CS 7140 - Advanced Software Engineering

Fall 2020

Project Design Documentation

Team Members:

Lancius (Lance) Matthieu

Abhishek Pandya

Griffin Mosley

1. System Overview

The overall purpose of DITAA is to convert ASCII designs to a rendered bitmap diagram. Using conversions with lines, symbols, and text, the ASCII text files, or html files, are converted into a diagram displayed in an image format. DITAA allows for a quick transition from ASCII to display diagrams. Our goal is to provide a GUI, a plugin, and code reduction to aid in the transition of ASCII to diagrams.

The purpose of the GUI is to provide a display for the diagrams in an easier way than running DITAA and opening the resulting files. The GUI should be able to display this from running the DITAA program and be able to make changes to files easier. The purpose of the plugin is to provide more utilities to the main DITAA program. The purpose of reducing the code will be to make the code more efficient and be more accessible to future maintenance.

2. Referenced Documents

2.1 DITAA

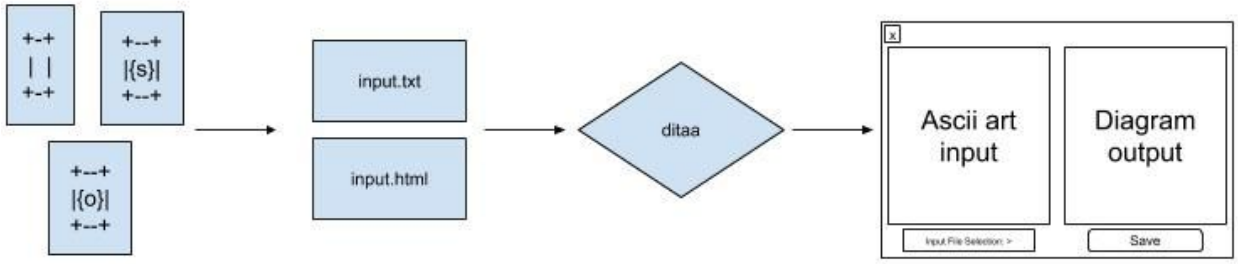
The main bulk of code for this project will be used from DITAA. DITAA allows for this transition between ASCII and rendered bitmap images. DITAA can be found with the link below.
https://github.com/stathissideris/ditaa

2.2 Count Lines of Code (CLOC)

CLOC is the software that the team utilized to measure the number of lines of code in the DITAA project, as a whole and by class. CLOC counts blank lines, comment lines, and physical lines of source code in the Java programming language, as well as others.
http://cloc.sourceforge.net/
https://github.com/AlDanial/cloc

3. Architectural Design

3.1 Concept of Execution



For traditional execution of the DITAA program, please reference the DITAA requirements and specification documentation. There are two additional features in which the team will provide to enhance the user experience with DITAA.

The first feature will be the inclusion of additional color presets, to provide the user a more intuitive experience when selecting colors for their output diagrams. The user can use these new color tags in their ASCII art instead of inputting hexadecimal color values. The second feature that will be included is a graphical user interface (GUI) for those users who do not wish to utilize the command line version of DITAA. Instead of outputting the diagram directly to an image file, the diagram will be displayed in a GUI prior to saving it to the host filesystem. The purpose of this allows for the user to know the end result of the diagram prior to getting the file saved and taking up space on the computer and make the overall program much easier to use.

Another goal is to reduce the overall amount of code by 5% or more. By reducing the overall amount of code, this will allow the code to be more accessible to new developers, be easier to maintain, and be more concise and less cluttered.

3.2 Code Outline

There are three major sections for the code: Command Line, Diagram, and Rendering.

The command line section handles the command line input when running the jar. This allows the user to set flags and define the input and output files to do the conversion. The diagram section is the main bulk of the code. This section creates the shapes from the ASCII input. For example, the diagramShape class creates the diagram shapes by getting the location and path of the shape defined by ASCII. Finally the rendering section applies the transition from ASCII to a diagram, putting what was originally an ASCII formatted file into a picture format (jpg for example).

3.2.1 Command Line

The command line portion handles the major flags run with the DITAA. The main two classes for this section are CommandLineConverter.java and ProcessingOptions.java.

