DITAA Final Report
Version 1.0
4 December 2020

Kevin Porter
Adam Shiveley

Overview

Contents

Overview

This document describes the work done by Team 2 for the CS-7140 project of the Fall 2020 semester. It consists of five primary sections representing a standard waterfall approach to software development: requirements, specification, design, implementation, and testing. This final report represents the cumulative effort of the team for the semester.

The requirements section includes a description of the source project which was the base of the work done for this project. For the purposes of this semester, the team modified and expanded existing capabilities of an open source software module written in Java.

1. Requirements

1.1 Introduction

1.1.1 Purpose

The purpose of this section is to present a detailed description of a proposed program that will convert ASCII art in documents to diagrams. The program will be titled "Diagrams Through ASCII Art," or DITAA for short. The use of ASCII art in the field of computer science ranges for early text based games from the 1980s to acting as a "Maker's Mark" for code authors. While the use of ASCII art is generally fun to create, the only known method to easily save the ASCII art is to copy and paste the art into a new file. The proposed program will leverage the use of modern computing to enable users to create diagrams from embedded ASCII. These diagrams will be saved as image format files that can be easily printed or loaded into word editing programs for future use. [1]

1.1.2 Scope of Project

This software system will be a Java-based program executed from the command line. The program will execute as automated as possible, thus reducing input burden on the user.

1.1.3 Glossary

The DITAA (DIagrams Through ASCII Art) program is a command line utility which provides functionality to convert ASCII diagrams into images.

The GUI is the Graphical User Interface with which the user will interact with the DITAA program.

Mentions to a '*.jar' file are in reference to Java archive files which can be a standalone executable final product for a Java program. This is the planned product format for the executable to run the DITAA GUI.

1.1.4 References

This document refers to the DITAA guide on Github as well as help documentation in the repository for functionality descriptions and available options.

1.1.5 Overview of Section

The next chapter, the Overall Description section, of this document gives an overview of the functionality of the product. It describes the informal requirements and is used to establish a context for the technical requirements specification in the next chapter.

The third chapter, Requirements Specification section, of this document is written primarily for the developers and describes in technical terms the details of the functionality of the product.

Both sections of the document describe the same software product in its entirety, but are intended for different audiences and thus use different language.

1.2 Overall Description

1.2.1 System Environment

The DITAA program has minimal user input. The only requirement from the user is to have an ASCII text file ready to be converted. The user also has options to select the encoding, input file type, and some output configurations regarding file output and image style.

1.2.2 Functional Requirements Specification

This section outlines the use cases for both the GUI and command line version.

1.2.2.1 Command Line Use Case

Use case: Execute Via Command line Brief Description

The User executes the program and types all of the user required options on the command line.

Initial Step-By-Step Description

Before this step can be executed, the user needs to have identified a file with ASCII art to convert.

1. The User locates the ASCII art file (it can be user created, found online, or art buried in a text file.
2. The user opens a shell.
3. The user runs the program with the selected options, passing the text file as an argument.
4. The program processes the user defined file and creates the object file in the same directory.
5. The user is notified of program completion.

1.2.2.2 GUI Use Case

Use case: Execute Via GUI Executable *.jar

This user case allows for the Graphical User Interface (GUI) component of the program to be executed. The GUI is basically a wrapper around the main program, which facilitates ease of program execution for the user.

Brief Description

The user executes a *.jar file by double clicking on it. The program opens to a screen which allows the user to decide which parameters to use and which file to scan looking for ASCII art.

Initial Step-By-Step Description

1. The user double clicks the *.jar file
2. The GUI opens and displays program options to the user.
3. The user chooses the file with the ASCII art from a button click.
4. The user chooses which format to save the file as (*.png or *.jpg).
5. The user presses the Execute button.
6. The program runs.
7. Program notifies the user it is completed.
8. The user can repeat the steps above to run another ASCII art file.

1.2.3 User Characteristics

The user is expected to be able to operate a computer. The user is also expected to be able to operate a common GUI with push buttons and file browser dialogs. The user is expected to understand common file formats and text encodings.

1.2.4 Non-Functional Requirements

The DITAA GUI program will operate on the user's computer. It will be a *.jar file and be easily executed by double clicking the file. The data will be saved in the user chosen location through the GUI.

2. Requirements Specification

2.1 External Interface Requirements

The DITAA program operates on external files. The program also depends on user input from the command line or a GUI interface.

2.2 Functional Requirements

The Logical Structure of the Data is contained in Section 3.3.1.

2.2.1 Command Line

Use Case Name	Command Line
Trigger	User presses the "Run" button
Precondition	The User has an ASCII Art file to convert
Basic Path	1. The User types "java -jar Team_2_CS_7140_Group_Project" 2. The User types the path and filename to be converted. 3. The User enters optional parameters. 4. The User presses "Enter"
Alternative Paths	The User can choose to quit the program without converting text to an image.
Postcondition	The file is verified to exist and the conversion is complete
Exception Paths	The User may abort the program at any time.
Other	None

2.2.2 Graphical User Interface

Use Case Name	GUI
Trigger	The double clicks the *.jar file
Precondition	The user has an ASCII Art file to convert.
Basic Path	1. The user selects the file to be converted. 2. The user chooses which save format and output name 3. The user presses "Execute" 4. Program notifies User is has completed 5. The User can run the program again, if desired. 6. The User can exit the program.
Alternative Paths	The user may abort the program without converting text files to images. The user may also continue to repeat the process any number of times.
Postcondition	The file is verified to exist and the conversion is complete.
Exception Paths	The user may abort the program at any time.
Other	None

2.3 Detailed Non-Functional Requirements

2.3.1 Logical Flow of DITAA Graphical User Interface

The logical structure of the DITAA program is given below.

