DITAA Software Design Project

Final Report

Daniel Longendelpher | Christopher Menart | Brad Schneider

Table of Contents

Table of Contents 1

Introduction 4

DITAA Functional Requirements 5
- Requirements Listing 5
- Use Case Description 7
- Non-Functional Requirements 7
- Added Functional Requirements 8
- Maintainability Improvements 8

DITAA Specifications 9
- Input Specification 9
  - Overview 9
  - Lines and Closed Shapes 9
  - Dashed Lines 11
  - Styling Elements/Modifiers to Closed Shapes 12
  - Colors 12
- Alternate Shapes 13
  - Bullet Points 13
  - Arrowheads 13
  - Point Markers 14
- Output Specification 14
- Specifying Non-Functional Requirements 15
  - Source Size Reduction 15
  - Bug Fixing 15
  - Acceptance/Conformance Tests 15

DITAA Design 17
- The Design of DITAA 17
  - Entry Point 17

Text Representation 17
Diagram Representation 18
Step 1 - Abstraction 18
Supporting Structures 18
Step 2 - Shape Isolation 19
Step 3 - Shape Labeling 20
Step 4 - Obsolete Shapes 22
Step 5 - Output Diagram Construction 22
Step 6 - Handling Special Tags 24
Step 7 - Render to Image 25
Handling HTML 26
Contracts of Core Classes 26
Diagram 26
Additional Diagram Methods 28
BitmapRenderer 29
Additional BitmapRenderer Methods 30
Design of New Features 31
GUI Design 31
Code Review 32
Documentation 33
SOLID Principles 33
Code Reduction 34
Relation to Previous Project Documents 34

DITAA Implementation 36
Existing Implementation 36
Classes 36
Maintainability Improvements 40
Java Version 40
Implementation of New Plugins 41
New Color Codes 41
GUI 41
Overview 41
Swing Components 41
Integration with DITAA 42
Usage 43
Building and Running 45
Smoke Testing 46
Setup 46
Test Case 1: Load Valid File 46

DITAA Testing 48

- Overview 48
  - Our Contributions 48
- Acceptance Testing 48
  - Dataset 48
  - Implementation 49
- Testing for New Plugins 49
  - Additional Color Codes 49
  - GUI Implementation 49
- Test Cases that Fail 50
- Stress Testing 51
- Unit Testing 51
- Coverage Breakdown 51
  - Additions 52
  - Results 52
- Coverage Breakdown 52
- Bug Discovery and Repair 53
  - Mixed Shape Bug(s) 53
  - The Wasteful Flood Fill Bug 57

Discussion and Conclusion 59

Appendix A: File Listing 61

Appendix B: Team Member Journals 69

- Daniel Longendelpher 69
- Christopher Menart 70
- Brad Schneider 73

Introduction

DITAA, or Diagrams Through ASCII Art, is a small (~10,000 lines) open-source Java program. It exists partly as a personal programming exercise for its original author, but its intended purpose is to convert ASCII art into full-fledged RGB bitmap images. In particular, it converts diagrams (such as UML and UML-like diagrams, which are often seen in ASCII format as code comments). It is a command-line program which can be called on individual text files, representing a single diagram, or HTML webpages containing multiple diagrams.

While relatively clean as far as single-person software projects go, DITAA leaves plenty of room for improvement in its maintainability. It contains numerous known, unfixed bugs, as well as dead code for unfinished features. Documentation within the code itself is limited almost entirely to informal comments, with JML or even Javadoc comments virtually absent. Documentation outside the code is essentially limited to instructions on how to invoke the program's various command-line options; even the description of the ASCII syntax DITAA is capable of accepting is incomplete. Methods and classes have no recorded contracts, leaving the finer points of their intended behavior as guesswork for future maintainers.

The following report documents the efforts of the authors to maintain DITAA as part of the Advanced Software Design course. We provide better documentation of DITAA's features (through requirements/specifications), design, and implementation. We fix several bugs in the program, add a couple of new features, and refactor some sections of the code for improved maintainability. These maintenance improvements are enhanced by the addition of documented contracts to the refactored section of the code, making the usage and purpose of the documented methods clear to any future maintainers.

DITAA Functional Requirements

Requirements Listing

The following table outlines functional requirements for the Diagrams Through Ascii Art (DITAA) software. This represents the existing features of the DITAA software. A separate listing of requirements will be provided for the proposed extended functionality.

Identifier	Requirement
REQ1	The system shall convert an input plain text file into a bitmap graphics output file.
REQ1.1	The system shall accept plaintext input using ASCII or UNICODE encodings.
REQ2	The system shall provide a command-line interface for operating the tool.
REQ3	The system shall display usage help with details for running the program.
REQ4	The system shall render open and closed polygons with solid lines in the output based on edges defined by characters in the input text file.
REQ4.1	The system shall render closed polygons defined by connected ‘-’, ‘|’, and ‘+’ characters (i.e. ‘-’ for horizontal edges, ‘|’ for vertical edges, and ‘+’ for corners).
REQ4.2	The system shall render open polygons defined by connected ‘-’, ‘|’, ‘+’, ‘<’, ‘>’, ‘^’, and ‘v’ characters (i.e. ‘-’ for horizontal edges, ‘|’ for vertical edges, ‘+’ for corners, and ‘<’, ‘>’, ‘^’, ‘v’ for left-, right-, upward-, and downward-facing arrow heads).
REQ4.3	The system shall render rounded corners when the input contains ‘/’ and ‘\’ characters in place of the ‘+’ (square corners) character on a rectangular shape.
REQ4.4	The system shall render polygons using dashed lines where any edge in an input polygon contains a ‘:’ (for vertical edges) or ‘=’ (for horizontal edges) character.
REQ4.5	The system shall separate or join shared edges of two polygons according to indicated user preferences.
REQ5	The system shall render textual labels where non-polygon-defining characters are encountered in the input.
REQ5.1	The system shall render a bulleted list where a line of input text is of the form ‘ o XXXX’ where X represents any text (a single space is required before and after

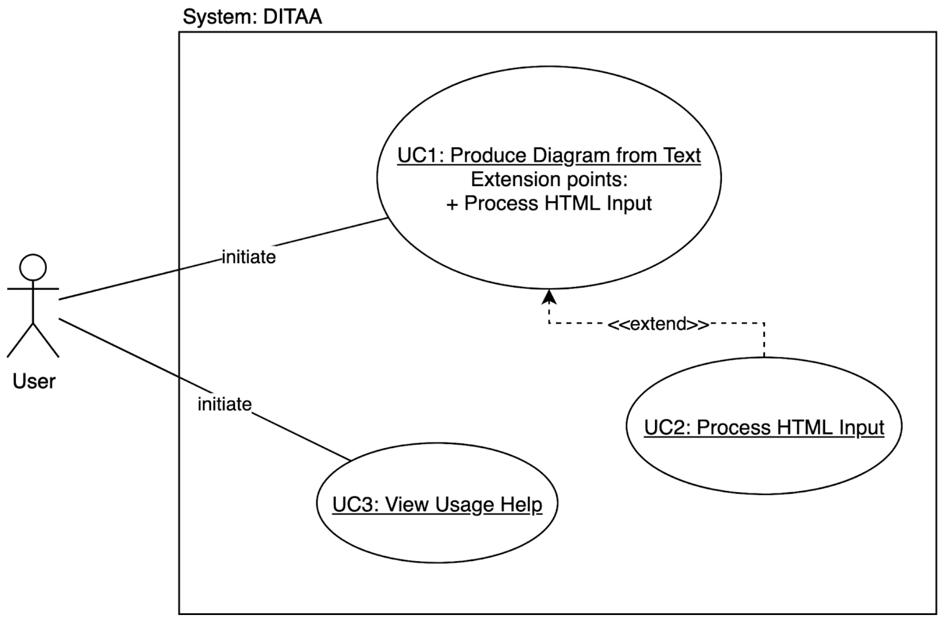
	the ‘o’ character).
REQ6	The system shall render a ‘point marker’ (i.e. a node) when the ‘*’ character is encountered on an edge.
REQ7	The system shall render non-rectangular shapes in the place of a rectangular closed polygon when tags of the form ‘{XX}’ are encountered inside of a closed polygon.
REQ7.1	The system shall render a UML Document symbol when the tag ‘{d}’ is encountered within a closed polygon.
REQ7.2	The system shall render a UML Database symbol when the tag ‘{s}’ is encountered within a closed polygon.
REQ7.3	The system shall render a UML Data (I/O) symbol when the tag ‘{io}’ is encountered within a closed polygon.
REQ8	The system shall render closed polygons with a colored fill when the input contains a color code of the form ‘cXXX’ where ‘X’ represents a valid hexadecimal digit. The digits shall be interpreted as the red, green, and blue components of the resulting color, respectively.
REQ8.1	The system shall support the following shorthand color codes: ● cRED (cD32) ● cBLU (c55B) ● cGRE (c9D9) ● cPNK (cEAA) ● cBLK (c000) ● cYEL (cEE3)
REQ9	The system shall accept HTML files as input. For each <pre> tag with the class attribute set to ‘textdiagram’ shall be interpreted as a separate diagram. The value of the ‘id’ attribute shall be used as the resulting diagram filename if provided. Otherwise, the filename shall be generated in the format ‘ditaa_diagram_X.png’ where ‘X’ represents a unique number.
REQ10	The system shall output a copy of the input HTML file with <pre> tags replaced with <img> tags. The <img> tags shall have an appropriate ‘src’ attribute set to the location of the bitmap diagram that replaces the <pre> tag from the original input file. If no output filename is provided the produced HTML file shall be named ‘xxxx_processed.html’ where ‘xxxx’ represents the original input filename.

Use Case Description

The DITAA software is a very simple command line tool with only a few modes of operation. While there are several features exposed through its ASCII syntax, there are only the following limited use cases identified:

UC1. Produce Diagram from Text
UC2. Process HTML Input
UC3. View Usage Help

The following diagram illustrates the relationship between the use cases and actors.