The CommandLineConverterClass, creates the command line options. It provides the abbreviated form of each flag and defines what they are when the help page is run with DITAA. Once the abbreviated forms are created, the command line is traversed to determine which flags are set while also obtaining the input and output file defined. If a flag is provided, a toggle is set and applied to the ProcessingOptions.java object that holds the flag toggles. When the flags need to be applied, they are retrieved from the object with a getter for the flag. The flags are shown below.

```
usage: java -jar ditaa.jar <inpfile> [outfile] [-A] [-d] [-E] [-e
       <ENCODING>] [-h] [--help] [-o] [-r] [-S] [-s <SCALE>] [-t <TABS>]
       [-v]
 -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.
 -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.
    --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,--no-shadows            Turns off the drop-shadow effect.
 -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).
 -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.
```

3.2.2 Diagram

The diagram section is the main bulk of the program. It is the main functionality to do the conversion from ASCII to diagram. The major classes here are Diagram, DiagramShape, TextGrid, and CellSet.

The Diagram class is the major overview of what the output becomes. It contains all the shapes that are converted from ASCII characters to diagram symbols and lines. This class contains all the information on where the shapes are located and what shapes need to be located. The DiagramShape class is where the shapes are created. In this class, the shapes are created by following by using the edge information for the shapes form the ASCII input file. A path is generated using the edges to create the shape according to the symbol defined in the input file.The TextGrid class contains the information about the input file. A grid is created from the file to be able to traverse and gather the characters for transformation. The CellSet class contains the information about the object. It provides a set of cells (points on the grid) that are connected to form a shape.

3.2.3 Rendering

The rendering section provides the ability to render the diagram from an ASCII file to a picture format. The major classes used for this are SVGBuilder and BitmapRenderer.

These two classes do pretty much the same thing, but render the output differently. The SVGBuilder renders the diagram in SVG format,, while the BitmapRenderer renders the diagram as a bitmap. Both classes use the diagram created by using the Diagram to render in the shapes, along with the command line options to do any last changes to the final results.

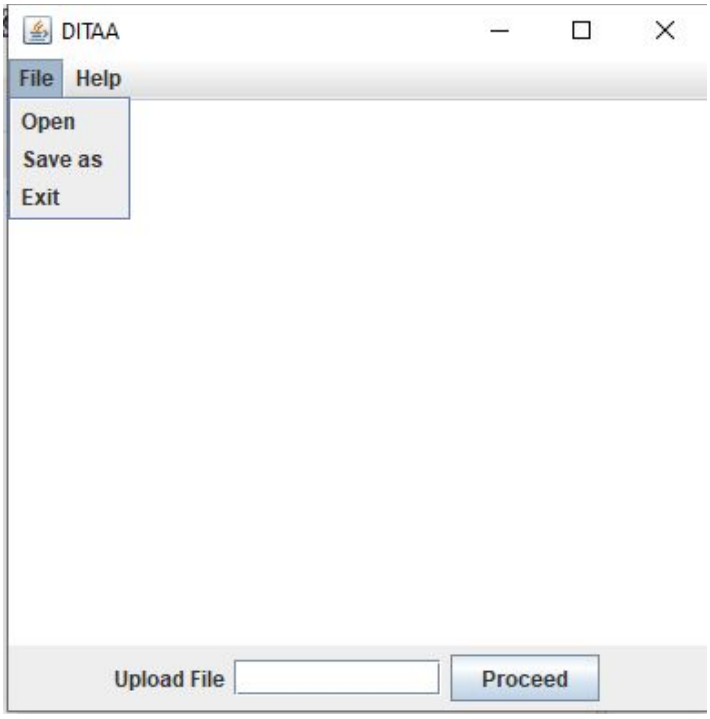
4. Detailed Design

4.1 GUI

For the GUI(Graphical User Interface), we will be using Java Swing which provides platform-independent and lightweight components. There are several packages that create a good graphical interface of the software like JFrame, JMenu, JMenuItem, JPanel, JTextField, etc.

4.1.1 Function

For each functionality we need different functions. For opening the text file which has the ASCII art, we will need openfile() function. This function will open up one particular file which will be selected by the user.
Demo GUI will look like below:



After uploading the file, the user will see the output and can save that file if the user wants. Otherwise the user can exit from the program.

4.2 Preset Colors

4.2.1 Data Structure - humanColorCodes

Additional preset colors will be included in a private HashMap<String, String> that is already included in the original code base in the TextGrid class located in the text package of the project.

The key in the HashMap is the 3 character label of the color (e.g. BLK → Black) and the value with the hexadecimal code for that particular color. The data structure from the original code base can be seen below.

```
private static HashMap<String, String> humanColorCodes = new HashMap<~>();
static {
    humanColorCodes.put( k: "GRE",  v: "9D9");
    humanColorCodes.put( k: "BLU",  v: "55B");
    humanColorCodes.put( k: "PNK",  v: "FAA");
    humanColorCodes.put( k: "RED",  v: "E32");
    humanColorCodes.put( k: "YEL",  v: "FF3");
    humanColorCodes.put( k: "BLK",  v: "000");

}
```

During execution, this data structure will be queried when a color tag is found during parsing of the ASCII art. The associated hexadecimal value with the found tag will be used to color the diagram when rendering.

