# **System Design Document**

## **Dependency Graph - CS490 - Fall 2020**

***Team Members:***

* Aaron Van De Brook
* Corrina Del Greco
* Luis Mora
* Samantha Shultz

|  |  |
| --- | --- |
| Version | Date |
| 1.0 | 10.1.20 |
| 2.0 | 10.22.20 |
| 3.0 | 11.30.20 |

## 

## 

## 

## ***Table of Contents***

[1. Introduction](https://github.com/AVanDeBrook/c-dependency-graph/wiki/SDS-Document#1-introduction)

[1.1 Purpose and Scope](https://github.com/AVanDeBrook/c-dependency-graph/wiki/SDS-Document#11-purpose-and-scope)

[1.2 Project Executive Summary](https://github.com/AVanDeBrook/c-dependency-graph/wiki/SDS-Document#12-project-executive-summary)

[1.2.1 System Overview](https://github.com/AVanDeBrook/c-dependency-graph/wiki/SDS-Document#121-system-overview)

[1.2.2 Design Constraints](https://github.com/AVanDeBrook/c-dependency-graph/wiki/SDS-Document#122-design-constraints)

[1.2.3 Future Contingencies](https://github.com/AVanDeBrook/c-dependency-graph/wiki/SDS-Document#123-future-contingencies)

[1.3 Document Organization](https://github.com/AVanDeBrook/c-dependency-graph/wiki/SDS-Document#13-document-organization)

[1.4 Project References](https://github.com/AVanDeBrook/c-dependency-graph/wiki/SDS-Document#14-project-references)

[1.5 Glossary](https://github.com/AVanDeBrook/c-dependency-graph/wiki/SDS-Document#15-glossary)

[2. System Architecture](https://github.com/AVanDeBrook/c-dependency-graph/wiki/SDS-Document#2-system-architecture)

[2.1 System Software Architecture](https://github.com/AVanDeBrook/c-dependency-graph/wiki/SDS-Document#21-system-software-architecture)

[2.2 Internal Communications Architecture](https://github.com/AVanDeBrook/c-dependency-graph/wiki/SDS-Document#22-internal-communications-architecture)

[3. Human-Machine Interface](https://github.com/AVanDeBrook/c-dependency-graph/wiki/SDS-Document#3-human-machine-interface)

[3.1 Inputs](https://github.com/AVanDeBrook/c-dependency-graph/wiki/SDS-Document#31-inputs)

[3.2 Outputs](https://github.com/AVanDeBrook/c-dependency-graph/wiki/SDS-Document#32-outputs)

[4. Detailed Design](https://github.com/AVanDeBrook/c-dependency-graph/wiki/SDS-Document#4-detailed-design)

[4.1 Software Detailed Design](https://github.com/AVanDeBrook/c-dependency-graph/wiki/SDS-Document#41-software-detailed-design)

[4.2 Internal Communications Detailed Design](https://github.com/AVanDeBrook/c-dependency-graph/wiki/SDS-Document#42-internal-communications-detailed-design)

[5. External Interfaces](https://github.com/AVanDeBrook/c-dependency-graph/wiki/SDS-Document#5-external-interfaces)

[5.1 Interface Architecture](https://github.com/AVanDeBrook/c-dependency-graph/wiki/SDS-Document#51-interface-architecture)

[5.2 Interface Detailed Design](https://github.com/AVanDeBrook/c-dependency-graph/wiki/SDS-Document#52-interface-detailed-design)

[6. System Integrity Controls](https://github.com/AVanDeBrook/c-dependency-graph/wiki/SDS-Document#6-system-integrity-controls)

## ***Overview***

The System Design Document describes the system requirements, operating environment, system architecture, file design, input formats, output formats, human-machine interfaces, detailed design, processing logic, and external interfaces.

## ***1. Introduction***

### **1.1 Purpose and Scope**

The purpose of the system is to create visual representations of the dependencies between modules, to assist in reducing the amount of work necessary to refactor modules.

### **1.2 Project Executive Summary**

Test-driven development (TDD) has been used widely to improve the quality for software development. To use TDD for embedded systems based on the C programming language, we must be able to run the tests on the host computer, which involves the mockup of the functions designed to run on the target. To reduce the amount of work of mocking-up, we need to reduce the number of function calls (the dependencies) between modules (a module is usually contained in a single source file) as much as possible. Seeing the graph of dependencies is the first step to reduce it. Doxygen can be used to generate call graphs and caller graphs of all functions in a project by generating the Dot (a graph description language) source files, which can be processed by Graphviz (the tool that processes Dot files). While these graphs can show us the dependencies between different modules, they are too cluttered. We need to reduce these graphs to only display the dependencies between the modules.

#### **1.2.1 System Overview**

The system takes Dot files (generated by Doxygen) and then reads and parses them to find the relationships between each of the modules. The system uses these relationships to output a new Dot file with the reformatted graph. The system then takes these generated text files and uses Graphviz to create an actual dependency graph image. The system will only take in applicable Dot files.