Non-Functional Requirements

These non-functional requirements are intended to be met by the end of the Testing Phase.

Identifier	Requirement
REQ11	The system shall contain 5% fewer SLOC than the pre-existing implementation
REQ12	The system shall pass tests for at least 1 software bug known to exist in the

	pre-existing implementation.

Added Functional Requirements

These functional requirements are proposed in addition to those in the pre-existing implementation.

Identifier	Requirement
REQ13	The system shall render diagonal polygon edges in place of horizontal/vertical edges when two or more diagonally-consecutive ‘/’ or ‘\’ characters are used. ‘/’ or ‘\’ characters must have row offsets of 1, and column offsets of +1 and -1 from the character below them, respectively, to indicate a diagonal line. Diagonal edges may not be rendered as dashed lines.
REQ13.1	The system shall render ‘r’ characters at the juncture of two diagonal edges as a rounded corner.
REQ14	The system shall render the consecutive character sequences ‘:)’, ‘:(‘, and ‘:D’ as smiley face icons, in line with any surrounding free text.

Maintainability Improvements

Improvements to the clarity and maintainability of the pre-existing implementation are also intended:

1. Adding Design-by-Contract specifications to each class, including class invariants, pre-conditions, post-conditions, and loop invariants.
2. Reviewing the class structure: Consolidating classes, and potentially creating new classes, leading to a more logical organization with only valid uses of inheritance.

DITAA Specifications

(Specifications for ‘additional functionality’ were not included in this document, pending determination of final ‘additional functionality’ requirements.)

Input Specification

The DITAA application accepts input files with a variety of ASCII constructs which are translated into diagram elements. This section specifies the various structures that may be provided to the DITAA application.

Overview

Allowable inputs can consist of free text and line groups. Line groups consist of any number of straight line components connected by corners. Any cycles in a line group which are not bisected by other lines from the same group form *closed shapes*, which are rendered with drop shadow and may be impacted by additional styling elements from both the input and command-line arguments.

Since the input language for DITAA is a two-dimensional language (a *picture language*), we express the grammar rules as either row vectors, column vectors, or two-dimensional matrices. The grammars represent most common structures for the elements of the input, but do not always express some of the special cases that may occur where elements overlap as this would lead to an unreasonably large number of element types that would likely be more difficult to understand.

Note that any text occurring in the document and not conforming to the following specification is treated as plain text and inserted without translation into the bitmap graphic output. The position of the plain text relative to the diagram elements is retained.

Lines and Closed Shapes

Horizontal Line Characters =	{‘-’, ‘=’}
Vertical Line Characters =	{‘|’, ‘:’}
Corner Characters =	{‘+’, ‘/’, ‘\’}

Note that ‘*’ character may also count as belonging to ‘corner characters’ under certain circumstances! See section ‘Point Markers’.

A **horizontal line** consists of a sequence of one or more consecutive horizontal line and/or corner characters, containing at least one horizontal line character.

The following grammar rules describe the syntax of horizontal lines (*HL*) and dashed horizontal lines (*DHL*).

Horizontal Lines		
HL	$\rightarrow$	$[\,-\,] \mid [\,+\,] \mid [\,HL\ HL\,] \mid DHL$
DHL	$\rightarrow$	$[=\ HL\,] \mid [\,HL\ =]$

A **vertical line** is a collection *E* of one or more vertical line and/or corner characters, containing at least one vertical line character, such that

∃ natural numbers a, b, c s.t.

∀ i in [a, b], input[c,i] ∈ *E*

Each horizontal and vertical line character is considered to be part of the *largest* such line that can be defined. Lines are to be rendered as contiguous straight lines in the output.

The following grammar rules describe the syntax of vertical lines (*VL*) and dashed vertical lines (*DVL*).

Vertical Lines		
VL	$\rightarrow$	$[\,\vert\,] \mid [\,+\,] \mid \begin{bmatrix} VL \\ VL \end{bmatrix} \mid DVL$
DVL	$\rightarrow$	$\begin{bmatrix} : \\ VL \end{bmatrix} \mid \begin{bmatrix} VL \\ : \end{bmatrix}$

A **corner** is a corner character within a horizontal or vertical line. A corner can be a component of *both* a horizontal line and a vertical line.

While the corner character ‘+’ appears as a terminal in the grammar rules for both horizontal and vertical lines, semantic rules overlaid on the grammar dictate that the ‘+’ character is only expected where two lines meet (corners of shapes, and perpendicular intersections of lines).

Lines consisting of solely a corner character are somewhat ambiguously defined by the grammar above (a single ‘+’ character is both a horizontal and vertical line). However, this definition is necessary to account for the special case of a 2x2 matrix of ‘+’ characters, which does produce a closed shape.

A **line group** is a collection *L* of one or more horizontal, vertical lines, and corners such that

```
∀ corners c0, c1 ∈ L, they can be arranged in a sequence c0, c1, c2 s.t.
        ∀ c in [0, #L-1], ci and c(i+1) are joined by a line ∈ L or adjacent
AND all lines ∈ L are adjacent to one of the corners in L.
```

If shared lines/adjacency can be modeled as edges which join two lines as nodes in a graph, a line group is a connected component.

Note that by the above definition, it is possible to have a line group with no lines, only corners.

A **closed shape** is a cycle of corners in a line group which does not contain any smaller cycles (i.e. it is a minimal cycle).

A closed shape will be rendered as such in the output diagram, with a drop shadow. The following grammar rules describe the format of a closed shape (*CS*) in the input. It is assumed in this case that the length of the horizontal lines are equal and the length of the vertical lines are equal. As described above, the lines meet at the corner characters, ‘+’ (square corners) or ‘\’ and ‘/’ (round corners) to form a line group.

Shapes
$CS \rightarrow \begin{bmatrix} + & HL & + \\ VL & & VL \\ + & HL & + \end{bmatrix} \mid \begin{bmatrix} / & HL & \backslash \\ VL & & VL \\ \backslash & HL & / \end{bmatrix}$

Dashed Lines

Dashed Line Characters = {‘:’, ‘=’}

Any line or line group containing one or more dashed line characters is rendered as with dashed instead of solid lines.

Styling Elements/Modifiers to Closed Shapes

Input text is **inside** a closed shape if

```
NOT ∃ a sequence of coordinates (I, J) s.t.
    I[0], J[0] is the coordinate of the input text AND
    ∀ i in [0, length(I)-1], |I[i] - I[i+1]| + |J[i] - J[i+1]| = 1 AND
    ∀ i in [0, length(I)-1] I[i], J[i] is NOT the coordinate of a line element from
        the shape AND
    (I[end] ∈ {0, max line length of input} OR J[end] ∈ {0, numlines in input})
```

In English, input text is **inside** a closed shape if there is no path of spaces from the input text to the outer edge of the input that does not pass the lines composing the shape--it is fully surrounded by those lines.

Note that for the purposes of two-dimensional grammar definition, the grammar rules do not indicate the size of the closed shape (that is, the size of the lines that create the enclosure). It is assumed that the shapes are large enough for all characters comprising the modifier to fit within the boundary of the shape.

Input text inside a close shape is a **styling element** if it matches one of the patterns defined as possible style elements that follow.

Colors

Hexadecimal Characters = [‘1’, ‘2’, ‘3’, ‘4’, ‘5’, ‘6’, ‘7’, ‘8’, ‘9’, ‘0’, ‘A’, ‘B’, ‘C’, ‘D’, ‘E’, ‘F’]
Predefined Colors = [‘RED’, ‘BLU’, ‘YEL’, ‘BLK’, ‘PNK’, ‘GRN’]

A color styling element consists of four characters, where the first character is ‘c’, and the next three characters either match one of the strings in the Predefined Colors, or consist only of characters from the Hexadecimal Characters set.

A closed shape with a color styling element is rendered with the indicated color. [List the hex codes of the predefined colors or something?] If there are multiple color elements, only one is applied.

The following table illustrates grammar rules involved with modifying colors.

Colors		
HD	$\rightarrow$	$0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9 \mid A \mid B \mid C \mid D \mid E \mid F$
PD	$\rightarrow$	$RED \mid BLU \mid YEL \mid BLK \mid PNK \mid GRN$
CM	$\rightarrow$	$c\,PD \mid c\,HD\,HD\,HD$
CS	$\rightarrow$	$\begin{bmatrix} + & HL & + \\ VL & CM & VL \\ + & HL & + \end{bmatrix} \mid \begin{bmatrix} / & HL & \backslash \\ VL & CM & VL \\ \backslash & HL & / \end{bmatrix}$

Alternate Shapes

An alternate shape styling element is one of three strings:

1. "{d}": A closed shape with this tag is rendered with a wavy bottom edge.
2. "{s}": A closed shape with this flag is rendered as a cylinder.
3. "{io}": A closed shape with this flag is rendered as a skewed rhombus.

If there are multiple alternate shape styling elements in a single closed shape, only one is applied. They have no effect if the closed shape is not a rectangle (has 4 composing lines).

The following table illustrates a grammar relating to alternate shapes:

Alternate Shapes		
SS	$\rightarrow$	$\{d\} \mid \{s\} \mid \{io\}$
AS	$\rightarrow$	$\begin{bmatrix} + & HL & + \\ VL & SS & VL \\ + & HL & + \end{bmatrix} \mid \begin{bmatrix} / & HL & \backslash \\ VL & SS & VL \\ \backslash & HL & / \end{bmatrix}$

Bullet Points

If the pattern " o " is present, and followed by one or more characters of free text (text not part of another special command as specified in this document) the 'o' is rendered as a round bullet point.

Arrowheads

A character *c* at position *x, y* is an **arrowhead** if any of the following are true:

c == '>' && input[x-1, y] is in a line.
    If so, *c* is rendered as a right-facing arrow at the end of that line.

c == ‘<’ && input[x+1, y] is in a line.
    If so, *c* is rendered as a left-facing arrow at the end of that line.
C == ‘^’ && input[x, y-1] is in a line.
    If so, *c* is rendered as an up-facing arrow at the end of that line.
c ∈ {‘v’, ‘V’} && input[x, y+1] is in a line.
    If so, *c* is rendered as a down-facing arrow at the end of that line.