4.3 Code Reduction

Code reduction will primarily be achieved through the refactoring and/or re-coding of the previous code base. There are a multitude of unused and incomplete functions that make the code complicated to maintain. These functions, as described below, will be removed from the production code base.

The CLOC software analysis tool was used to identify classes in the DITAA program which had the most lines of source code and this was used to prioritize the team's efforts. A breakdown of the classes and the number of lines of blanks, comments, and code can be seen below.

Core Package:

File	Blank	Comment	Code
CommandLineConverter.java	53	44	210
ConfigurationParser.java	27	22	155
ConversionOptions.java	20	75	98
DebugUtils.java	1	19	6
DocBookConverter.java	7	20	40
FileUtils.java	23	30	98
HTMLConverter.java	46	45	147
Pair.java	2	19	9
PerformanceTester.java	13	23	28
ProcessingOptions.java	42	89	112
RenderingOptions.java	28	33	71
Shape3DOrderingComparator.java	7	27	12
ShapeAreaComparator.java	7	27	12

Graphics Package:

File	Blank	Comment	Code
BitmapRenderer.java	88	54	354
CompositeDiagramShape.java	50	35	227
CustomShapeDefinition.java	5	19	55
Diagram.java	148	172	678
DiagramComponent.java	14	23	84
DiagramShape.java	157	151	670
DiagramText.java	32	59	101
FontMeasurer.java	27	30	144
ImageHandler.java	26	28	85
OffScreenSVGRenderer.java	25	23	90
ShapeEdge.java	43	62	152
ShapePoint.java	24	41	75
SVGBuilder.java	136	10	270
SVGRenderer.java	7	3	8

Text Package:

File	Blank	Comment	Code
AbstractCell.java	16	23	84
AbstractionGrid.java	25	40	117
CellSet.java	102	132	453
GridPattern.java	38	107	188
GridPatternGroup.java	82	30	322
StringUtils.java	25	50	94
TextGrid.java	287	181	1318

A few functions that have already been identified for refactoring/re-coding/deletion will be shown below, but will be discussed in further detail in the implementation documentation. Note that the below is ***NOT*** all-inclusive.

4.3.1 TextGrid::replaceAll(char c1, char c2)

```
/**
 * Replace all occurrences of c1 with c2
 *
 * @param c1
 * @param c2
 */
public void replaceAll(char c1, char c2){
    int width = getWidth();
    int height = getHeight();
    for(int yi = 0; yi < height; yi++){
        for(int xi = 0; xi < width; xi++){
            char c = get(xi, yi);
            if(c == c1) set(xi, yi, c2);
        }
    }
}
```

4.3.2 TextGrid::exactlyOneNeighbourIsBoundary(Cell cell)

```
public boolean exactlyOneNeighbourIsBoundary(Cell cell) {
    int howMany = 0;
    if(isBoundary(cell.getNorth())) howMany++;
    if(isBoundary(cell.getSouth())) howMany++;
    if(isBoundary(cell.getEast())) howMany++;
    if(isBoundary(cell.getWest())) howMany++;
    return (howMany == 1);
}
```

4.3.3 TextGrid::seedFill(Cell seed, char newChar)

```
private CellSet seedFill(Cell seed, char newChar){
    CellSet cellsFilled = new CellSet();
    char oldChar = get(seed);

    if(oldChar == newChar) return cellsFilled;
    if(isOutOfBounds(seed)) return cellsFilled;

    Stack<Cell> stack = new Stack<~>();

    stack.push(seed);

    while(!stack.isEmpty()){
        Cell cell = (Cell) stack.pop();

        //set(cell, newChar);
        cellsFilled.add(cell);

        Cell nCell = cell.getNorth();
        Cell sCell = cell.getSouth();
        Cell eCell = cell.getEast();
        Cell wCell = cell.getWest();

        if(get(nCell) == oldChar && !cellsFilled.contains(nCell)) stack.push(nCell);
        if(get(sCell) == oldChar && !cellsFilled.contains(sCell)) stack.push(sCell);
        if(get(eCell) == oldChar && !cellsFilled.contains(eCell)) stack.push(eCell);
        if(get(wCell) == oldChar && !cellsFilled.contains(wCell)) stack.push(wCell);
    }

    return cellsFilled;
}
```