#### **1.2.2 Design Constraints**

The development team designed the system assuming that the user has used Doxygen to generate the Dot file(s) and that the user will run with Windows. It is also assumed that function names within modules shall follow the following naming convention: public functions are MOD\_\* and private functions are mod\_\*, where "mod" represents the module the function pertains to and \* represents the function name. As long as these assumptions are followed through, there should be no conflicts between systems.

#### **1.2.3 Future Contingencies**

A future contingency is the addition of a graphical user interface (GUI). A GUI could be used to facilitate a user’s interaction with the system. If the GUI is not built, the system shall use command line arguments to generate output.

### **1.3 Document Organization**

This System Design Document is organized into six sections, including this introduction section. Following this, the document will go over the system architecture, the human-machine interface, the software and internal communications design, external interfaces, and the system integrity controls.

### **1.4 Project References**

* System Requirements Specification Document (<https://github.com/AVanDeBrook/c-dependency-graph/wiki/SRS-Document>)
* Dot Language Grammar (<https://www.graphviz.org/doc/info/lang.html>)

### **1.5 Glossary**

* .dot - The extension for a Dot file
* Dot - A software language to describe directed graphs, also the name of the tool that converts Dot files into images
* Doxygen - A tool for generating software reference documentation, used to generate Dot file(s) from a project for the system to use

## ***2. System Architecture***

The system is able to take in applicable Dot files to understand the relationship between modules. Dot files are produced by Doxygen and are textual representations of a directed graph. Once the Dot files are processed by the system, the system can then develop a dependency graph for the user to better understand the relationships within their system.

### **2.1 System Software Architecture**

This system is programmed in Java and has various classes (see Section 4 for a chart of all the classes and a description of their relationships). The Manager class acts as a kickoff point and an intermediary that passes along necessary data for lower classes to handle. The Configurator class takes in the command line arguments, which set how the program will run. This includes the output, logging, and providing the location of the input Dot files. The Configurator then passes the location of the Dot files to the Reader class, which validates that they have an extension of .dot and converts their contents into strings which are added to a list. The Parser class then processes this list to determine the relationships between graph attributes and create objects to represent them. The Lexer is a helper class that assists the Parser by tokenizing each line of a Dot file. The resultant graph objects are then given to the Writer class, which will output the new Dot files that will be used to create the images of the dependency graph.

### **2.2 Internal Communications Architecture**

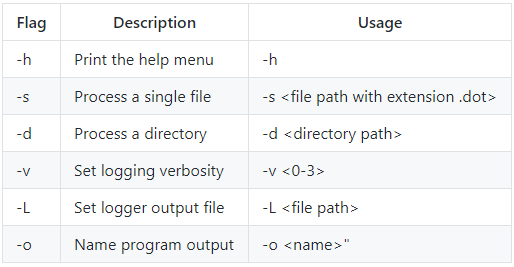
At a high level, the program sequence can be viewed sequentially (predictably from one class to the next), aside from the Manager serving as an intermediary between multiple classes. When the program is run, each class depends on the class before it to have run because it provides information needed for the next step. See Appendix C for the many detailed diagrams illustrating the system's modules and their functions.

## ***3. Human-Machine Interface***

Since there is no need for user input between the input and output stages of the program, the user will interact with the program through the command line. There will be a well-defined list and format of arguments that can be passed to and processed by the program, which are detailed below.

### **3.1 Inputs**

The user will use command line arguments to control the program. A -h command will invoke a help menu for the user to view the various accepted inputs. These include: inputting the level at which the program should log, inputting values for the output files, and most importantly, inputting a single Dot file or a directory location containing Dot files. If the Dot files exist and are valid the program will run through the sequence to produce an output image of the dependency graph. The syntax for the command line arguments are:



**Figure 1: Syntax for Command Line Arguments**

**3.2 Outputs**

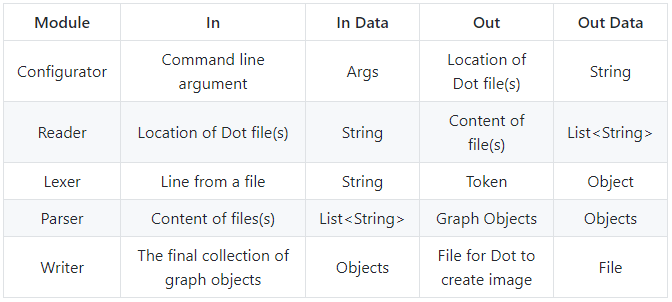
The program will write a text file formatted according to the Dot language specification (see Appendix A). That file will then be processed by Dot and converted into an image file, which is a graphical representation of dependencies between the graphs that were input as pre-existing Dot files. The generated image will conform to the format seen in Appendix B.

## ***4. Detailed Design***

### **4.1 Software Detailed Design**

At a high level, the program flow can be viewed sequentially, meaning from one module to the next in a predictable sequence. Each module serves to perform some kind of transformation which gets the input, Dot file(s), closer to the desired output, an image of a graph. The following text includes descriptions of each class and their relationships with one another, as well as a table showing the input and output for each class.