Note that *c* is not considered to be part of the line. If there are more line characters on other sides of *c*, *c* does not join them as a corner would. A set of grammar production rules describing arrows is located below.

Arrows		
LA	$\rightarrow$	$\begin{bmatrix} < & HL \end{bmatrix}$
RA	$\rightarrow$	$\begin{bmatrix} HL & > \end{bmatrix}$
UA	$\rightarrow$	$\begin{bmatrix} \wedge \\ VL \end{bmatrix}$
DA	$\rightarrow$	$\begin{bmatrix} VL \\ \vee \end{bmatrix}$

Point Markers

Under certain circumstances, the ‘*’ character counts as a corner.

It must appear directly adjacent to at least one horizontal or vertical line character to count, and a second ‘*’ character must not appear directly adjacent to it opposite from that line character.

There is one difference between ‘*’ corners and ‘+’ corners. Two ‘*’ corners that are directly adjacent, with no intervening line characters, are not adjacent for the purposes of forming line groups.

This functionality is still experimental in the pre-existing implementation.

Output Specification

DITAA can be run in two modes, standard or html. In the standard mode of operation, the software outputs a single PNG file capturing the graphical depiction of the input based on the specification rules described above. In the HTML mode of operation, DITAA outputs PNG files and also outputs modified copies of the input HTML file.

In HTML mode, the same input specification rules that are listed previously in this document apply. The input ASCII markup must be contained in a pre-formatted text HTML tag:

<pre class=”textdiagram”> (ASCII contents) </pre>

The tag may optionally contain an ‘id’ attribute, which determines the name of the output PNG file that was built from the ASCII contents. If no ‘id’ attribute is provided, then the name is generated as ‘ditaa_diagram_x.png” where ‘x’ is a number.

The output HTML is a modified version of the input that includes references to the images produced by DITAA via <img> tags in place of the original <pre> tags..

Specifying Non-Functional Requirements

Source Size Reduction

It is a requirement to keep the size of the source code at a maximum of 95% the size of the pre-existing implementation found at <http://ditaa.sourceforge.net/>.

The size metric used will be source-lines-of-code, which will be determined using the Statistic plugin to Intellij IDEA. This tool reports 9580 SLOC for the pre-existing implementation, implying a target of 9101 SLOC.

Bug Fixing

The program should exhibit correct behavior in at least one case which causes a bug in the pre-existing implementation. This bug should be documented with a test case that demonstrates the incorrect/correct behavior, using a file identified as ‘fixedbug.txt’ in the project git repository at <https://github.com/CJMenart/CS-7140-2020>.

Acceptance/Conformance Tests

The program must pass all unit tests provided in the pre-existing implementation by exhibiting the same behavior as the pre-existing implementation, except where it is explicitly noted that behavior should or may diverge. (In particular behavior related to point markers as corners appears to be partly incidental, and is to be considered flexible.)

The program must pass all unit tests written in the course of constructing these specifications, again by matching the behavior of the pre-existing implementation except where otherwise noted. These additional test cases are to be packaged with the final code.

Additional test cases must be constructed for any/all additional functionality beyond that in the pre-existing implementation, as determined by the final requirements. These test cases are also to be packaged with the final code.

While unit testing is used to prove the correctness of specific functions within the implementation, additional testing is required to confirm that the final deliverable meets the specifications in this document. Since there are few high-level functions in the DITAA software, acceptance and conformance testing are largely considered to be one event. Sample inputs of two types will be provided. The first will cover instances of unique element types per input, such as lines, arrows, and closed shapes of varying sizes. The second test input type will cover combinations of these elements used together, testing common structures such as intersecting lines and shapes, combinations of dashed and solid lines, rounded corners, shared edges, etc. Test inputs will be designed such that all possible options (shapes, colors, etc.) are tested in multiple inputs.

DITAA Design

The Design of DITAA

This section of the document describes the logical design of DITAA, including the objects which represent the large pieces of functionality and contracts within the application. While the object names were kept similar to the existing DITAA implementation for clarity, the mentioned objects are not intended to refer specifically to the implementation.

Entry Point

The primary entry point of DITAA is the CommandLineConverter object. It is responsible for interpreting user-provided options and then invoking the other core logic, such as Diagram, which is responsible for parsing ASCII diagrams, and BitmapRender, which is responsible for drawing the parsed diagrams as PNGs. We will depict the core logic of DITAA, referencing only its most important classes and methods, using pseudo-code and high-level comments in a literate programming-like style:

```
CommandLineConverter.main(commandLineArgs: List of strings):
    Parse command-line arguments, configure any flags
    If the input is an HTML file, according to commandLineArgs:
         new HTMLConverter.convertHTMLFile(designated input file)
    Otherwise, input is ASCII art according to commandLineArgs:
        String asciiInput := read the designated input file
        grid := new TextGrid(asciiInput)
        diagram := new Diagram(grid) # parses input
        new BitmapRenderer.renderToImage(diagram)
```

Text Representation

A TextGrid is a representation of ASCII art as a 2D grid of text cells instead of a 1D string. This logical structure contains methods for querying and modifying the grid as such.

Diagram Representation

Most of the logical design described herein will focus on Diagram, which represents the ASCII art as a series of shapes, and whose constructor bears primary responsibility for interpreting the ASCII art in a TextGrid.

Step 1 - Abstraction

The first step in constructing a Diagram is about extracting connected components of lines and corners:

```
Diagram(grid: TextGrid):
    workGrid := copy of grid
    Remove text, and remove point markers that are 'in front' of
        otherwise continuous lines, replacing them with line
        characters, on workGrid.
    abstractionGrid := new AbstractionGrid(workGrid)
    # get distinct shapes, i.e. connected components
    List of CellSet boundarySetsStep1 :=
        abstractionGrid.getDistinctShapes()
    ...
```

Supporting Structures

A couple of the structures mentioned in the above design bear further explanation.

A CellSet is simply a Set of cells (i.e. locations) contained in a TextGrid.

An AbstractionGrid is similar to a TextGrid in that it is a representation of the input. However, a TextGrid directly represents the original ASCII character input, whereas in an AbstractionGrid every original ASCII character is ‘upsampled’ into a 3x3 grid. For example, a horizontal line would change from ‘-’, to this:

A horizontal line, for example, is
000
111
000

An intersection would go from ‘+’ to:

010
111

This ‘upsampling’ has several perks. Here in Step 1, it allows our definition of ‘connected components’ to reflect the ways different line characters can connect. Because we use an AbstractionGrid, a simple flood fill or breadth-first-search on that abstraction grid can easily tell that the following diagram contains two separate connected components:

```
---------
/------\
|      |
\------/
```

Step 2 - Shape Isolation

Isolating the connected components is not enough to fully understand a diagram. The ultimate goal of the DITAA algorithm is to isolate a set of paths which are closed, non-self-intersecting curves (defining closed shapes) or open curves that can later be rendered as lines.

```
<Diagram constructor, part 2>:
    ...
    boundarySetsStep2 := another list of CellSets
    For each CellSet in boundarySetsStep2:
        copy the set into a fresh TextGrid copyGrid
        while any arbitrary empty cell emptyCell can be found in
        copyGrid:
           Flood-fill outwards from the empty cell, filling all empty
           characters encountered with a filler character. Take the
           set of line characters surrounding the final flood-fill
           and add it to boundarySetsStep2.
    for each CellSet in boundarySetsStep2:
        if this set is identical to another set in boundarySetsStep2:
            remove it.
    ...
```

This step produces connected components that consist of several closed areas, which are broken apart in this step. For example, this input:

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

could be a single connected component from Step 1. But after Step 2, the four ‘interior’ rectangles and one large ‘exterior’ rectangle have been isolated as shapes.

This information is still being manipulated in the ‘upsampled’ AbstractionGrid space, and as such can identify tiny closed areas such as the following:

```
++
++
```

The ‘duplicate removal’ function is necessary because we can identify ‘boundaries’ from two directions in this stage. For example, a simple square:

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

Would be identified by both the flood fill from the *inside*, and one from the empty space *outside*, so caution must be taken to not count this object twice.

Step 3 - Shape Labeling

In Step 3, the boundaries identified in Step 2 will be labeled as Closed, Open, or Mixed. Any boundaries which are labeled as Mixed will be broken into a series of Closed and Open ones. A definition of Mixed boundaries follows the pseudo-code.

```
<Diagram constructor, part 3>:
    ...
    For each CellSet boundary in boundarySetsStep2:
        startCell := an arbitrary cell from boundary
        follow adjacent boundary cells until reaching a dead-end,
        intersection, or startCell
        if startCell was reached:
            boundary.type := Closed
        if a dead-end was reached:
            boundary.type := Open
        if an intersection (three or more adjacent cells!) was
        reached:
            boundary.type := Undetermined
    for each CellSet boundary in boundarySetsStep2 s.t. boundary.type
    = Undetermined:
        copy the set into a fresh TextGrid copyGrid
        Cell blank = arbitrary blank in boundary
        flood-fill blanks outward from blank with filler characters
        if no blank characters remain in boundary:
            boundary.type := Open
        else:
            boundary.type := Mixed
        for each CellSet boundary in boundarySetsStep2 s.t.
         boundary.type = Mixed:
           remove boundary from boundarySetsStep2
           List of CellSet simplerBoundaries := decompose boundary
            into Closed and Open boundaries
           boundarySetsStep2.addAll(simplerBoundaries)
    ...
```

The logic above identifies the type of all shapes. A closed shape can be traversed starting from a random point on the line and back to the starting point without ever hitting an intersection where there is a choice about where to trace next. If a dead-end is reached during traversal, the shape is Open. Shapes with intersections (which are never closed shapes) can be identified by determining if a single flood-fill captures all the empty area in and around a shape; the shape is Open if it does not surround any space, and Mixed otherwise.