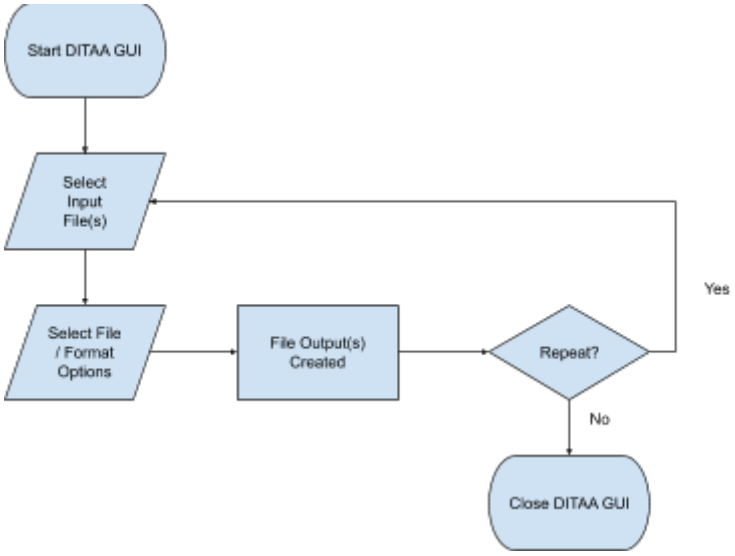


Figure 1 - Logical Flow of DITAA Graphical User Interface

The DITAA program will output a *.png or *.jpg file containing a diagram of the chosen ASCII art.

2.3.2 Security

The program will attempt to be developed with cyber security in mind. All loops will contain bounds, all memory will be properly managed, and error messages to the user will be easy to understand. The GUI interface will be tested for proper input into the requested fields. The program should not crash or hinder access to the user's machine nor access, store, or transmit any unauthorized data.

3. Specification

3.1 Introduction

3.1.1 Purpose

The purpose of this section is to develop the specification for the Designs Through ASCII Art (DITAA) program. This section contains the team's original specification, along with references to Team 1's specification. The class ultimately selected Team 1's specification as the common specification to which all teams would build their designs, implementations, and testing plans. [2]

3.1.2 Scope of Section

This section will describe the general outline the DITAA program will take. A pseudo code written in a Very High Level Language (VHLL) will explain the general flow of controlling functions of the program. There are other functions that will be used within the program, but those functions are considered helper functions. Helper functions are not controlling functions within a program and will not be discussed.

3.2 Specifications of Design

3.2.1 Very High Level Language Functions

This section describes at a very high level the implementations of the CLI and GUI as described in this team's SRS.

3.2.1.1 CLI Input

```
while user_input_not_enter:
    collect_user_input()
args = string_of_user_args
```

```
filename = user_filename
parsed_args = parse_args(args)
textfile_contents = user_input(filename)
if is_valid(textfile_contents):
    generate_graphics(textfile_contents)
save_diagram(output_file)
```

3.2.1.2 Parse Arguments

```
parse_args(arg_string):
    # pre: list of arguments for program are entered
    # post: list of arguments are parsed parsed_args
    = list()
    for arg in arg_string:
        parsed_args.add(collect_and_validate_arg(arg))
    return parsed_args
```

3.2.1.3 User Input

```
user_input(textfile):
    # pre: user launches CLI with name of textfile
    # post: return contents of textfile
    return contents(textfile)
```

3.2.1.4 Graphical User Interface (GUI)

```
main_menu_dialog():
    # pre: user has executed the .jar file
    # post: user has created zero or more output ASCII files
    while running:
        wait_for_user_input()
```

```
    if select_file:
        infile = file_dialog(true)
    if configure_arguments:
        parsed_args = arguments_dialog()
    if select_outfile:
        outfile = file_dialog(false)
    if run_converter:
        run_converter(infile, outfile, parsed_args)
    if exiting:
        # close window
```

3.2.1.5 File Dialog

```
file_dialog(input_mode):
    # pre: user has opened the file selection dialog
    # post: user has selected a file
    wait_for_user_input()
    if file_selected and user_select("OK"):
        return file_selected.path
```

3.2.1.6 Arguments Dialog

```
arguments_dialog():
    # pre: user has opened the arguments dialog
    # post: user has selected arguments
    wait_for_user_input()
    if user_select("OK"):
        arguments = get_selected_arguments()
```

return arguments

3.2.1.7 User Input

user_input(textfile):

 # pre: user has supplied a textfile name

post: return contents of textfile return

contents(textfile)

3.2.1.8 Run Converter

run_converter(infile, outfile, parsed_args):

pre: user has selected an input file

 # pre: user has configured arguments

 # pre: user has designated an output file

 # post: infile contents have been converted and stored in outfile

infile_content = get_content(infile)

if is_valid(infile_content):

draw_and_save(infile_content, parsed_args)

return

3.2.1.9 Get Content

get_content(file):