* ***Manager:*** The Manager will serve as an intermediary between the other modules. For the most part, they do not pass information to each other. Instead they give it to the Manager which will provide it to the next class in the sequence.
* ***Configurator:*** This is the first module in the sequence. Here, the command-line arguments from the user are used to determine the location of a single Dot file or directory containing Dot files. A string containing either the file path or directory is passed (through the Manager) to the Reader. The arguments can also be used to save user-named log files, set verbosity levels of logging, set a name for the output graph file, or print out help instructions.
* ***Reader:*** If a single Dot file’s path is received, that file is read. If a directory is received, every file determined to be a Dot file in that directory is read. A file's contents are converted to a string. A file to a string is a one-to-one relationship. These strings are collected into a list and passed (through the Manager) to the Parser.
* ***Lexer:*** The Lexer class is a helper for the Parser. It takes in a single line from a Dot file and interprets it as a specific type of token (we call this action to "tokenize"). Tokens can be viewed as key value pairs, where the key is a type determined by the Lexer (i.e. node, edge) and the value is where the Lexer stores the rest of the string. The Parser uses the token type to know how to handle the string.
* ***Parser:*** The Parser module receives a list of strings, where each string represents the contents of one Dot file. In this case, to parse means to use the string to create objects which represent the connections, modules, and nodes. Only one string (file) will be parsed at a time. The Parser will loop through the list of strings to parse them one by one and collect the resultant graph objects. It will also remove connections within the same module.
* ***Writer:*** The writer takes the final modified graph objects and writes a file in the Dot language which Dot can create a visual image of. This is the output to the user.

**Figure 2: Input and Output of Each Class**

### **4.2 Internal Communications Detailed Design**

Communications between the modules within the system will be done through Java primitive types, built-in classes (such as Strings), and well-defined enumerators that will be defined within the specified module package (e.g. Configurator package contains the ConfigType enumerator). Otherwise, communication from modules to the user will be done in standard out, using the Java Util Logging package.

## ***5. External Interfaces***

The external interface, which is outside of the scope of the system, is the graphical visualization program Dot. Dot is a program that takes in the text files (in the Dot language, with the extension .dot) and creates the image of the relationship between modules.

### **5.1 Interface Architecture**

Dot is contained within its own binary file. The interactions of our program with the Dot API will be primarily through loading and running the Dot binary on the text file that Doxygen has generated to create an image.

### **5.2 Interface Detailed Design**

Input files must have a file extension of .dot in order for the program to recognize them as input files. Additionally, input files must adhere to the dot language syntax in order to be processed correctly by the system's parser. Errors detected by the program will result in exceptions raised in their respective modules and handled by the main statement in the Manager module. If, for whatever reason, the error cannot be reconciled with, this will be reported as an issue for future handling.

## ***6. System Integrity Controls***

* The system will make copies of the contents of files handled, and will never modify them directly. The only case in which the contents of a file are modified is upon the creation of the output file. Therefore integrity of data on the system does not need to be addressed.
* Test cases have been written to ensure proper function of the Reader system. Beyond that, there is no concern over loss of data integrity with this system.

## ***Appendix A: Dot Language Specification***

The formal specification of the Dot Language can be found here: <https://www.graphviz.org/doc/info/lang.html>

## 

## 

## 

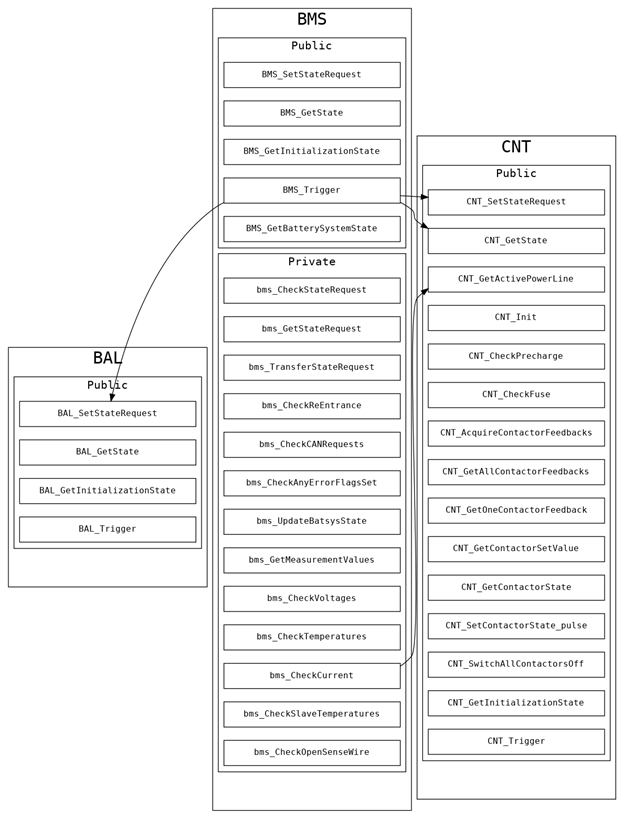
## 

## 

## 

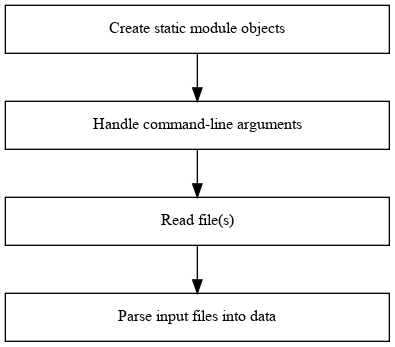
## ***Appendix B: Manually Generated Graph***

The following graph was generated manually in order to illustrate the intended output of the system.

