# **Test Plan**

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

***Team Members:***

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

|  |  |
| --- | --- |
| Version | Date |
| 1.0 | 11.10.20 |
| 2.0 | 11.30.20 |

**1. Introduction**

#### **1.1 Purpose**

This document is a test plan for C Dependency Graph System Testing, produced by the Scrum team, doubling as the System Testing team. It describes the testing strategy and approach to testing the team will use to verify that the application meets the established requirements prior to release.

#### **1.2 Objectives**

* Meets the requirements and specifications laid out by the System Requirements Specification.
* Supports the intended business functions and achieves the required standards.
* Satisfies the Entrance Criteria for User Acceptance Testing.

### **2. Functional Scope**

Refer to the detailed module diagrams in Appendix C of the C Dependency Graph Software Design Specification. The Modules within scope of testing for the C Dependency Graph System Testing are mentioned in the document attached in the following path:

* Configurator
* Reader
* Lexer
* Parser
* Writer

### **3. Overall Strategy and Approach**

#### **3.1 Testing Strategy**

C Dependency Graph System Testing will include testing of all functionalities that are in scope (refer to the Functional Scope section) identified. System testing activities will include the testing of new functionalities, modified functionalities, screen level validations, workflows, functionality access, and testing of internal & external interfaces.

#### **3.2 System Testing Entrance Criteria**

In order to start system testing, certain requirements must be met for testing readiness. In testing the modules (refer to the Function Scope section) the readiness can be classified by:

* A story card has been implemented
* The demo criteria is believed by the developer to be complete enough for testing

#### **3.3 Testing Types**

##### **3.3.1 Usability Testing**

Because this is a command line-based user interface and not a graphical user interface, usability testing will be conducted through the command line and manual verification. The usability of the command line will be judged by how stable the program is with various inputs or arguments, how many arguments the program can accept at once, and how informative errors are to the user. The program also has command that produces a help menu which will be judged the same way a user manual would be.

##### **3.3.2 Functional Testing**

The objective of these tests is to ensure that each element of the component meets the functional requirements of the Customer/Product Owner as outlined in the:

* Functional Requirements
* Program design
* Change requests/feedback from the Customer and Product Owner

### **4. Execution Plan**

The execution plan that follows details the test cases to be executed. The Execution plan ensures that all of the requirements are covered, as well as accommodate some changes if necessary. All of the test cases are arranged in a logical order depending upon their inter dependency. Because the modules of this program are in a sequential flow, the tests also often require sequential completion.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Module** | **No.** | **Test Case Description** | **Action/Input** | **Expected Response** |
| Config | 1.1 | Use the -s command line argument | Command line -> -s <path> | Graph image |
| Config | 1.2 | Use the -d command line argument | Command line -> -d <path> | Graph image |
| Config | 1.3 | Use the -h command line argument | Command line -> -h <path> | Help menu |
| Config | 1.4 | Use the -v command line argument | Command line -> -v <path> | Log level is set |
| Config | 1.5 | Use the -o command line argument | Command line -> -o <path> | Specific output file |
| Reader | 2.1 | Process single file | Command line -> -s <path> | Returns string |
| Reader | 2.2 | Process directory | Command line -> -d <path> | Returns strings |
| Reader | 2.3 | Attempt bad file path | Command line -> -s <path> | Alert user |
| Reader | 2.4 | Attempt bad directory | Command line -> -d <path> | Alert user |
| Lexer | 3.1 | Create digraph definition token | Input file line to Lexer | Output token |
| Lexer | 3.2 | Create node statement token | Input file line to Lexer | Output token |
| Lexer | 3.3 | Create edge statement token | Input file line to Lexer | Output token |
| Lexer | 3.4 | Ignore other lines | Input file line to Lexer | Output token |
| Parser | 4.1 | Turn node token into Node object | Input token to Parser | Output graph object |
| Parser | 4.2 | Turn module token into Module object | Input token to Parser | Output graph object |
| Parser | 4.3 | Turn edge token into Edge object | Input token to Parser | Output graph object |
| Parser | 4.4 | Check grouping of Nodes into Modules | Run program on directory | Output graph image shows grouping |
| Parser | 4.5 | Check removal of internal connections | List of graph files | Output graph image shows no connections in a module |
| Parser | 4.6 | Check grouping of private/public functions | List of graphs/list of nodes | Output graph image shows grouping |
| Parser | 4.7 | Check removal of duplicate nodes | List of graph files | Internally checked |
| Writer | 5.1 | Check writing of Dot file | Input list of Modules, Edges | DOT file |
| Writer | 5.2 | View generation of graph image | Input list of Modules, Edges. Call draw graph function | Image file (e.g. PNG or JPG) |
| Misc | 6.1 | View JavaDoc | Access JavaDoc page | Accessible and complete |
| Misc | 6.2 | View logger levels | Command line -> -s <0-4>, run program on files | Command line output varies |

### **5. Traceability Matrix & Defect Tracking**

#### **5.1 Traceability Matrix**

This traceability matrix is a tool for tracing requirements to test cases and vice versa. For details related to the listed requirement, refer to the C Dependency Graph System Requirements Specification. For details related to the listed test case, refer to Section 4, Execution Plan.

|  |  |
| --- | --- |
| **Requirement** | **Test Case** |
| FR1 | 1.1, 2.1, 5.1 |
| FR2 | 1.2, 2.2, 5.1 |
| FR3 | 1.2 |
| FR4 | 1.5 |
| FR5 | 1.5 |
| FR6 | 4.6 |
| FR7 | 4.3, 4.4, 4.5 |
| FR9 | 5.2 |
| IR1 | 1.1, 1.2, 1.3, 1.4, 1.5 |
| IR2 | 4.4, 4.5, 4.6, 4.7, 5.2 |
| UHR1 | 1.3 |
| UHR2 | 2.3 |
| UHR3 | 2.4 |
| DoR1 | 6.1 |
| DR1 | 4.1, 4.2, 4.3 |
| DR2 | 5.1 |
| SR1 | 6.2 |
| SR2 | 2.1, 2.2 |
| QA1 | 2.1, 2.2 |

### **6. Environment**

The testing consists mainly of two methods: Manual and Junit. Manual testing is done by running the program through the user interface (command line) and visually inspecting and verifying results. The Junit testing environment is handled by Gradle.

### **7. Assumptions**

* The tests are performed using Dot file(s) generated by Doxygen.
* Junit test cases are run using Gradle or Eclipse.

### **8. Risks and Contingencies**

This test plan is contingent upon the Dot grammar language remaining as it was defined as of writing this. If this were to change, the Lexer and Parser would likely fail to interpret the lines of the given Dot files. This is of low risk because it is unlikely that the language structure would change.

The program and it’s testing is also contingent on the naming convention for public/private functions, which is expected to be upper case prefix for public functions and lower case prefix for private functions. Additionally, the naming conventions for functions is expected to be XXX\_ \*. where the XXX stands for the module prefix it belongs to. Specifically, the risk is in the Parser breaking.