A Mixed shape is one that is neither an open shape, nor a single non-self-intersecting curve defining a closed shape. An example is given in the comments and repeated here:

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

Step 4 - Obsolete Shapes

The final step of boundary processing is to eliminate ‘obsolete’ shapes, which are those whose area is equal to the sum of other shapes. An example is the figure below, in which our Diagram only needs to define four rectangles, and can discard the fifth, larger one they collectively comprise as ‘obsolete’:

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

The pseudo-code for this step follows:

```
<Diagram constructor, part 4>:
    ...
    for each CellSet boundary in boundarySetsStep2:
        sumSet := new empty CellSet
        for each other CellSet other in boundarySetsStep2:
            if other is a subset of boundary:
                add other to sumSet
        if sumSet = boundary
            remove boundary from boundarySetsStep2
    ...
```

This concludes the core logic of identifying what shapes are represented in an ASCII diagram.

Step 5 - Output Diagram Construction

The Diagram constructor uses the list of CellSets yielded by the proceeding steps to construct DiagramShape objects, which are the final internal representation of parts of a picture.

```
<Diagram constructor, part 5>:
    ...
    shapes := new empty list of DiagramShapes
    For each closed shape in boundarySetsStep2:
        shape := extract ordered list of corners and construct new
         DiagramShape
        Add shape to shapes
    If command-line args indicate to separate shared edges:
        For each shape in shapes:
            For each other shape in shapes:
                If both shapes share an edge, pull them apart a bit
    ...
```

Shapes logically represent locations in pixel space. If the user has requested that shapes do not share the exact same line for an edge, the edges for offending shapes are pulled apart from each other.

```
<Diagram constructor, part 6>:
    compositeShapes := new empty list of lists of DiagramShapes
    For each open shape in boundarySetsStep2:
        shape := extract list of ordered lists of corners
         representing the shape
        Add shape to compositeShapes
    For each shape in compositeShapes:
        connected endpoints to corners or far ends of cells
```

We are eliding over the CompositeDiagramComponent object in the above, which essentially wraps a list of DiagramComponents. DiagramShapes, in turn, are defined by an ordered list of corner points, which makes the logic to draw them relatively simple.

Open shapes must be represented by CompositeDiagramShapes because, while every closed shape by this point must be a simple closed curve, open shapes could entail things like the following:

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

Multiple branching paths can't be represented by a single ordered list, which assumes that every point is connected by lines only to its neighbors. However, the paths can be broken into a collection of shapes that collectively represent the Open shape.

Generally, lines end at the pixel location corresponding to the 'middle' of the cell in which they terminate. However, if those lines connect to a dot marker, corner, or arrowhead, they need to be adjusted to connect to the edge of the cell.

Step 6 - Handling Special Tags

The final step in initializing the Diagram involves handling other special tags from the input, as outlined below.

```
<Diagram constructor, part 7>:
    ...
    For each shape in shapes:
        If there is color code inside area of shape:
            Shape.color = indicated color
    For each shape in shapes:
        If there is a markup tag inside area of shape:
            shape.type = custom shape tape indicated by tag
    For each arrowhead character in grid:
        Add new Arrowhead to shapes
    For each point marker in grid:
        Add new little circle to shapes

    Subtract all characters that have been processed from grid
    # what remains is free text
    Fill all blanks directly between two non-blanks with filler
    CellSet list textGroups := isolate connected components of text
    For each textGroup in textGroups:
        Decide on color and text alignment for textGroup
        add textGroup to textObjects
```

Step 7 - Render to Image

Once the Diagram is constructed, rendering becomes easier. The primary class of interest is BitmapRenderer, whose primary method renderToImage takes a Diagram and returns a bitmap.

```
<BitmapRenderer.renderToImage, part 1>:
    Diagram diagram = a previously-constructed Diagram
    2D vector offset = a standard offset for drop shadows, to
     simulate light source
    bitmap := new bitmap
    Color bitmap white
    # render shadows
    For each closed shape in diagram:
        path := extract boundary of shape as a path
        Translate the path by offset
        Fill the path with dark gray in bitmap
        Apply Gaussian blur to bitmap
    ...
```

Applying a blur to the image after rendering only the shadows allows our drop shadows to have fuzzy edges, without affecting any other element of the image.

The rest of the rendering process does not require a particular order, except for storage shapes, a particular custom shape which is a special case as they are pseudo-3D and must be rendered ‘bottom’ to ‘top’.

```
<BitmapRenderer.renderToimage, part 2>:
    ...
    For each shape in diagram s.t. shape.type = Storage Shape:
        Draw shape
    For each shape in diagram s.t. shape.type != Storage Shape:
        If shape.type is a type defined by SVG:  # custom shapes
            Load and draw indicated SVG from assets onto bitmap
        If shape.type is a type defined by paths:
            Extract and trace the path(s) onto bitmap
    For each point marker in diagram:
        Draw a small circle at their location onto bitmap
    For each text group in diagram:
        Render the text at the indicate position and alignment
    return bitmap
```

Handling HTML

With the core methods elucidated, a final option step is required to support the HTMLConverter, which is the last bit of high-level logic mentioned in the main block of pseudocode. A web page can refer to many diagrams; HTMLConverter uses the same logic we have shown here for processing ASCII diagrams, but can loop over all the images referred to in an HTML document.

```
<HTMLConverter.convertHTMLDocument>:
     htmlsource := load the input
     List of text segment outputSegments := an empty list
     For each <pre> tag in htmlsource:
          diagram := construct new Diagram from text in source
          Bitmap := BitmapRenderer.renderToImage(diagram)
          Save bitmap to disk
          Replace the <pre> tag with an <img> tag which refers
           to the bitmap just written out
     Write out the new HTML document alongside the new images
```

The OutputDocument class constructs the ‘modified’ HTML document by looping over diagrams found within the original document at output time, rather than modifying one large string in place.

Contracts of Core Classes

The original implementation of DITAA does not make significant use of Design-by-Contract or Correct-by-Design elements. However, this section develops contracts for some of the core objects discussed above that align with the intention and behavior of the design.

Diagram

Class Invariant

- Every entity in shapes, compositeShapes and textObjects has x-bounds between 0 and this.width
- Every entity in shapes, compositeShapes and textObjects has x-bounds between 0 and this.height

Public Methods

- Diagram(TextGrid grid, ConversionOptions options) (constructor)
    - Precondition: No arguments are null.
    - Postcondition: shapes, compositeShapes and textObjects represent every entity in the ASCII art represented by grid. (The way in which input text is represented is involved and detail-dependent, making this the strongest post-condition we can develop here. We cannot even guarantee that every character is represented in some fashion, as there are edge cases where this is not true.)
- getAllDiagramShapes()
    - Precondition: True
    - Postcondition: Returns an array with all the elements of shapes and compositeShapes
- getMinimumOfCellDimension()
    - Precondition: True
    - Postcondition: Returns the minimum of cellWidth and cellHeight.
- getShapesIterator()
    - Precondition: True
    - Postcondition: Returns an iterator over shapes.
- getShapes, getHeight, getWidth, getCellWidth, getCellHeight, getCompositeShapes, getTextObjects are all getters that return their respective data members.
- getCellMinX, getCellMixX, getCellMaxX, and the corresponding 'Y' methods are pure getter-like methods which return the corresponding coordinates of a cell in pixel space rather than cell space.

Private Methods

- removeObsoleteShapes(grid, sets)
    - Precondition: Arguments are not null. The cellSets in sets are within the bounds of grid.
    - Postcondition: A minimal number of CellSets are removed from sets s.t. no CellSet in sets covers an area which is exactly the same as the union of the area covered by any number of other CellSets in sets. Returns True iff any CellSets were removed to meet this condition.
- separateCommonEdges(shapes)
    - Precondition: Argument is not null
    - Postcondition: For each pair of edges in shapes which partially or fully overlap, the endpoints of both edges are shifted by getMinimumOfCellDimension() / 5, such that the orientation of each edge

    is unchanged and the area of each parent shape decreases rather than increases.

- removeDuplicateShapes()
    - Precondition: this.shapes is not null.
    - Postcondition: Remove a minimal set of shapes from this.shape s.t. there exists no two integers i and j where i != j but this.shapes[i].equals(this.shapes[j]).
- addToTextObjects, addToShapes, addToCompositeShapes are void methods that add DiagramShapes to the corresponding data members of this.

Additional Diagram Methods

As a part of refactoring the code for improved maintainability, we refactor Diagram by decomposing its larger constructor, which results in many additional private methods. Contracts for these are given here:

- findInteriorExteriorBoundaries(connectedComponents, grid)
    - Precondition:grid does not contain text or point markers 'over' what are intended to be contiguous lines. Topologically, each CellSet in connectedComponents contains no 1-dimensional holes (i.e. each are contiguous)
    - Postcondition: The union of all CellSets in the result matches the union of all CellSets in connectedComponents. Topologically, each CellSet in the result contains either 0 or 1 2-dimensional holes.
- breakBoundariesIntoOpenAndClosed(unclassifiedBoundaries, grid)
    - Precondition: grid does not contain text or point markers 'over' what are intended to be contiguous lines.
    - Postcondition: Let open = result[0] and closed = result[1]:
        - Each CellSet in open is an open line
        - Each CellSet in closed is a simple closed curve
        - The union of all CellSets in open and closed covers the union of cells in the old value of unclassifiedBoundaries
- assignColorCodes(grid)
    - Precondition: this.shapes is not null
    - Postcondition: For each DiagramComponent shape in this.shapes, IF any color codes are inside the area of that shape in grid, shape.fillColor is set to one such color.
- assignMarkup(grid, processingOptions)
    - Precondition: All arguments are non-null and this.shapes is non-null

    - Postcondition: For each DiagramComponent shape in this.shapes, IF any markup tags are inside the area of that shape in grid, the shape's type and definition are set according to one such markup tag.
- <u>extractText(grid)</u>
    - Precondition: this.textObjects is non-null, this.shapes is non-null, and grid is non-null
    - Postcondition: All 'free text' in grid (text not part of another recognized DITAA entity) is represented by a DiagramText object in this.textObjects. Any DiagramText spatially overlapping any shapes in this.shapes is set to a color contrasting with one such shape. Any other DiagramTexts are colored black. Any DiagramText overlapping any shapes of type TYPE_CUSTOM in this.shapes is given an outline.
- <u>makeArrowheads(grid)</u>
    - Precondition: grid is non-null and this.shapes is non-null
    - Postcondition: All arrowhead characters in grid are represented by arrowhead shapes in this.shapes
- <u>makePointMarkers(grid)</u>
    - Precondition: grid is non-null and this.shapes is non-null
    - Postcondition: All point markers ('*' characters) in grid are represented by round shapes in this.shapes

BitmapRenderer

Class Invariant

- normalStroke and dashStroke are either null, or hold Stroke objects which draw solid and dashed lines respectively.
- (BitmapRenderer has almost no internal state--there are a pair of private data members holding drawing tools, but these are mostly just used to cache the same values constructed in the course of render().)