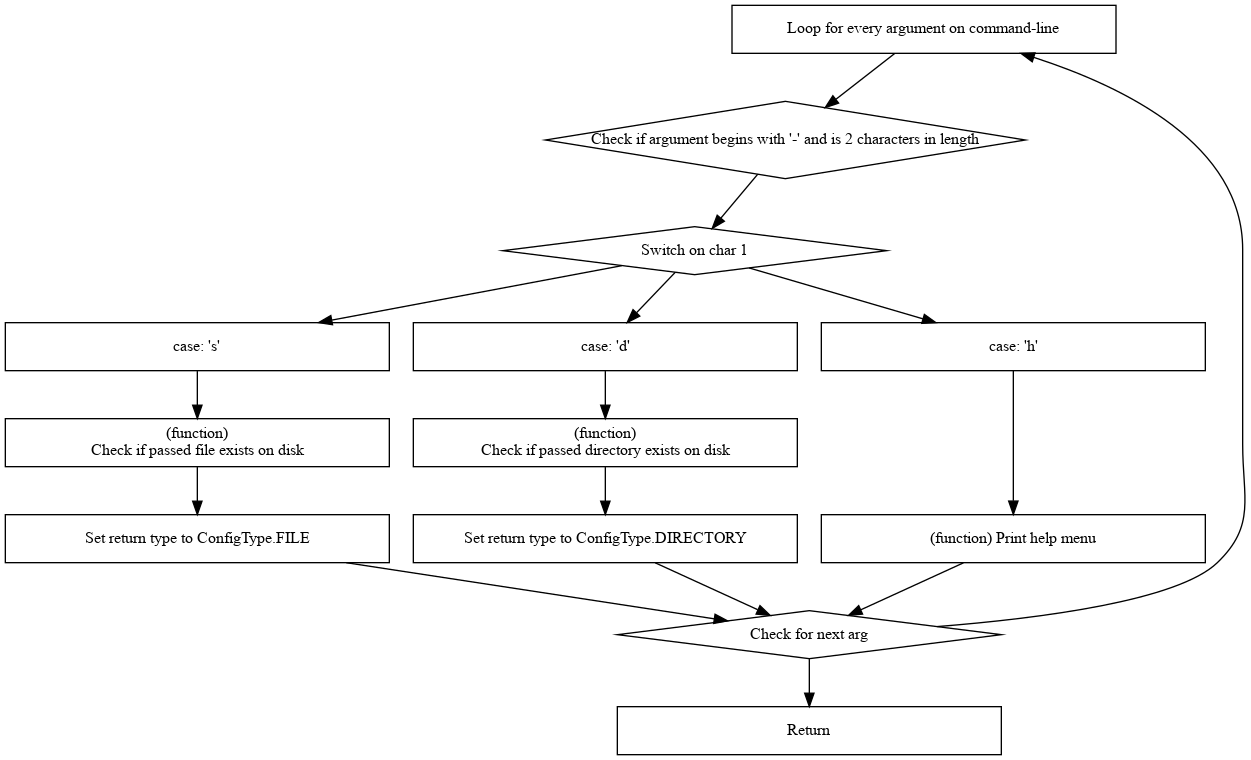


**Figure 3: A Manually Generated Dependency Graph**

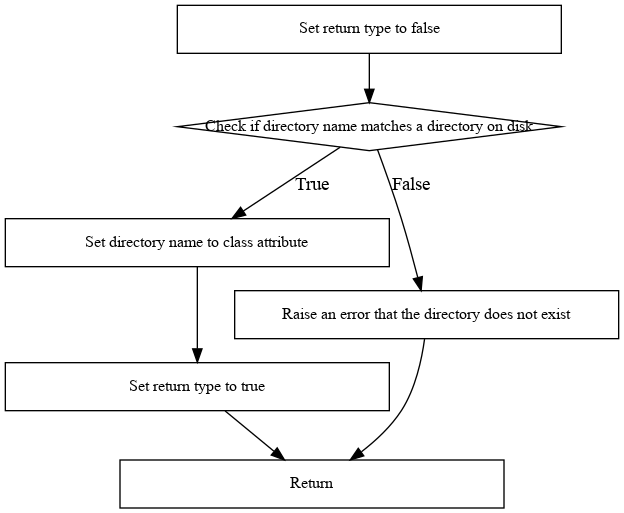
## ***Appendix C: Module Diagrams***

****

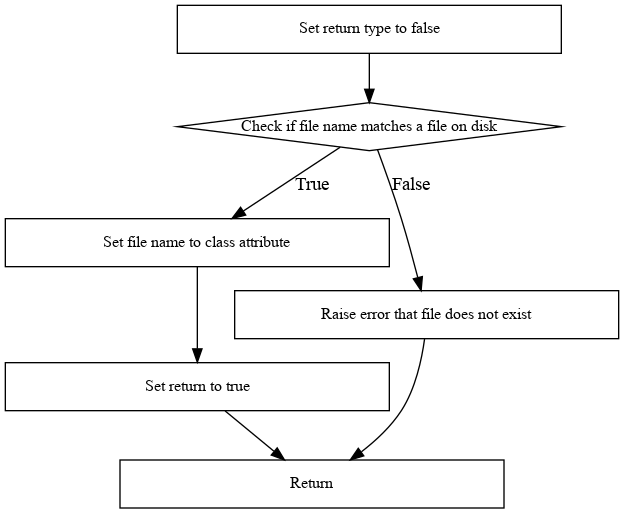
**Figure 4: Manager**

****

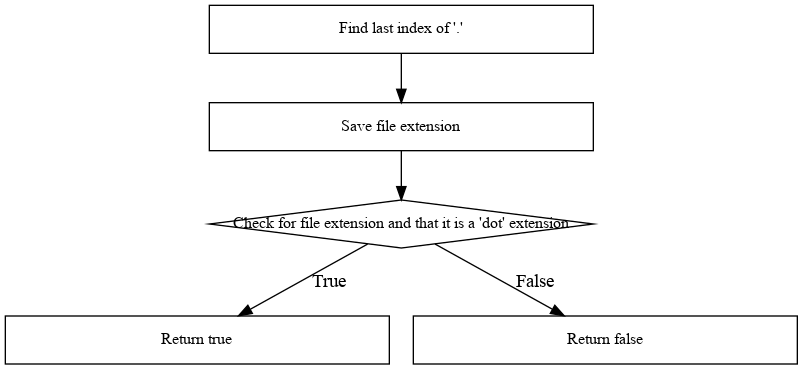
