System Design Document For Dependency Graph

Team Members:

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

Version/Author	Date
1.0	10.1.20

Table of Contents

Overview

- The System Design Document describes the system requirements, operating environment, system and subsystem architecture, files and database design, input formats, output layouts, 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 reduce the amount of work necessary to refractor modules.

1.2 Project Executive Summary

- The project creates a visual representation of a software project through a dependency graph between modules.

1.2.1 System Overview

- The system takes in generated .dot files (from Doxygen) and then reads/parses them to find the different 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 Dot to create an actual dependency graph image (png). The system is required to only take in applicable files.

1.2.2 Design Constraints

- The assumptions made by the project team in developing the system design include that the user has generated the Doxygen .dot file(s) and that the user will run with Windows. As long as these assumptions are followed through, there are no conflicts between systems.

1.2.3 Future Contingencies

- Future contingencies include the addition of a graphical user interface (GUI). The GUI will be used to facilitate a user's interaction with the system. If the GUI cannot be 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/hardware and internal communications design, external interfaces, and the system integrity controls.

1.4 Project References

- Product Backlog - 9.8.20
- Product Vision Statement - 9.15.20
- System Requirements Specification - 9.25.20

1.5 Glossary

- Doxygen - a documentation generator, a tool for writing software reference documentation
- Dot - a graph generator, a tool for processing text files with the .dot extension and converting them to PNG images
- PNG - portable network graphic
- .dot - . template files created by Microsoft Word to have pre-formatted settings for generation of further DOC or DOCX files

2. System Architecture

- The system is able to take in applicable .dot files to understand the relationship between modules. Once that is produced, it can then develop a dependency graph for the user to better understand the relationships within their system.

2.1 System Hardware Architecture

- This system requires no hardware.

2.2 System Software Architecture

- This system is programmed in Java and has various classes. The Manager class acts as an intermediary system that passes along necessary data for lower classes to handle. The Configurator class takes in the file or directory location and ensures that these paths exist. The Configurator then passes these along to the Reader class that then checks to see if these files are valid .dot files and converts their contents into strings which are added to a list. The Parser class then processes this list to figure out the relationships between modules with the help of the Graph Manipulator class. This information is then given to the Writer class to output reformatted .dot files that will be used to create the images of the dependency graph.

2.3 Internal Communications Architecture

- The system utilizes the Manager class to be a major communication between all of the various classes. When the program is run, each class depends on the class before it to run because it passes the appropriate information that is needed. Just as shown in the Use Case Diagram in the System Requirements Specification Document, at a high level, the program sequence can be viewed sequentially (predictably from one class to the next), but the Manager will serve as an intermediary between classes so that they do not pass data directly between the two.

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 input command line arguments to control the program. There are three main commands the system may process, but it can only process one at a time. The syntax for the usage would be:
- -h : help functions
- -s : option to process single file with the path to file provided as argument
- -d : option to process directory with path to directory provided
- A help function can be invoked by the system when the user inputs a -h command. The user can also provide a single input file or a single directory location following the command -s and -d respectively. The program processes these arguments, and if they exist and are valid, stores the location for the next class.

3.2 Outputs

- The program will output a text file formatted according to the Dot language specification (see Appendix A), which will then be processed by the Dot program and converted into an image file with a graphical representation of the dependencies between graphs that were input. The generated image will conform to the following format: [INSERT MANUALLY GENERATED GRAPH HERE]

4. Detailed Design

4.1 Hardware Detailed Design

- C-Dependency Graph is a software project; there is no hardware.

4.2 Software Detailed Design

- At a high level, the program flow can be viewed sequentially--that is, 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.

- ***Manager*** - will serve as an intermediary between the modules. 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.
- ***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. File contents are converted to a string. A file to a string is a one-to-one relationship. These strings are collected into a list passed (through the Manager) to the Parser.
- ***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 object into a list of graph objects. The list of graph objects is the output of the Parser, and it will be passed (through the Manager) to the Manipulator to be cleaned up.
- ***Graph Manipulator*** - The Manipulator requires all of the graph objects at once, hence the Parser collecting them into a list. It removes connections within a graph object and adds connections between functions in separate modules (not functions inside the module).
- ***Writer*** - The writer takes the final modified graph objects and writes a file which Dot can create a visual image of. This is the output to the user.

| Module | In | In Data | Out | Out Data | | :----: | :----: | :----: | :----: | :----: | | Configurator | Command line argument | Args | Location of DOT file(s) | String | | Reader | Location of DOT file(s) | String | Content of file(s) | List< String> | | Parser | Single file's Contents | String | Graph Object | Object | | Graph Manipulator | A collection of graph objects | Objects | A collection of modified graph objects | Objects | | Writer | The final collection of graph objects | Objects | File for Dot to create imae | File | * *"Objects" refers to developer created Java objects representing needed details of the graph

4.3 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, which contains the ConfigReturnType enumerator). Otherwise, communication from within modules will be to the user in standard out through Java's built-in logging functions.

5. External Interfaces

- The external interface, that is outside of the scope of the system and under development, is the graphical visualization program Dot. Dot is a program that takes in the text files (.dot) that will create the png image of the relationship between modules.

5.1 Interface Architecture

- Dot is contained within its own binary file. The interactions in our program with the Dot API will be primarily through loading and running the dot binary on the text (.dot) file that the program (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 by the program'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 by the Manager, the user will be alerted of the error, be provided with any additional error information, and the program will exit. [Program Flowchart Here (?)]

6. System Integrity Controls

- All files handled by this program will make copies of the contents in those files and will never be modifying 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