Public Methods

- <u>renderToImage(diagram, options)</u>
    - Precondition: No arguments are null.
    - Postcondition: Makes a BufferedImage image with the same size as diagram and returns render(diagram, image, options)
- <u>render(diagram, image, options)</u>
    - Precondition: No arguments are null.
    - Postcondition: draws a visual representation of diagram onto image. Overwrites any existing contents in image.

- <u>renderToPNG(diagram, filename, options)</u>
    - If this returns true, it writes renderToImage(diagram, options) to filename as a PNG.
- <u>isColorDark(color)</u>
    - Precondition: Argument is not null
    - Postcondition: Returns true if no color channel in color is greater than 200.

Private Methods

- <u>renderTextLayer(textObjects, width, height)</u>
    - Precondition: Arguments are not null
    - Postcondition: Returns a RenderedImage of size width x height with transparency which renders all the text in textObjects (that fall within its bounds).
- <u>renderCustomShape(shape, g2)</u>
    - Precondition: Arguments are not null. this.normalStroke and this.dashStroke are not null. shape.type = TYPE_CUSTOM and the PNG or SVG asset defining the custom shape is available at the location specified by shape.getDefinition().getFilename().
    - Postcondition: The custom shape is rendered onto g2.
- <u>renderCustomSVGShape(shape, g2)</u>
    - Precondition: Arguments are not null. shape.getDefiniton().getFilename() exists and is an SVG file.
    - Postcondition: Renders the shape defined by file at shape.getDefinition().getFilename() fit to shape.getBounds() on g2.
- <u>renderCustomPNGShape(shape, g2)</u>
    - Precondition: Arguments are not null. shape.getDefiniton().getFilename() exists and is an PNG file.
    - Postcondition: Renders the shape defined by file at shape.getDefinition().getFilename() fit to shape.getBounds() on g2.

Additional BitmapRenderer Methods

As a part of refactoring the code for improved maintainability, we refactor BitmapRenderer by decomposing the large method render(), which results in some additional private methods. Contracts for these are given here:

- <u>dropShadows(diagram, image, performAntialias)</u>
    - Precondition: No arguments are null. Image contains a blank white background and nothing else.
    - Postcondition: A shadow of every shape in diagram.getAllDiagramShapes() for which shape.dropsShadow() is

rendered onto a copy of image, which is returned as the result. The shadows are blurry dark-grey copies of their respective shapes, and offset from the shape's true positions by diagram.getMinimumOfCellDimension() / 3.333f. The image is anti-aliased iff performAntialias = true.

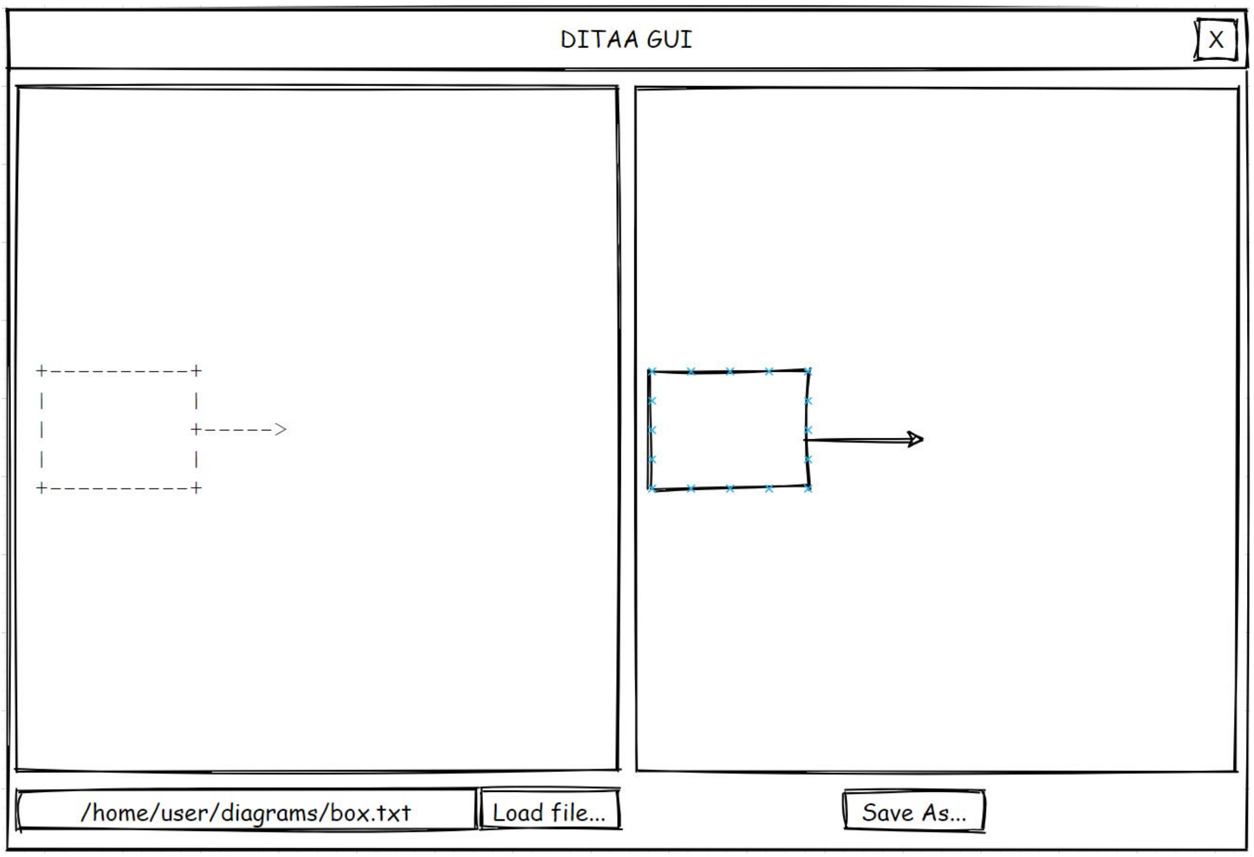
- renderText(diagram, g2)
    - Precondition: No arguments are null.
    - Postcondition: Every TextObject in diagram.getTextObjects() is drawn onto g2.

Design of New Features

Two new additions are being added to DITAA - new color codes and a GUI. Since the addition of color codes falls under the existing logic described in Step 6 - Handling Special Tags, this section focuses on the design of the GUI.

GUI Design

The GUI will consist of a small number of simple components that provide the ability to graphically select an input file, run DITAA, preview the output, and save the diagram to a bitmap graphic. A mockup of the desired layout is presented below.



```
+----------+
|          |
|          +----->
|          |
+----------+
```

The addition of the GUI comes with a small number of logical changes and additions to the DITAA software. The pseudocode below represents a small revision to the CommandLineConverter presented earlier:

```
CommandLineConverter.main(commandLineArgs: List of strings):
    Parse command-line arguments, configure any flags
    If the input is an HTML file, according to commandLineArgs:
        new HTMLConverter.convertHTMLFile(designated input file)
    Otherwise, input is ASCII art according to commandLineArgs:
        if is-empty(commandLineArgs) or “--gui” in commandLineArgs:
            launchGUI()
        else:
            # proceed with headless conversion
            String asciiInput := read the designated input file
            grid := new TextGrid(asciiInput)
            diagram := new Diagram(grid) # parses input
            new BitmapRenderer.renderToImage(diagram)
```

The GUI itself will be implemented in a standard event-based GUI framework and is expected to fulfill the following logical flow which focuses on reacting to an input file being chosen:

```
GUI.event_loop():
    while not exit_requested:
        event := get_user_event()
        if event.type == input_file_loaded:
            contents := get_file_contents()
            run_ditaa_on_input(contents)

            display_input_contents()
            display_output_diagram()
```

Code Review

This section contains a critique of the implementation of DITAA. Since the design contained in this document is derived from this implementation (no original design exists), comments pertain to both the design and implementation.

Documentation

We find DITAA’s code to be readable and somewhat well documented by comments (although most methods are missing javadoc comments which would further assist comprehension). Contract documentation, which is minimal to nonexistent, would also have increased the maintainability of the software.

SOLID Principles

Large portions of the code are compressed into a small number of large classes, indicating that these classes may be taking on overly broad responsibilities:

- Much of the code in Diagram could be refactored to other classes, or at least pulled out into helper methods so that the body of Diagram() is not left at over 500 lines--too long by any measure.
- It is understandable for the BitmapRenderer.renderToImage(), the method responsible for creating a bitmap image from a Diagram, to be one of the large methods in DITAA. But it still weighs in at hundreds of lines, indicating that it may be decomposed in some way.
- TextGrid is also one of the largest classes in DITAA’s implementation, with well over 100 methods! It appears that TextGrid is taking on multiple responsibilities which could potentially be decomposed: that of a data object represented a 2D grid of text, with all the getters and setters this entails, but *also* numerous sophisticated operations on that grid, many of which used by the Diagram constructor and involve processing that encodes knowledge about how ASCII characters come together to represent shapes!
    - While this class’ size would suggest it is best decomposed into multiple smaller classes, an attempt to do so revealed that it is difficult to break TextGrid’s logic into two or more classes that do not all have data envy for a core TextGrid class. There is at least some justification for the author’s design in this case.

