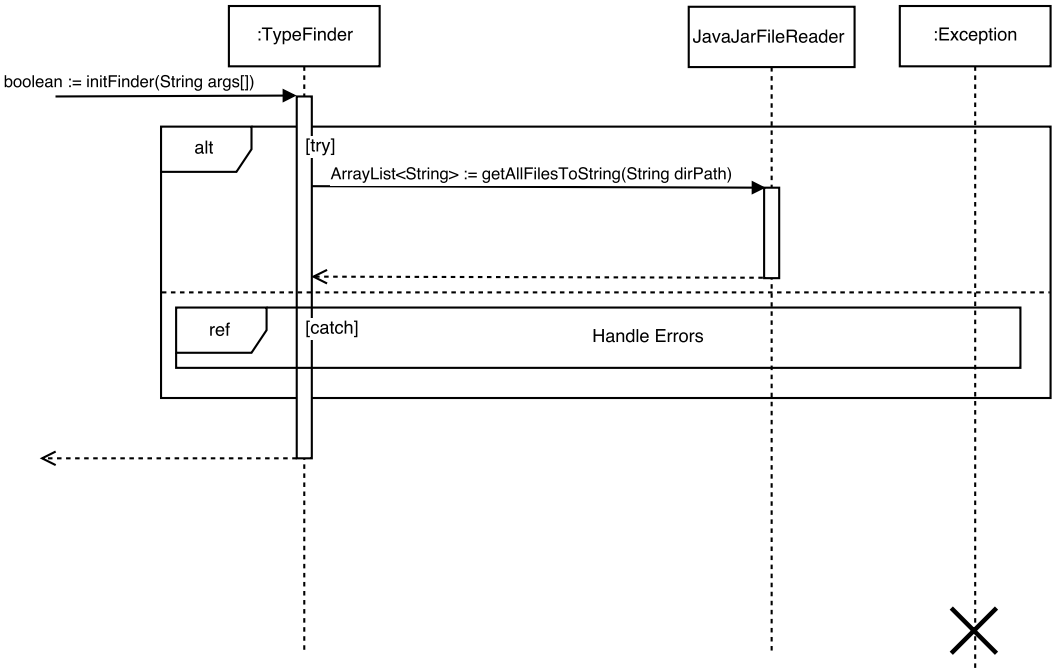

Figure 3: Initializing TypeFinder involving checking for valid user input. If valid, it acqures the Java file contents and setting up the information necessary for parsing. If invalid, prompt the user with an error message.