 # read and return content from file

3.2.1.10 Valid

is_valid(textfile_content):

 # pre: structure with textfile content

 # post: true/false whether the textfile contains convertible contents

```
textfile_content = get_content(file)
if textfile_content.match(required_pattern):
    return True
else:
    return False
```

3.2.1.11 Draw and Save

```
draw_and_save(textfile_content):
    # pre: textfile_content contains valid ASCII art
    # post: return rendered image of ASCII art
    list_of_objects = find_objects(textfile_content)
    for object in list_of_objects:
        diagram = draw_diagram(object)
    save_diagram(outfile)
```

3.3 Non-Functional Requirements

3.3.1 Code Reduction

The goal is to limit the program to only the required code. Assuming such a program already exist, then the goal of this project would be to use the existing program as a basis upon which to add features and reduce the code base. The overall reduction percentage will not be based on lines of code, but rather subject matter expert opinion's when comparing both code basis. This version of DITTA will focus on condensing looping conditions and implementing an object-oriented programing approach.

3.3.2 Limiting Bugs

The program will strive to achieve limiting common bugs within a program. Such bugs are unbound buffers, not binding printf statements, under and overflows, etc. These bugs will be detected during unit test and eliminated in the final product.

3.4 Testing

The program will be run through a series of tests. Each unit of the test will contain known outputs along with unknown outputs. Mixed in with the known outputs will be cases in which the program should pause and wait for user input. Such cases could contain unknown ASCII values in the file, different file encoding, or different file language to name a few. Should an error occur, the error will be displayed to the user/developer and action taken accordingly.

3.5 Common Specification

As previously mentioned, the common specification to which all teams built their designs, implementation, test plans, and final report is Team 1's specification. That specification can be found at Team 1's repository for this project: https://github.com/CJMenart/CS-7140-2020/blob/master/DITAA%20Project%20Specs.pdf

4. Design

4.1 Introduction

4.1.1 Purpose

The purpose of this section is to describe the overall design for the Designs Through ASCII Art (DITAA) program. The DITAA program has been explained in full in a previous sections of this document.

4.1.2 Scope of Section

This document will describe the general design of the DITAA program. The document will include the overall flow of the program and the Graphical User Interface (GUI) that the program will implore.

4.2 System Architecture

4.2.1 Architectural Design

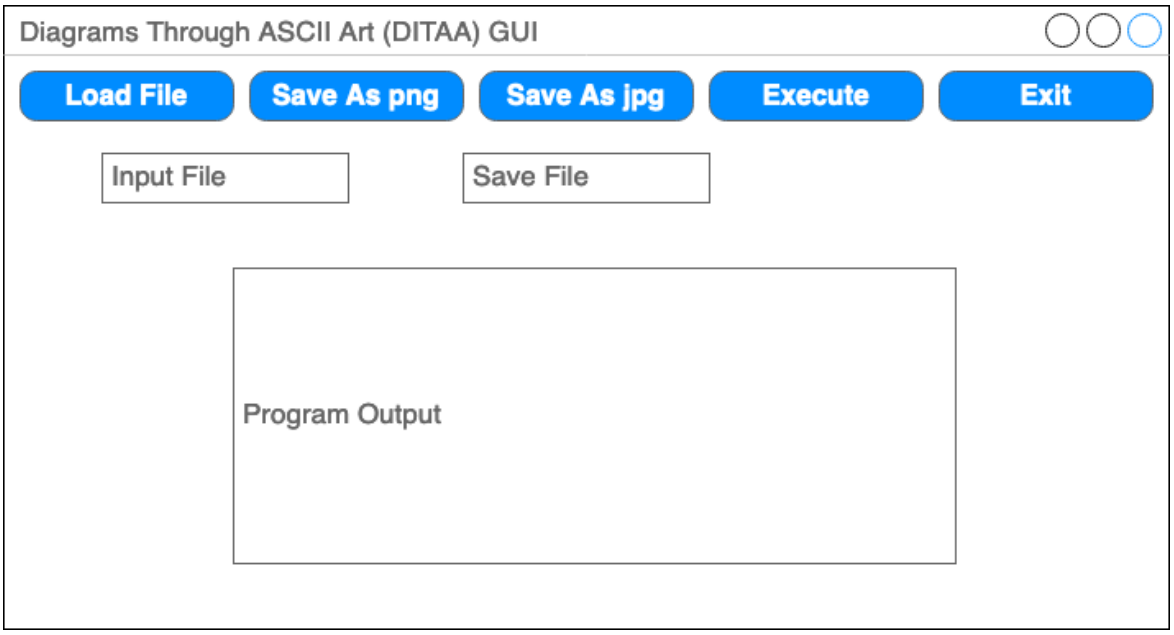
The program will be written in Java version 8. No custom of special plugins will be used. The design goal is to allow the program to operate on any modern computer without installing various need packages and addons.