**Figure 5: Configurator, Command Arguments**

****

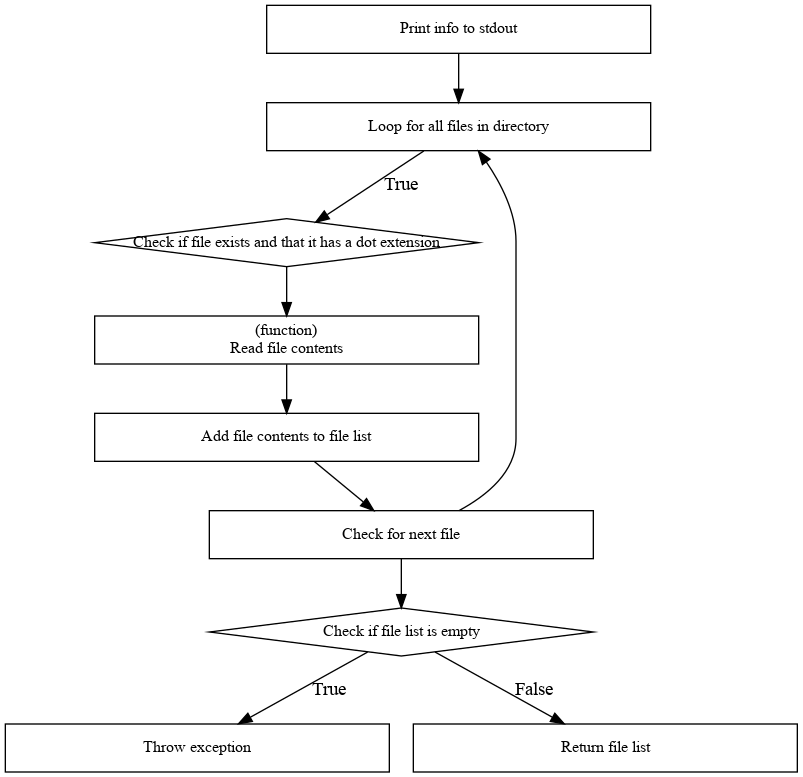
#### **Figure 6: Configurator, Process Directory**

****

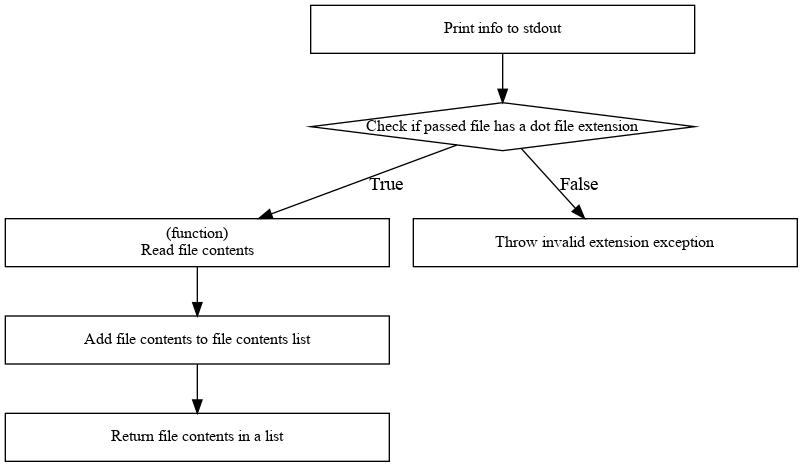
### **Figure 7: Configurator, Process Single File**