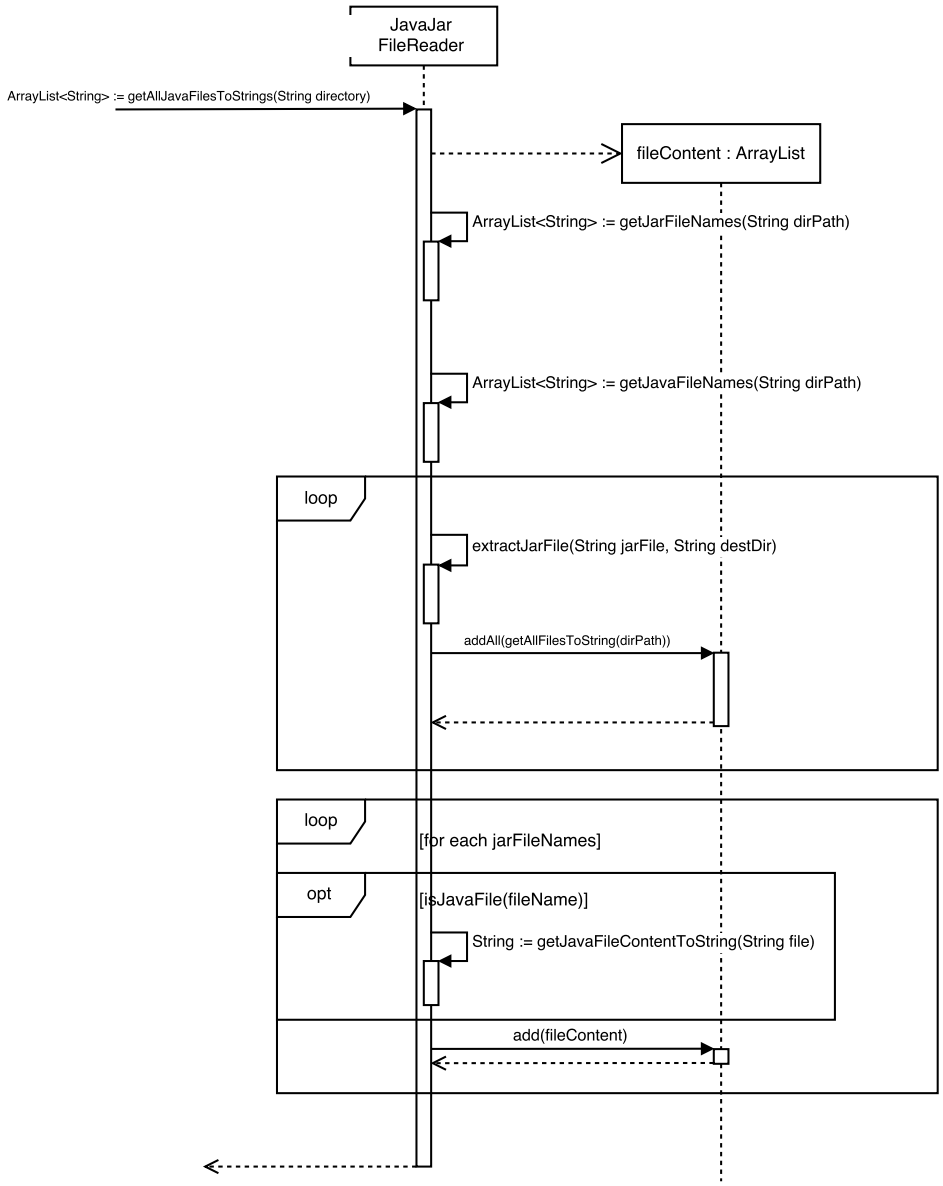

Figure 4: JavaJarFileReader retrieves the contents of all Java files in a directory, one file at a time