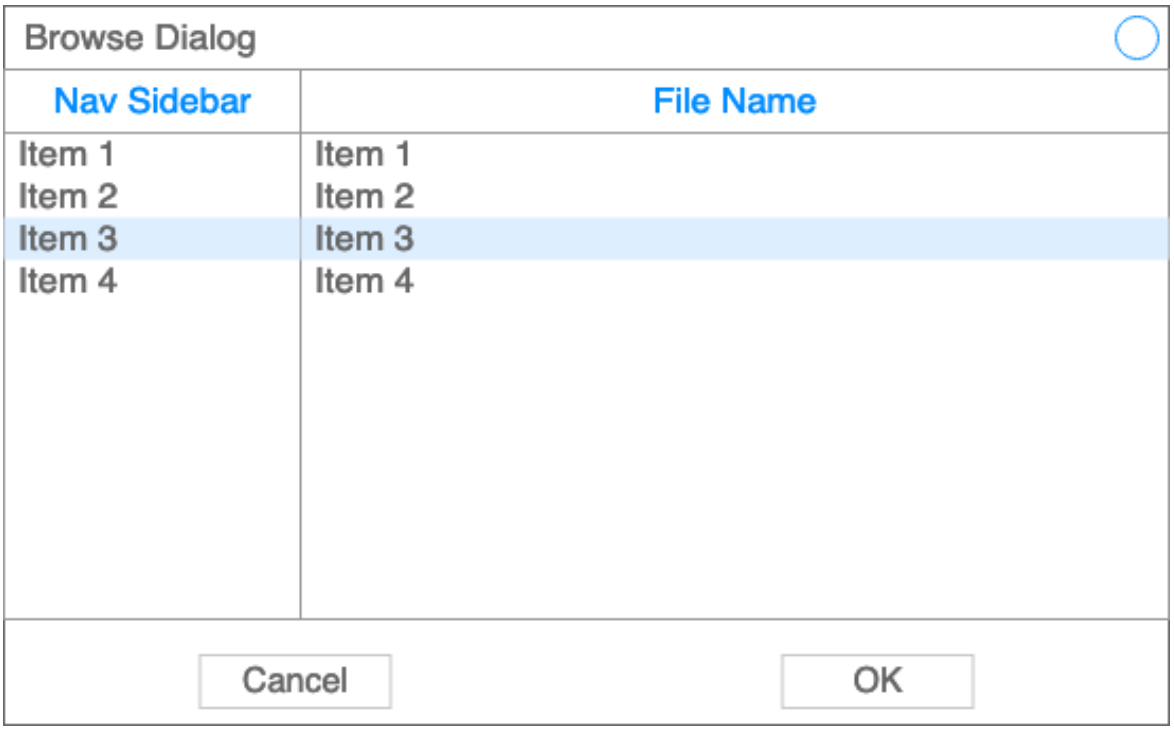
4.2.2 Design Rationale

The DITAA program can be divided into 2 separate parts: User Interface and computational program. The User Interface section is described in detail in Section 3.0 of this document. The computational portion of the program will be described in this section. The general requirements and specifications of the program have been described earlier as noted in Section 1.1 of this document. The program will use the Very High Level Language (VHLL) Pseudo code from the specification document to convert the ASCII art to a graphical representation.

4.3 Human Interface Design

The DITAA program will implore a Graphical User Interface (GUI) to facilitate the ease of user input. The GUI will consist of multiple fields. One field, a text box, will be used in conjunction with a “Browse” button to enable users to select which ASCII file they want to convert. Another field will contain a text box field and a “Browse” button to enable the user to choose where they will save the final result. Wireframe mockups of the GUI elements are given below.



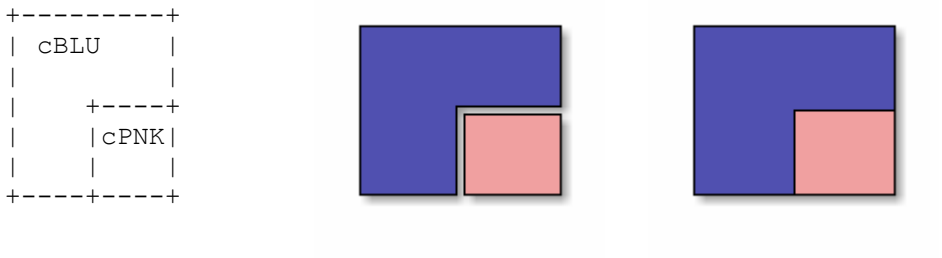


The DITAA program will also facilitate the usage of a command line input feature. This mode of the program will allow a more advanced user to control various elements of the program through command line flags. An example and description of these flags can be found below:

-A,--no-antialias Turns anti-aliasing off.

-d,--debug Renders the debug grid over the resulting image.

-E,--no-separation Prevents the separation of common edges of shapes. You can see the difference below:



Before processing | Common edge separation (default) | No separation (with the -E option)

-e,--encoding <ENCODING> The encoding of the input file.

-h,--html In this case the input is an HTML file. The contents of the <pre class="textdiagram"> tags are rendered as diagrams and saved in the images directory and a new HTML file is produced with the appropriate <img> tags. See the HTML section.

--help Prints usage help.

-o,--overwrite If the filename of the destination image already exists, an alternative name is chosen. If the overwrite option is selected, the image file is instead overwriten.

-r,--round-corners Causes all corners to be rendered as round corners.