In the implementation of the Diagram constructor, it is possible to get the sense that much of DITAA’s core logic evolved by the accretion of edge-case handling over time. An example would be how Mixed shapes are first analyzed using the ‘trace method’, and then analyzed using the ‘fill method’ of this does not work. Mixed shapes are decomposed using one approach--the subtraction of closed shapes--but then decomposed using a completely different method if this first approach is unsuccessful.

It is likely that this approach has proliferated many ways of accomplishing the same tasks, and that this proliferation has introduced a number of bugs, some of them noted through comments, but some unnoted. For instance, we believe that there are edge cases where the logic for decomposing mixed shapes would fail--logic that may not fail if the ‘advanced’ method were simply complete and applied to all shapes.

Even where the logic is correct, of course, the code could be made considerably simpler and more concise if one single method was usable in all cases for a particular task in diagram processing. In some cases, it may be that this is not possible, of course, but even then, further documentation should be included as to the rationale behind these decisions. Many of the rationales for the existing implementation, documented above in this document, had to be inferred at some effort.

Code Reduction

The best path to reducing the size of the codebase by 5% seems to be by fixing pieces of the implementation to share common code and ensure that class implementations follow the SOLID principles as noted in previous sections. Reducing the size of the codebase by re-designing the high-level algorithm of DITAA (i.e. changing the design laid out in this document) may be possible, but would require rewriting large pieces of implementation in a project with very little existing ability to test and verify the result.

Relation to Previous Project Documents

Primarily, a Design is a strategy for meeting a specification. Many parallels can be seen between the project specifications and the core logic contained primarily in the Diagram constructor. Steps are included for detecting each of the diagram elements indicated in the specification: shapes, arrowheads, point markers, color tags, markup tags, etc.

Of course, the most difficult portion of both the design and specifications concern open and closed shapes made up of lines. In this, the design meets the specification in a non-straightforward way.

The specification contains the concept of a connected group of lines and corners, which is matched by the first step of diagram processing in the implementation. From there, DITAA’s core logic further decomposes these groups into entities that can each be defined by an ordered sequence of points. Both the specification and this design contain the concept of a ‘closed shape’, and the way closed shapes are processed in the design mirrors the definition in the specification. Simple cycles are extracted in Step 2 of the Diagram constructor, and these are later decomposed into ‘minimal’ cycles, as in the spec, via the removal of ‘obsolete’ shapes. Both specification and design ultimately define shapes foremost by corners/endpoints.

But the design’s concept of ‘open shapes’ does not map as closely to an element of the specification. It is a strategy used to properly render connected groups of lines and corners.

Special shapes indicated by markup tags are rendered using a straightforward strategy--by replacing the normal rendering logic for those shapes with predefined SVGs or PNGs.

DITAA Implementation

Existing Implementation

DITAA was originally implemented as a headless Java application. The application is built by compiling the Java source into classes and bundling them into a runnable JAR file. The JAR file manifest specifies a main class, CommandLineConverter, which is invoked when the JAR is run. The CommandLineConverter logic is contained mostly in a large main method, which parses command line parameters and then runs the DITAA conversion on either the single input file or the input files referenced in HTML if running with the --html flag.

Classes

Below is a listing of the major classes in the DITAA implementation along with a brief description.

CommandLineConverter:
Entry point for the program. Parses command-line arguments and calls the other major parts of the program to turn ASCII art into bitmaps. This primarily entails constructing a TextGrid from the input, using it to construct a Diagram, and feeding that Diagram to a BitmapRenderer.

ConfigurationParser
Reads and parses a config.xml. Used by ConversionOptions

ConversionOptions
A struct holding miscellaneous options. Holds a ProcessingOptions and a RenderingOptions.

DebugUtils
Can get the line number currently being executed.

DocBookConverter
Unused, and deleted from our version of DITAA.

FileUtils
Boilerplate for handling file/path names.

HTMLConverter
Has only one public method, convertHTMLFile, which loops through an HTML document, converts any ASCII art found in tags labeled “textdiagram”, and creates a new version of the HTMLFile where those diagrams are real pictures.

JavadocTaglet
Allows ditaa diagrams to be involved in javadoc comments. Not actually used by any other part of the current implementation.

Pair
Generic class representing a 2-tuple.

PerformanceTester
Contains a main() which tests how fast DITAA is.

ProcessingOptions
A struct with miscellaneous options about how TextGrids will be turned into Diagrams, but also other eclectic settings of the program--basically anything that doesn’t go in RenderingOptions.

RenderingOptions
Small struct with options affecting how Diagrams are drawn by BitmapRenderer. Examples include whether to use drop shadows and anti-aliasing.

Shape3DOrderingComparator
Comparator that allows DiagramShapes to be compared for sorting purposes. Used by BitmapRenderer to make sure the right things are rendered on top of the right things.

VisualTester
Has a main() which runs some tests.

BitmapRenderer
One of the most important classes. Responsible for taking Diagrams and turning them into bitmaps.

CompositeDiagramShape

A DiagramShape which wraps an array of other DiagramShapes. This is used in practice to represent ‘spiderwebs’, or connected components of open lines with intersections and multiple dead ends.

CustomShapeDefinion
A small struct which encodes how to render a custom (user-defined) shape, usually by pointing to an SVG file somewhere.

Diagram
The god class of DITAA. Contains the bulk of the code for actually processing input. Once constructed, this class exposes collections of DiagramShapes and TextObjects which can be used by clients like BitmapRenderer to easily draw any diagram; the constructor is enormous, and does most the heavy-lifting of taking raw ASCII text and turning it into DITAA’s internal representation of diagrams. We do a lot of work on this class, including refactoring and several bug fixes.

DiagramComponent
The base class of all ‘parts’ of a Diagram. Once the input is processed, a diagram basically comprises a collection of these.

DiagramShape
A DiagramComponent that represents shapes (non-text entities, basically) that can exist in a diagram. Has a should-be-an-enum indicating its type--an arrow, a trapezoid, etc. Can also return a ‘renderPath’, an outline essentially, which is most of what you need to know to draw a given shape.

DiagramText
A DiagramComponent representing a block of text.

FontMeasurer
Figures out how big strings will appear, in pixels, when rendered in a certain font. Used by Diagram.

ImageHandler
Singleton class that has to do with making SVGs. Used by BitmapRenderer. Handles pre-defined shapes like ‘trapezoids’ that are packaged with DITAA as SVGs.

OffscrenSVGRenderer
Renders SVGs.

ShapeEdge
A (directed, though it may not always matter) edge belonging to a particular DiagramShape.

ShapePoint
An x,y point with some methods for checking geometric properties about it.

AbstractCell
This is used by Abstraction Grid to create ‘upsampled’ representations of TextGrids. It has 3x3 int[] representations for various shapes.
A horizontal line, for example, is
000
111
000

Whereas a cross is
010
111
010

AbstractionGrid
When you instantiate an AbstractionGrid, it takes a TextGrid and creates an ‘upsampled’ 3x3 representation of it. Every character in the original TextGrid is represented by a 3x3 block of characters which are either blanks or asterisks.

This allows flood-filling algorithms to detect the space in between adjacent line or corner characters. It also allows for the adjacency to adjacent characters to be detected intelligently: two adjacent pipes “||” are not part of the same component, because the AbstractionGrid reveals that they don’t actually “touch” the way two dashes “--” do.

It’s actually rather clever, and a crucial part of the logic in the Diagram constructor.

CellSet
A set of TextGrid.Cells. Despite the simple premise, this is a relatively large class, as it contains many methods for performing both manipulations of the set and querying high-level properties, such as whether the set represents an open or closed curve. A non-trivial chunk of the logic for processing ASCII inputs ultimately lives in this file.

GridPattern

This is essentially a regular expression for 2D text, which can be used to detect certain multi-character shapes in TextGrids.

GridPatternGroup
A static library of predefined GridPatterns for the kinds of patterns DITAA is interested in, such as 'dead ends' and different orientations of corners.

StringUtils
Simple utilities for Strings.

TextGrid
While we believe Diagram is DITAA's "God class", TextGrid is physically the largest class in DITAA (with over 100 methods!).
In a nutshell, it represents a 2D grid of ASCII text. But it also contains methods for all sorts of complex manipulations and queries on this grid, ultimately containing a significant chunk of DITAA's logic.

Maintainability Improvements

In re-writing to improve maintainability, we refactor Diagram and BitmapRenderer, primarily by decomposing their largest methods into several methods of more manageable size, and documenting existing methods with contracts. In the process, we fix several bugs in Diagram resulting from the interaction of all the shape processing strategies (these fixes are detailed in Testing).

We make a public CellSet method (addAll) private because clients should only use addSet instead.

We remove unused code from BitmapRenderer.

We add asserts reflecting the contracts in Diagram and BitmapRenderer where feasible.

Other opportunities that were identified for improvement but not completed include:
- Turning the ints in DiagramShape into actual enums
- Why is the factory for closed shapes in DiagramComponent but the one for open shapes in CompositeDiagramShape? This seems inconsistent.

Java Version

The existing project was implemented against Java 1.6, which is extremely outdated (released in 2006). While this limits the amount of "modern" Java conveniences that are available, it does maximize the compatibility of DITAA in dated environments. As a compromise to retain maximum compatibility, which may be an important requirement for an ascii-based diagramming tool, the project was only incremented a single version to use Java 1.7 (released in 2011). This allows use of now-standard constructs such as try-with-resource blocks and handling multiple exceptions in a single catch clause.

Implementation of New Plugins

New Color Codes

Human-Readable color codes in DITAA are implemented using an in-code static dictionary from human-readable codes to hex RGB codes. This dictionary is statically initialized in the TextGrid class. We expand this table with additional entries using the same syntax.

GUI

Overview

The existing DITAA application is a headless application with a command line interface. To increase the usability of the tool, a GUI was developed. The GUI is accessed in one of two ways, either by running DITAA with no arguments (no input filename) or by running DITAA with the --gui/-g argument. The latter method supports the use of additional arguments to DITAA which will be used by the GUI each time a new source file is loaded.