****

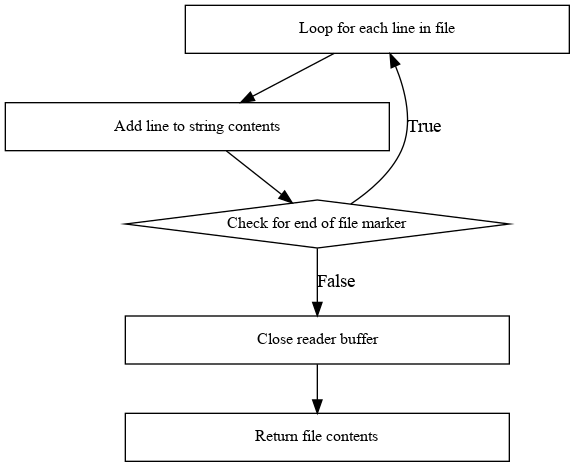
#### **Figure 8: Reader, Is Dot File**

****

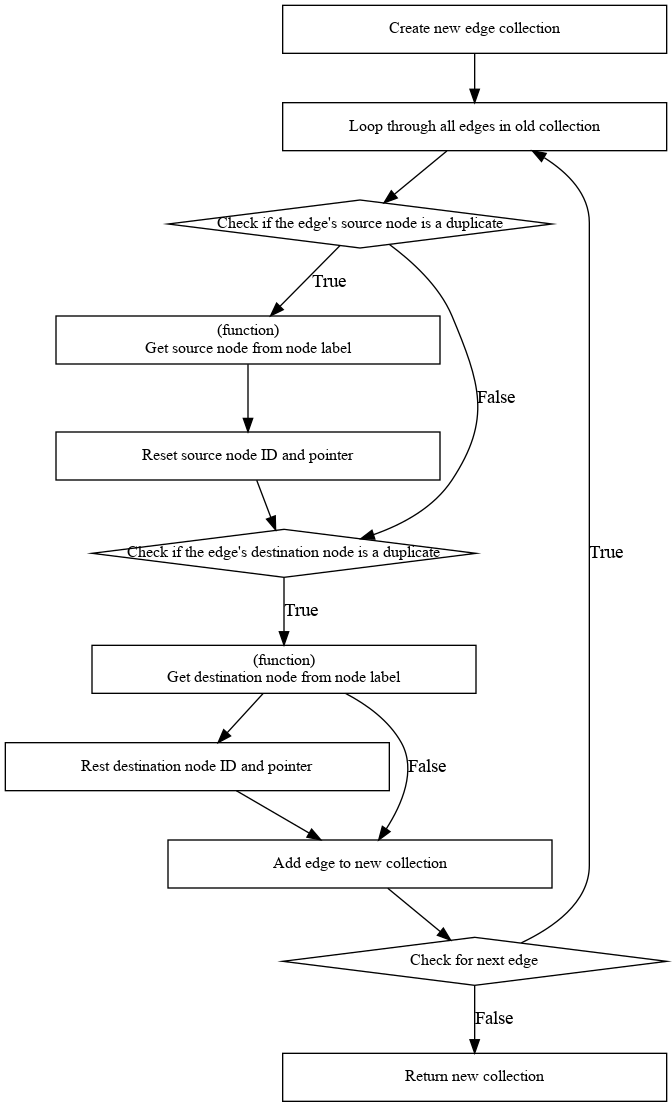
#### **Figure 9: Configurator, Read Directory**

****

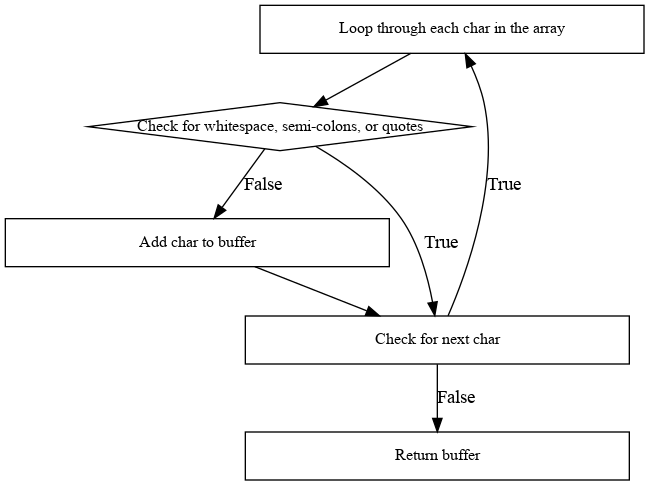
#### **Figure 10: Reader, Read Single File**