-s,--scale <SCALE> A natural number that determines the size of the rendered image. The units are fractions of the default size (2.5 renders 1.5 times bigger than the default).

-S,--no-shadows Turns off the drop-shadow effect.

-t,--tabs <TABS> Tabs are normally interpreted as 8 spaces but it is possible to change that using this option. It is not advisable to use tabs in your diagrams.

-v,--verbose Makes ditaa more verbose.

4.4 Testing

The DITAA program will be tested against known test cases. Those test cases are not yet fully developed.

5. Implementation

5.1 Introduction

5.1.1 Purpose

The purpose of this implementation document is to describe how our team will implement the suggested program called Diagrams Through ASCII Art (DITAA). This is a companion document to the team's existing Requirements, Specifications, and Design documents.

5.1.2 System Overview

The DITAA program will accept standard ASCII art, commonly found embedded within text documents, and convert the text-based art into a graphical representation. ASCII Art was commonly used to embed comments or logos in older programs. The goal of DITAA is to quickly extract the ASCII Art information and render a more human readable format to the user.

5.1.3 Assumptions and Constraints

The authors assume the intended users have a basic understanding of how to operate programs on a modern computer. The authors also assume the users know how to locate and navigate a modern Operating System.

5.1.4 Security and Privacy

The DITTA program should not introduce any security or privacy concerns. It is a standalone executable with no built-in connection to the internet.

5.2 Implementation Support

5.2.1 Hardware and Software

5.2.1.1 Hardware

The DITAA program will operate on any modern computer executing Linux or Windows. While not directly tested, the program should also operate on MacOS based systems, but no support it offered.

5.2.1.2 Software

The DITAA program will be written in Eclipse and compatible with Java version 8.

5.2.1.3 Documentation

A simple Readme and Help document will be included with the program.

5.2.3 Personnel

5.2.3.1 Staffing Requirements

The DITTA program will be developed by the two team members.

5.2.3.2 Training of Implementation Staff

The team members require no specific training, as they are both fully qualified Computer Scientists with years of experience and at least Bachelor level degrees.

5.2.4 Outstanding Issues

Currently, a testing matrix is not yet fully accepted for testing. This does not affect the team's ability to implement our proposed program.

5.2.5 Implementation Impact

When fully implemented, the DITAA program will enable basic and advanced user the ability to quickly parse ASCII Art embedded with text documents. Text documents is defined to be any file readable by a text parsing program. This will enable users to quickly read program's comments about the program. DITAA will also allow for extracting logos designed in ASCII Art embedded in the program for future use.

5.2.6 Performance Monitoring

DITAA is expected to use very little computing power or resources. DITAA's resource management will be monitored using the Operating System's built-in resource monitor.

5.2.7 Configuration Management Interface

Currently only the single version of DITAA will be released. The authors do not plan on providing updates or maintaining the repository after the initial release.

5.3 Implementation Verification and Validation

The DITAA program will be implemented in Java version 8. The program will be tested on Linux using Ubuntu 16.04. Simple test ASCII Art will be used as input into the program to ensure program operation.

5.3.1 Acceptance Criteria

The acceptance Criteria for the DITAA program is known results from test cases. The authors will develop, by hand, expected results from the test cases. When the program is executed, the results will be compared to the truth set for accuracy. If the programmatic results are within an error tolerance of <5%, the DITAA program will have been deemed effective.

6. Testing

6.1 Introduction

6.1.1 Purpose

The purpose of this document is to describe the test plan, test approach and methodology, test cases, and test results for this team's implementation of the design for modifications to the Diagrams Through ASCII Art (DITAA) module. This document contains all the information a reader should need to understand what testing was planned for and accomplished, along with the results of those testing efforts.

6.1.2 Scope

The scope of this section is limited specifically to content related to testing. Any questions with respect to specification, design, requirements, or implementation plans should be referred to those to documents. Additionally, this document does not describe any testing that already existed in the module prior to the team's modifications, nor does it describe testing for any features that existed in the module prior to the team's modifications.

6.2 Test Plan

6.2.1 Testing Environment

All testing for this module has been accomplished using Java version 8. As such, the module should be expected to run successfully on any modern operating system (Windows, Linux, macOS) with the appropriate version of Java. However, all testing has been done on Windows or Linux, as no support is offered for macOS. The program will be officially tested on Linux using Ubuntu 16.04.

6.2.2 Testing Methodology

Testing for this project is divided into two sections: test cases and test results. Test cases are treated as descriptions of tests to be performed. A test case can be validated automatically through unit tests or manually through instrumentation of the module. The testing done for this project is not intended to be exhaustive or bulletproof, but rather a demonstration of the understanding of the amount of rigor and information required to execute a successful and useful testing scenario.

6.3 Test Cases

6.3.1 Command Line Interface

This test case refers to the Command Line Interface Use Case described earlier in this document.

Use Case Step	Description	Passing Result	Actual Result	Success/Fail
1	User generates and locates valid ASCII file	DITAA accepts ASCII file	DITAA accepts ASCII files with a certain format	Success
2	User launches DITAA from the shell	DITAA responds to inputs from the shell	DITAA responds to inputs from the shell	Success

3	User passes arguments to DITAA	DITAA processes arguments according to specification and documentation	DITAA processes arguments according to specification and documentation	Success
4	User retrieves processed output	DITAA processes inputs and generates output in the same directory as the input	DITAA processes inputs and generates a *.png with the same name in the same directory as the input	Success
5	User is notified of program completion	DITAA exits gracefully	DITAA exits gracefully	Success

6.3.2 Graphical User Interface

This test case refers to the Graphical User Interface Use Case described earlier in this document.