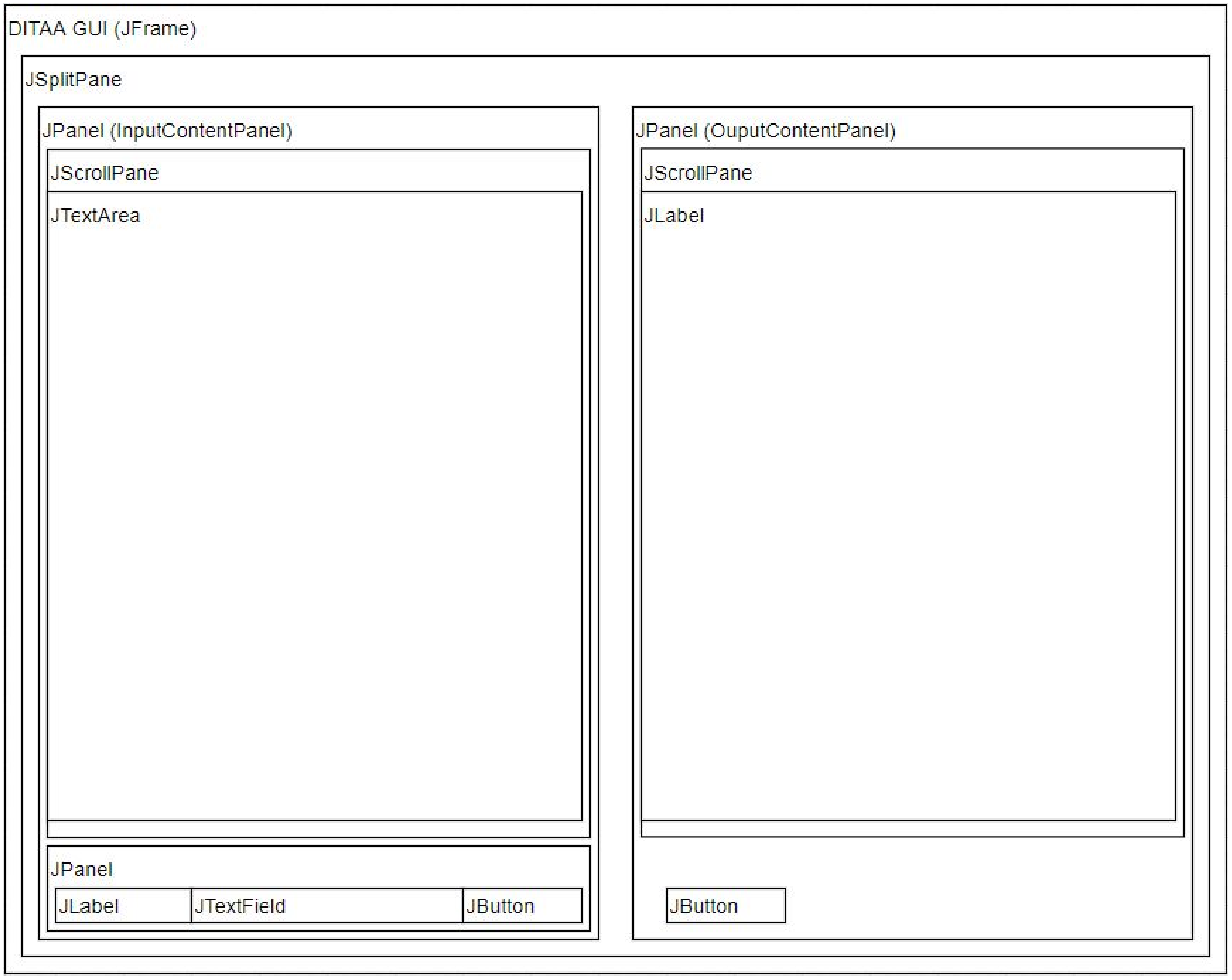
A single new class called DitaaGUI was added to encapsulate the GUI. The logic contained in the GUI is all private to this class and should not be exposed. The only public-facing method in the class is the method which shows the application, openGUI().

Creating and opening the GUI can easily be done in a single line, e.g.

```
new DitaaGUI(options).openGUI();
```

Swing Components

The GUI is implemented using the Swing library. This library provides a number of widget implementations (called “Components”) that can be used to build GUIs. The DITAA GUI implementation is a relatively simple arrangement of Components:



A JFrame contains the entire GUI in an external window. Within that JFrame is a JSplitPane, which provides the left (input) and right (output) content sections. Each section is contained within its own JPanel. The input text preview is provided by a JTextArea and the output image preview is provided by a JLabel. Each of these are wrapped in a JScrollPane which is configured to provide scrollbars if the content size exceeds the container size.

Integration with DITAA

It was desirable for the implementation of the GUI to rely as much as possible on the existing DITAA implementation. That is, care was taken to ensure that code was not duplicated from the existing CommandLineConverter class. This guarantees that as the command-line version of DITAA is updated, those updates are carried over and function equivalently in the GUI version.

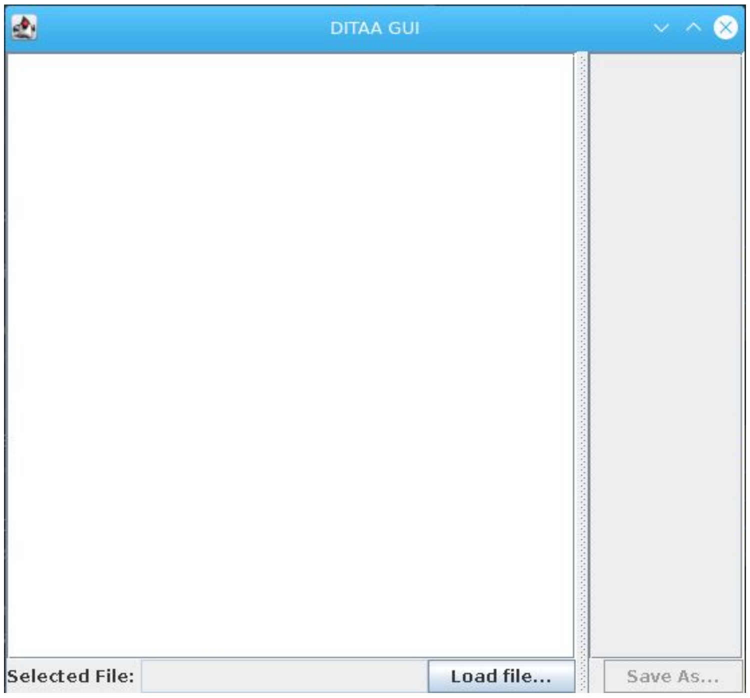
The CommandLineConverter class contained a main function that was run when invoking the DITAA JAR headlessly. Unfortunately, this main function contained a lot of the logic of parsing command line arguments, setting options, and then running DITAA. In order to re-use this implementation, the logic involved with invoking DITAA and saving the output was moved to a new runSimpleMode method. (In this context, “simple mode” indicates running DITAA without the --html option, which is incompatible with the GUI since it does many things at once).

The existing main function was augmented to include the new ‘--gui’ argument and corresponding usage description as well as the logic to invoke the GUI if the option was provided in the command line arguments.

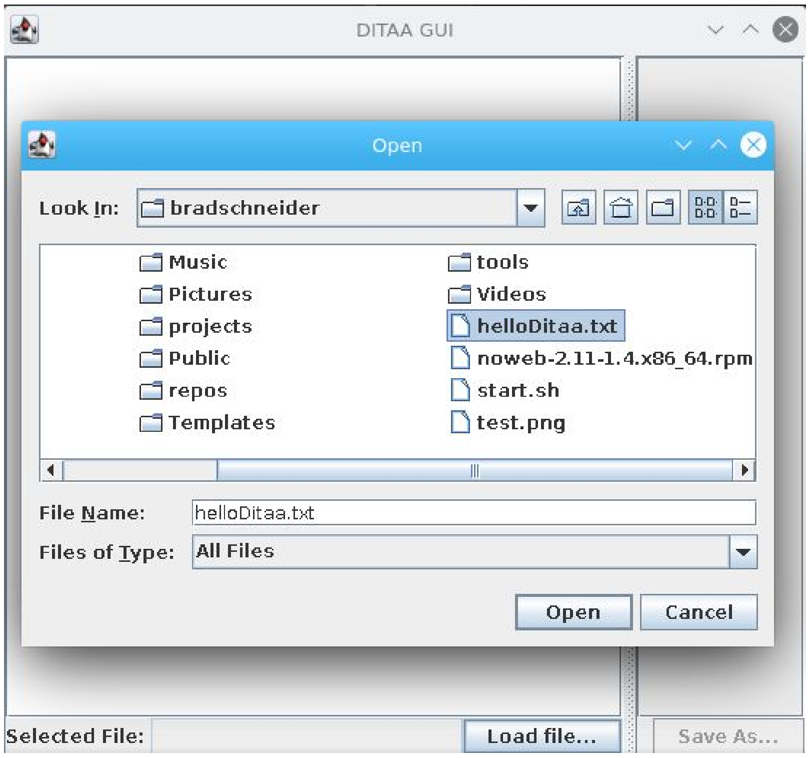
With these changes, it was possible for the GUI implementation to run DITAA programmatically in the exact same way that the headless interface does, maximizing code re-use and maintainability. Another option would have been to implement a completely separate GUI executable which delegated DITAA processing to a compiled version of DITAA (e.g. by running a process which called the JAR file in the background), but this architecture suffers from limited error information being available between processes and was avoided.