3 State Diagram

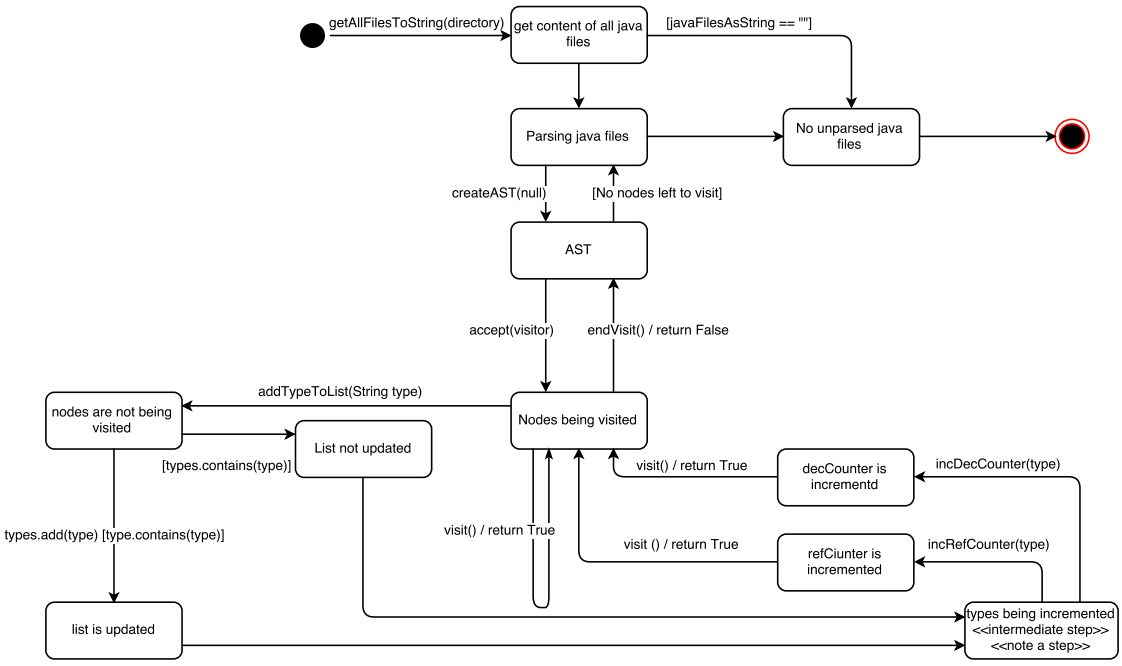

Figure 5: The state of the program in finding declaration and reference counts.

4 Explanation

4.1 Usage

Run the Java `TypeFinder.class` file through command line:
`java TypeFinder <directory> <Java type>`
`directory` is the path of the directory (either absolute or relative).

4.2 Structure

The UML diagram was constructed with simplicity in mind. We included all the key structures of the program and abstracted away the rest since they were not crucial to our audience in understanding our system. The TypeFinder class is the backbone of our system and drives the system during execution, so naturally most components were preserved. Next the JavaJarFileReader class had the purpose of parsing the directories, unpacking any jars during its recursive decent into the user-specified directory and returning their contents for [as a list of strings]. To accomplish this task, we implemented a TypeVisitor class to traverse the Abstract Syntax Trees associated to each of the Java files located by the JavaJarFileReader class. Many of the helper methods were abstracted away, as they only serve to facilitate the execution of the core methods. They were not pertinent to the understandability of the structure of the system. In particular, it should be noted that, based on the different node types, the visit() methods were overridden multiple times, so in order to avoid confusion we included only a base representation. To present this fact we used an override stereo type.
As mentioned earlier the main purpose of the diagram was to maintain simplicity, so only the key aspects of ASTParser, AST, ASTVistor, ASTNode were preserved. Everything non essential to the comprehension of the structure of out system was abstracted away.

4.3 Sequence Diagram

The System is designed to take in two command line arguments. The first, the directory of interest. The second, a fully qualified Java type name. A call is made to initFinder which recursively searches through any files, jars, nested jars, and any subdirectories that are located in the parent directory. Afterwards, the execution of the main for loop is entered where an AST tree is created for each file and the contents are parsed. The AST is traversed via a visitor when is created at the same time as the AST. the number of references and declarations are kept track of as the visitor continues down the tree. At the end the final count is printed to console.
The inner workings of the getConfiguredASTParser () method is non-essential to the client's understanding of the software. Thus, the details were abstracted away and not expanded on. Another aspect of the code that was abstracted away was most of the methods in JavaJarFileReader. We modeled a very broad overview, this informs anyone viewing the model (provided some basic knowledge of Java) of what is happening. Had we made our diagram anymore specific in this area, it would draw attention away from the more important functionality of the software and would only overwhelm the viewer. A very basic diagram shows the execution of initFinder, which was only meant to serve the purpose of showing that the JavaJarFileReader Class was being used to read the files. A more Specific sequence diagram was provided to show how getAllJavaFilesToStrings () was receiving information about the directory and files, as well as how it returned the result for ASTParser to use.

4.4 State diagram

The state diagrams show the transition of states while visiting the nodes. While visiting a node you can update the type list, once the list is either updated or not updated then then the count is incremented if the type is found. Continuing this cycle of visiting and checking nodes until there are no more nodes to analyze, the visiting state is left return to the loop at main in TypeFinder. Again, many details were abstracted away to reduce clutter and confusion. the main purpose was to aid in the understanding of functionality and execution of analysis tool.