Use Case Step	Description	Passing Result	Actual Result	Success/Fail
1	User generates and locates valid ASCII file	DITAA accepts ASCII file	DITAA accepts ASCII file with a certain format	Success
2	User double clicks the *.jar file	DITAA GUI is presented to the user	DITAA GUI is presented to the user	Success
3	User can input optional arguments as mentioned in the companion documents	Input by User are processed by the program	Input by User are processed by the program	Success
4	User chooses the file with the ASCII art from a button click	DITAA presents a browse dialog and accepts a path for an input file	DITAA presents a browse dialog and accepts a path and input file	Success
5	User can choose to save the final result as a *.jpg or *.png by clicking the appropriate button	A *.jpg or *.png with the User specified filename is created in the User specified folder	A *.jpg or *.png with the User specified filename is created in the User specified folder	Success
6	The program is executed	The program executes	The program executes	Success
7	The user can perform another run	The program can be executed again	The program was executed again	Success
8	User exits the program	DITAA exits gracefully and notifies the User	DITAA exits gracefully and notifies the User	Success

6.3.3 Added Color Options

This test case refers to the implementation of the additional color options described in Team 1's specification document, which is the common specification document for all teams' implementations.

Use Case Step	Description	Passing Result	Actual Result	Success/Fail
1	User generates an input ASCII file with new color options described in this document	ASCII file reflects a valid and compliant input file	ASCII file reflects a valid and compliant input file	Success
2	User proceeds through all steps of the CLI Test Case or the GUI Test Case	DITAA produces an output file with the appropriate colored sections as defined in the input file	DITAA produces an output file with the appropriate colored sections as defined in the input file	Success

6.4 Test Results

DITAA was executed from the command line with the following command:

Java -jar ditaa.jar "C:\DITAA\test\Test1.txt"

The original Test1.txt file contents is shown in Figure 1 and the results after processing with the DITAA program is shown in Figure 2.

```
+------+ +-------+
| cGRY | | cRSE  |
|      | |       |
+------+ +-------+
```

Figure 1: File contents for Test1.txt

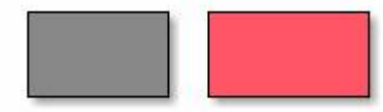


Figure 2: File contents for Test1.jpg

DITAA was also executed using the Graphical User Interface (GUI). A screenshot of the parameters used while operating in GUI mode is shown in Figure 3.