****

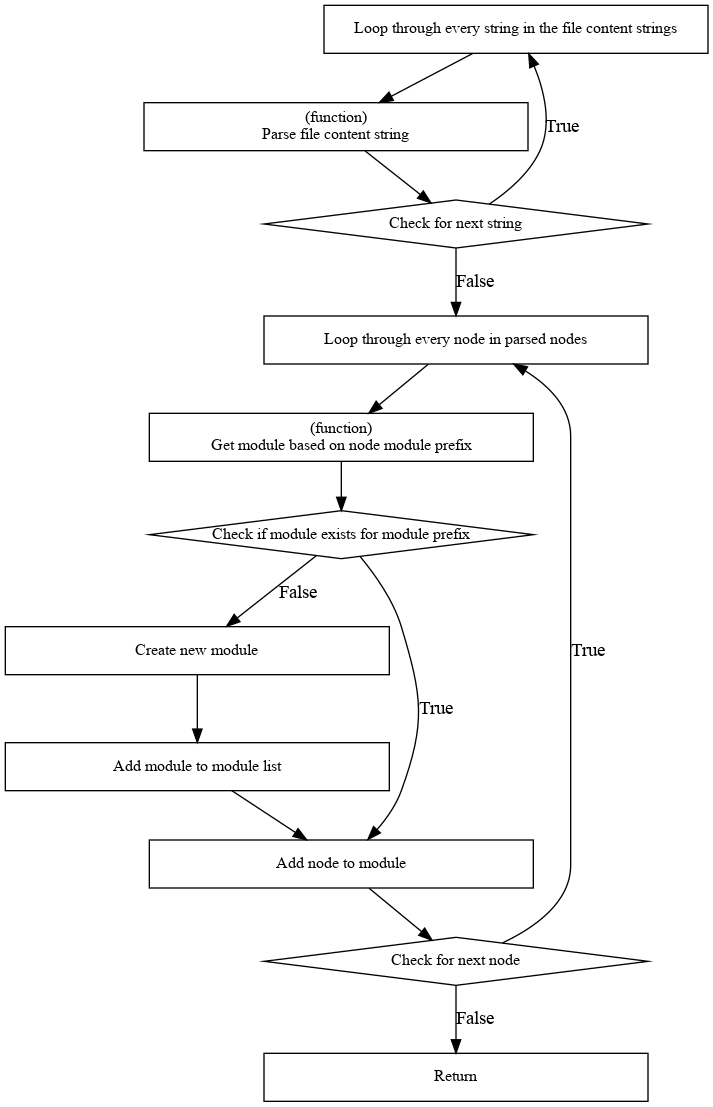
### **Figure 11: Reader, Read**

****

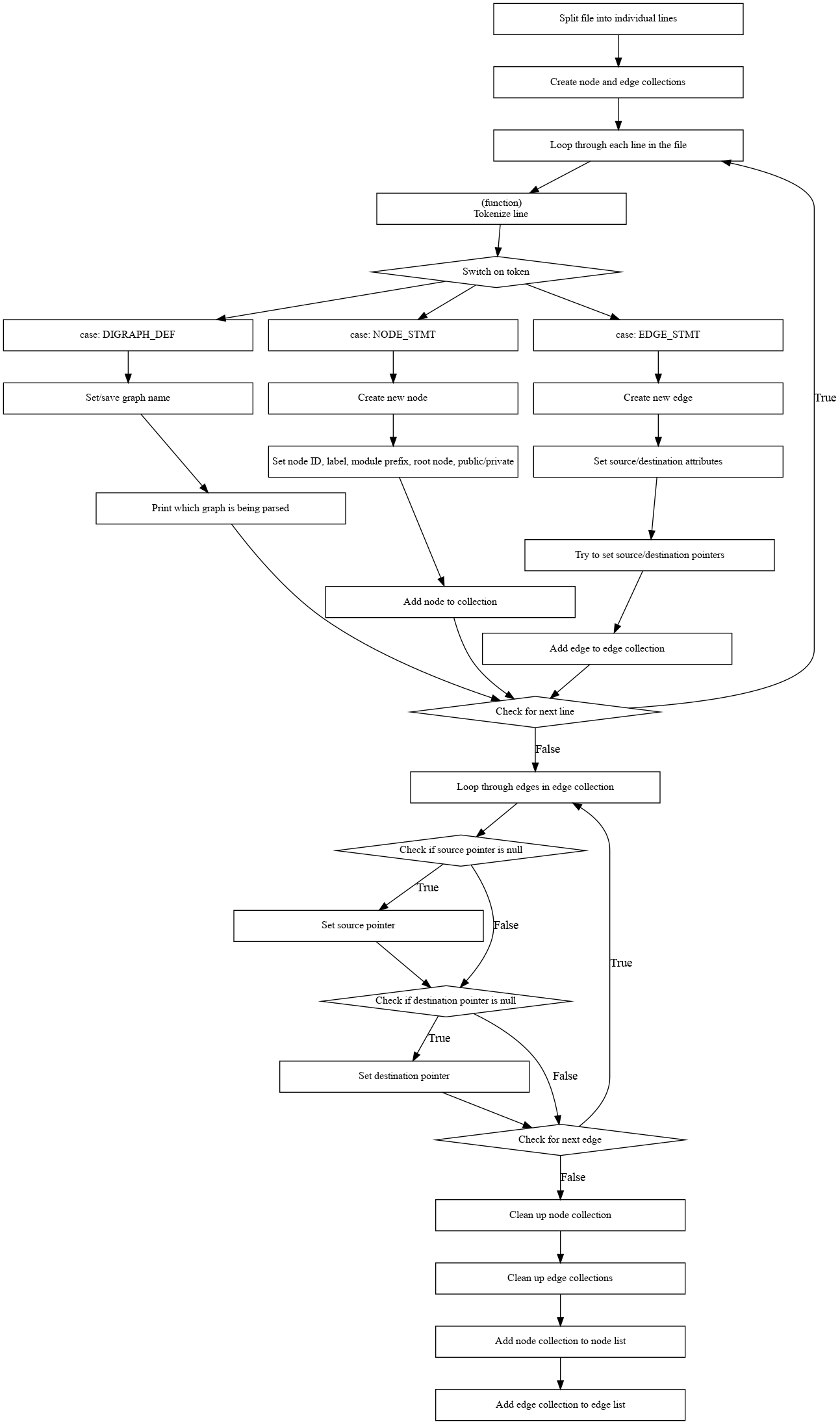
**Figure 12: Parser, Clean Edge Collection**