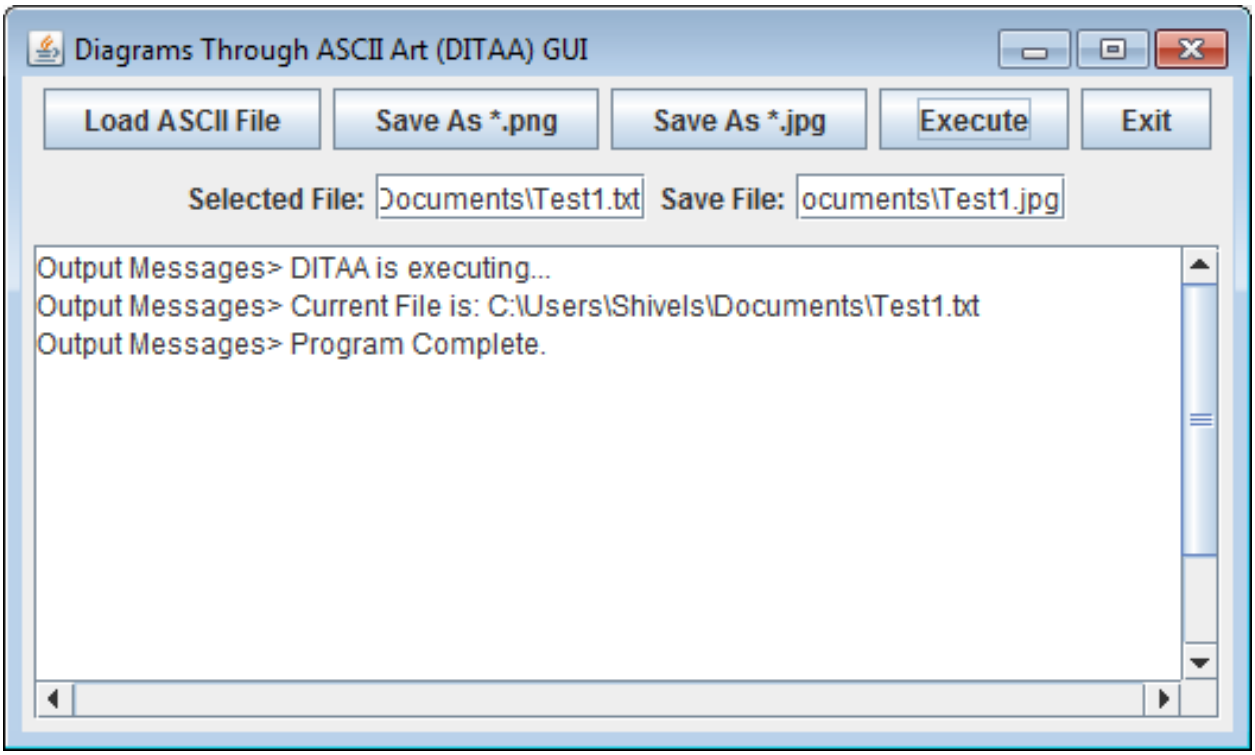


Figure 3: DITAA GUI during execution

As shown in Figure 3, the file path of the ASCII file is shown on the top left textbox and the resulting image after DITAA execution is output to the path in the textbox to the right. The user has the option of saving as either a *.png or *.jpg based on which button the user presses.

Optional arguments can be sent to the program via the command line. These options are described in a companion document. Figures 4 shows the program operating with optional operations.

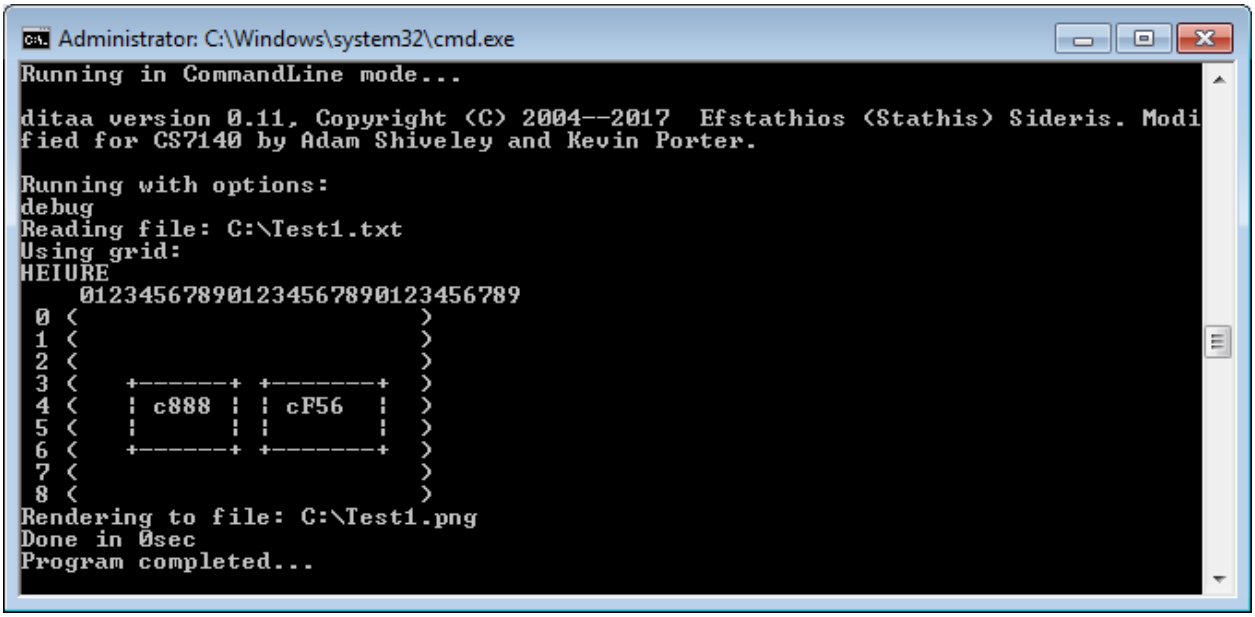


Figure 4: DITAA executed with debug option from the command line

To demonstrate that custom colors work as intended, custom colors were added outside of the colors hardcoded into the program by default. The custom colors are detected based on an 8-bit hex code. For instance, the color White would be “FFF” and the color Black would be “000.” Table 1 shows the custom colors added to the program.

Color	Color Abbreviation	Hex Code
Gray	cGRY	888
Rose	cRSE	F56

Table 1: Custom Color codes added to the DITAA program

The custom colors in table 1 were also hardcoded into the program. Figures 6 and 7 shows a simple object colored in Gray and Rose, respectively.

```
+-----+
|cGRY |
|     |
|     |
+-----+

+-----+
|cRSE |
|     |
|     |
+-----+
```

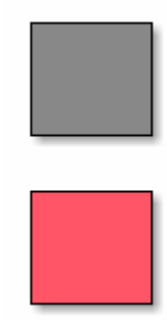


Figure 6: File contents using custom colors in ASCII *Figure 7: DITAA rendering using custom colors Gray (cGRY) and Rose (cRSE)*

7. Conclusion

Throughout this project, the team demonstrated a simple example of a standard waterfall software development pattern. Beginning with requirements, the team determined high level features that were desired for the existing software module. In waterfall, requirements are derived and eventually turned into a specification.

The team wrote a specification based on pseudocode to describe the functions of the desired product. The team then used the specification of another team to begin designing, implementing, and testing the ultimate product. The fact that two specifications can be derived from a similar set of requirements demonstrates that there is not necessarily a single way to create requirements or derive a specification.

From the specification, the team developed a design. Design documentation for the purposes of this project included narrative explaining how the program would function along with graphical wireframe diagrams for the graphical user interface. These are standard design elements that assist a developer in understanding what a specification is describing in more concrete detail.