****

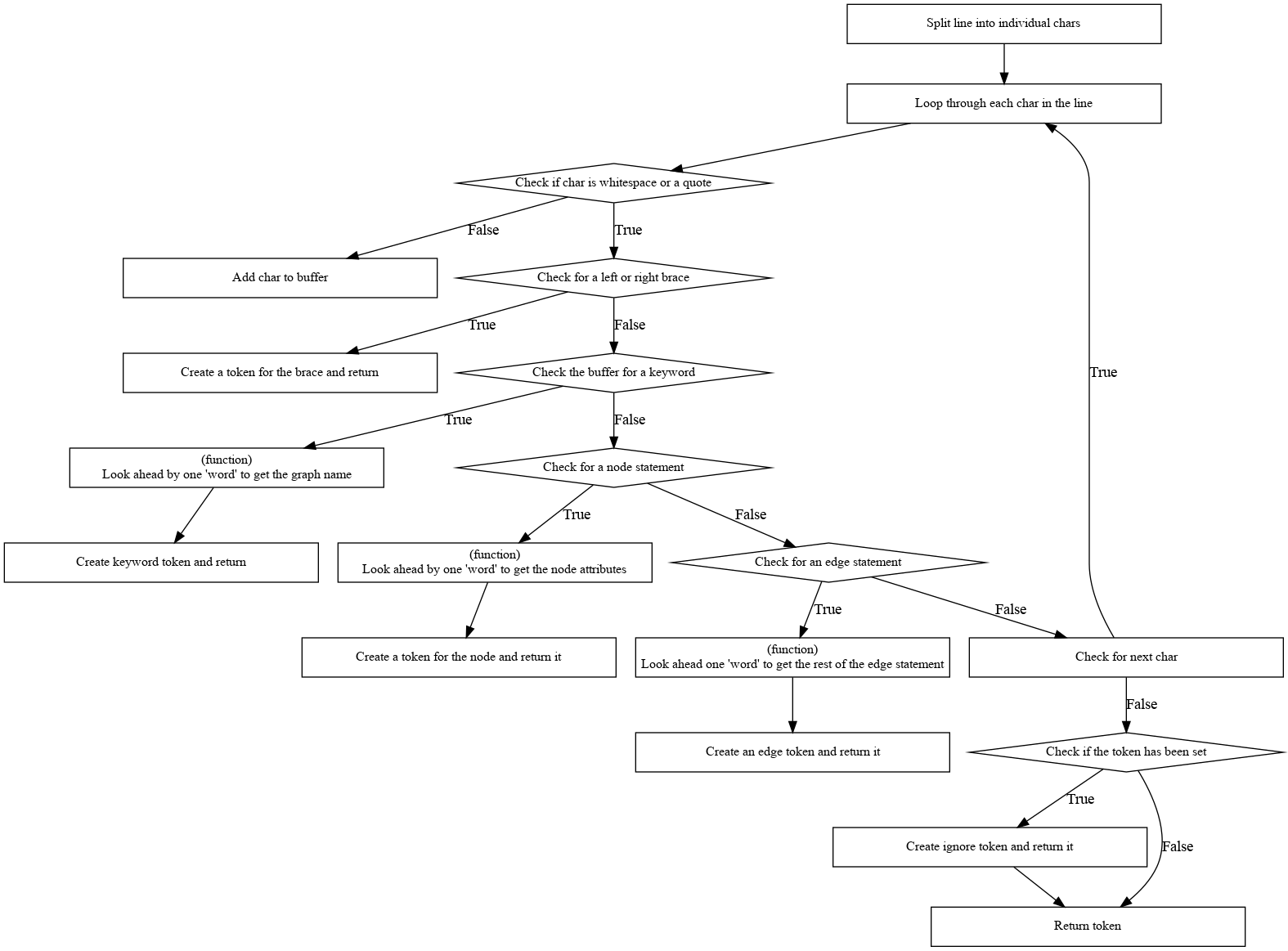
#### **Figure 13: Parser, Look Ahead**

****

#### **Figure 14: Parser, Parse All**

****

**Figure 15: Parser, Parse Single**

****

**Figure 16: Parser, Tokenize**