From the design, the team developed an implementation of the desired program. The implementation represented the fulfilment of the requirements, specification, and design work that had been prepared by the team.

Finally, the team developed test cases for the implementation. These test cases map directly back to the requirements, specification, and design documents to ensure that the implementation correctly fulfills the intent of those documents. By testing according to those higher level requirements, the test framework provides a check against the implementation against potential defects and deficiencies.

In conclusion, this project successfully illustrates that the team was able to develop an entire new feature set to an existing open source module using rigorous waterfall methods.

Appendix A: References

[1] S. Sideris, "stathissideris/ditaa: ditaa is a small command-line utility that can convert diagrams drawn using ascii art ('drawings' that contain characters that resemble lines like | / -), into proper bitmap graphics.," [Online]. Available: https://github.com/stathissideris/ditaa/. [Accessed 2 December 2020].

[2] C. Menart and B. Schneider, "DITAA Specification Document," [Online]. Available: https://github.com/CJMenart/CS-7140-2020/blob/master/DITAA%20Project%20Specs.pdf. [Accessed 2 December 2020].

Appendix B: Team Member Journals

Kevin

- 4 Sept
  - Joined Team 2
- 9 Sept
  - Joined Team 2 Discord
- 12 Sept
  - Downloaded DITAA code to investigate and analyze it
- 14 Sept
  - Agreed with team members that writing a GUI is the best course of action
  - Trying to build the DITAA code, unsuccessful so far
  - Team member Britt dropped the course
- 15 Sept
  - Able to successfully build the DITAA code using IntelliJ
  - Pushed built code to team GitHub
- 16 Sept
  - Adam produced first draft of the requirements document
- 17 Sept
  - Reviewed requirements document some more and offered edits and addition
  - Team submitted requirements document
- 11 Oct
  - Began first draft of the specification document
  - Added pseudocode descriptions of interfaces and logical flow for the program
  - Uploaded first draft for team review and inputs
- 12 Oct
  - Adam reformatted, edited, and submitted the final specification document
- 16 Nov
  - Began reviewing options for design and implementation plans for the project
- 25 Nov
  - Adam produces drafts of design and implementation documents
- 27 Nov
  - Made edits to both documents
  - Added details to design document, including wireframe diagrams and additional information
  - Submitted documents to GitHub
- 28 Nov
  - Received notification that all project materials are due 4 Dec
- 30 Nov
  - Received notification that all projects are to use Team 1's specification document
  - Began producing a test plan and test document along with a final report
- 1 Dec
  - Continued working on the test document and the final report
  - Added new test cases
- 2 Dec
  - Worked with Adam to validate changes to the project and testing and implementation
  - Continued work on the final report
- 3 Dec
  - Worked with Adam to validate changes to the project and testing and implementation
  - Continued work on the final report

- 4 Dec
    - Made modifications to documents and facilitated changes to implementation due to documented mistakes
    - Submitted new versions of final report, test document, and implementation

Adam

- 1 Sept
    - Joined Team 2
- 8 Sept
    - Joined Team 2 Discord

  14 Sept
    - Team agreed to developing a GUI
    - Downloaded DITAA from the official GitHub to start looking at it
    - Tried to build with Netbeans, but failed
    - Tried to build with Eclipse, but failed
    - Britt dropped the course
- 15 Sept
    - Kevin pushed a buildable version of DITAA to team GitHub drop
- 16 Sept
    - Produced first draft of the requirements document
- 17 Sept
    - Team submitted requirements document
- 11 Oct
    - Kevin began first draft of the specification document
    - Kevin pseudocode descriptions of interfaces and logical flow for the program
    - Kevin Uploaded first draft for team review and inputs
- 12 Oct
    - Provided inputs to the specification document
    - Team submitted the specification document
- 22 Oct
    - David drops the course
    - Midterm is taken
- 25 Nov
    - Drafted implementation and design documents
    - Not sure if or when they are due…
- 27 Nov
    - Kevin made edits and submitted them to the GitHub drop
    - I do not have upload permissions since David created the GitHub, but dropped the course
- 28 Nov
    - Notified all project material is due 4 Dec 2020
- 30 Nov
    - Informed that all projects will use Team 1's specification document
    - Began modifying the program to align with Team 1's specifications
    - Kevin begins drafting testing document and final report
    - Added error messages throughout the program
- 1 Dec
    - Continued modifying the program to included separate save buttons for *.png and *.jpg
    - Developed method to ignore file extension if it exists in the save filename
    - Began looking into adding custom colors
    - Added more user information messages throughout the program
- 2 Dec

    - Coordinated with Kevin on the testing report
    - Started testing the program against the testing matrix
    - Started adding additional features to the program
    - Added Help button
- 3 Dec
    - Finalized the testing matrix coordinating with Kevin
    - Removed Help button
    - Added a menu bar which includes Exit option and Help information
    - Added optional arguments (like command line) through the GUI
    - Reviewed the final report
    - Final Report is submitted
- 4 Dec
    - Notified the wrong repo was pulled, resulting in incorrect code
    - Recoded the entire GUI from the ground up
    - Kevin modified the documents with the new GUI
    - New test cases were executed