Usage

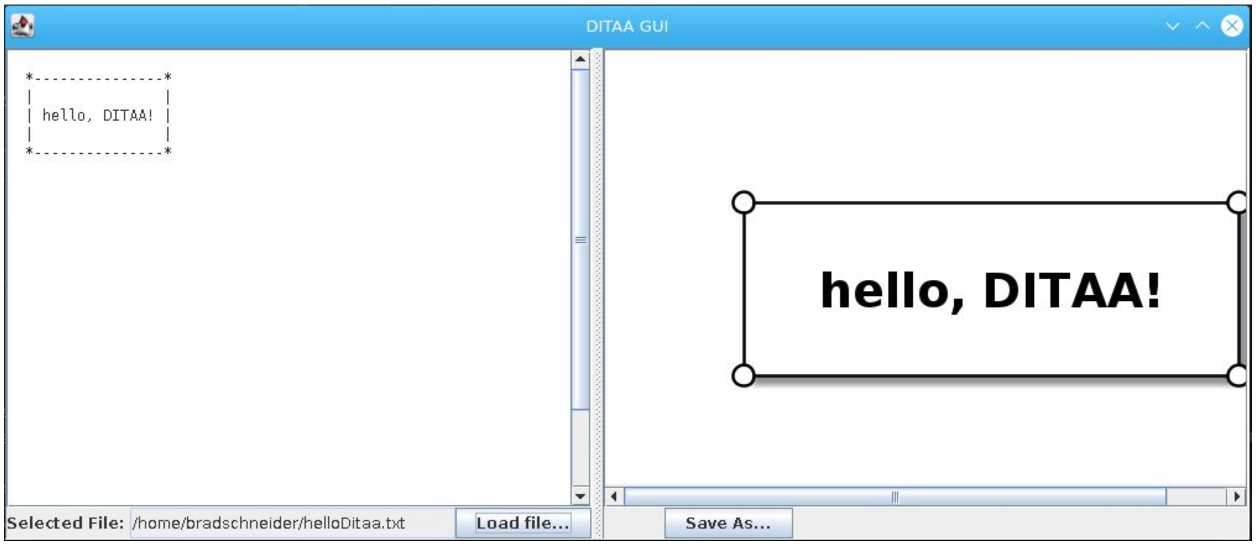
The GUI opens to blank input/output panels as shown below:



The “Load File” and “Save As” buttons are implemented with standard Swing FileChoosers, which are familiar to most users:



Once a text file is loaded using the “Load file” button, it’s contents are displayed in the file preview pane on the left. DITAA is automatically run in the background and the output is saved to a temporary directory for preview, which is loaded in the panel on the right side of the layout:

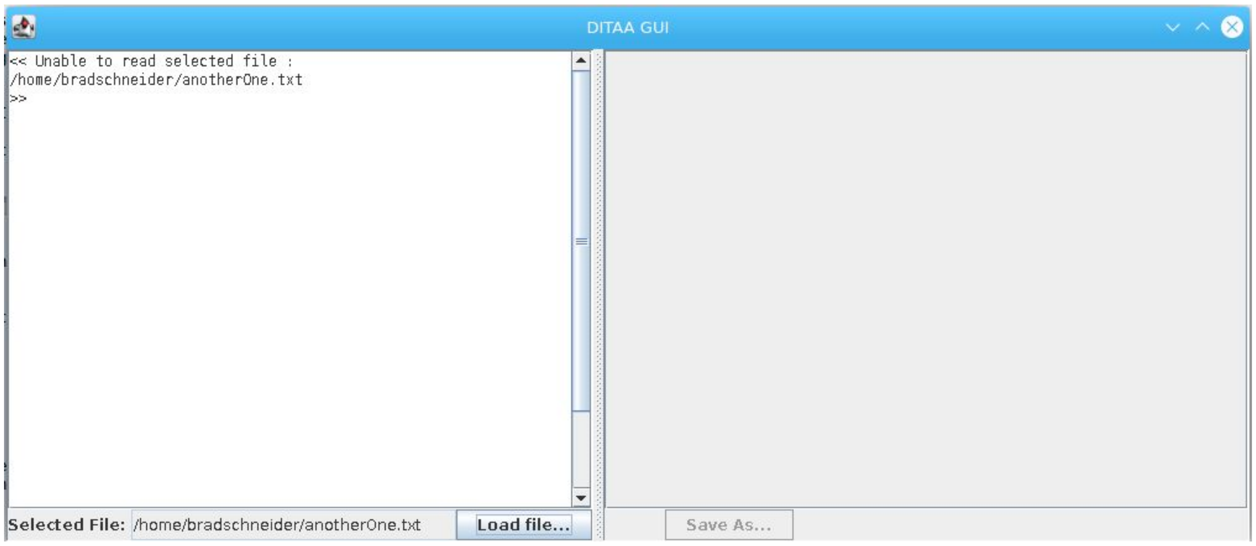


```
*---------------*
|               |
| hello, DITAA! |
|               |
*---------------*
```

It is worth noting that both the input and output preview areas are scrollable if they exceed the window’s current size, and the divider between the two panels is adjustable so that the ratio between the left and right panes is able to be modified for easier viewing.

The output images are computed in a temporary file space. If the user wishes to save the output graphic, the "Save As" button will allow the user to select a location to save the graphic to and provide a name for the output file.

In the event that there is an error opening or reading the selected file, an error message is displayed in the input file preview pane:



```
<< Unable to read selected file :
/home/bradschneider/anotherOne.txt
>>
```

Building and Running

The existing DITAA project can be built using Apache Ant, a build configuration tool. No additional configuration is needed to build the added features.

See Appendix A: File Listing for a list of source files included in the project.

On your local machine, navigate to a desired directory. Clone the git repository to your local machine or unzip the source tarball:

```
>> cd /home/username/repositories
>> git clone https://github.com/CJMenart/CS-7140-2020.git
```

Or

```
>> tar -zxvf ditaa_source.tar
```

Navigate to the build directory:

```
>> cd CS-7140-2020/DITAA/build
```

Run Ant using the provided release.xml file:
```
>> ant -f release.xml
```

This compiles the Java code and produces a new subdirectory in the DITAA directory called ‘releases’. This will contain the compiled runnable JAR file.
```
>> cd ../releases
>> java -jar ditaa0_9.jar
```

This will open the DITAA GUI. Additional command line arguments may be provided as long as the --gui command is also provided. For example, the following runs the GUI and passes the scale argument to DITAA for all files run in the GUI:
```
>> java -jar ditaa0_9.jar --gui --scale 2.5
```

Smoke Testing

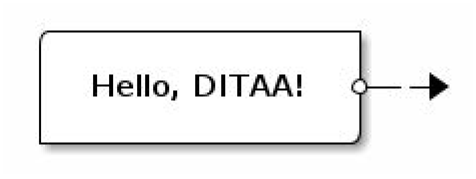
A quick ‘smoke test’ can be performed on the software after it has been built. This section documents the recommended smoke test procedure. This is not intended to be a complete test of the system, but is a quick check that it is operating.

Setup

Before smoke testing, create a small sample text file called helloDitaa.txt with the contents below:

```
/---------------+
|               |
|  Hello, DITAA *---->
|               |
+---------------/
```

The expected output for this file is the following diagram:



Test Case 1: Load Valid File

1. Run DITAA as described in the ‘Building and Running’ Section
   ```
   >> java -jar ditaa0_9.jar
   ```

    a. Confirm that the DITAA GUI is displayed
2. In the DITAA GUI, load the helloDitaa.txt file that was created in setup.
    a. Click the “Load file...” button
    b. In the file chooser, navigate to the helloDitaa.txt file and click “open”
    c. Confirm that
        i. Contents of the file are loaded in the left pane
        ii. DITAA diagram is displayed in the right pane
        iii. DITAA diagram matches the expected output
3. Save the output diagram
    a. Click the “Save As…” button
    b. In the filie chooser, navigate to a directory of choice
    c. Type a file name into the box, e.g. ditaaDiagram.png
    d. Confirm that
        i. The file containing the diagram has been created on the filesystem in the chosen location

DITAA Testing

Overview

This document describes tests that were implemented as part of the analysis and extension of DITAA. Various types of testing were implemented and are documented in the following sections.

Our Contributions

- JUnit tests for four additional classes in DITAA
- A total of 14 additional text-file test cases, or ASCII diagrams which DITAA can be invoked on through the command line. This is the format of most of the pre-existing implementation's tests, and they are tested through visual inspection. They include our stress tests, and tests created in the course of identifying and repairing bugs in DITAA.
- Manually-conducted smoke testing and testing of the GUI we implement.

Acceptance Testing

DITAA previously had very few unit tests and no automated method of integration testing. A manual procedure could be developed to perform acceptance testing, but in the case of DITAA there is a single main concern with accepting a release--the correctness of the diagrams that the software generates. Evaluating the correctness of a set of diagrams is easily accomplished through an automated test.

Dataset

Acceptance testing is performed on 30 different inputs of varying complexity. The inputs have been designed to test each entity that DITAA supports on its diagrams, e.g. boxes, lines, dashed boxes, dashed lines, special shapes, arrows, color codes, rounded corners, bulleted lists, free text, etc. The diagrams range from a single shape to a complex diagram of several shapes, arrows, and different colors.

Implementation

The JUnit framework was used to develop an automated test (AcceptanceTestSuite.java) that submits every test text file in the tests/text project directory to DITAA. For each text file that is submitted, a corresponding PNG is also supplied that represents the expected output. These expected outputs have been verified manually. Adding a new test case requires two simple steps: 1) adding the test input file, and 2) adding the expected output diagram file. The automated test is implemented such that it will automatically include the new files when the test suite is run.

For each test input file, DITAA output is generated and stored in a temporary location on the filesystem. An MD5 hash is computed for both the actual and expected output graphic files and the assertion is made that these hashes are equivalent. While MD5 may be vulnerable to exploits, it is extremely unlikely that the hash of two different files will collide unless they are specifically designed to exploit such a vulnerability.

Testing for New Plugins

Additional Color Codes

Test case 32 was created to test 6 new color codes added to DITAA. It was evaluated visually, as most of the test cases are, and based on the results, some changes were made to the color codes' RGB values.

GUI Implementation

The GUI implementation was carefully architected to share the same codebase performing the DITAA ASCII-to-Bitmap with the command line interface. In this regard, all of the above tests are relevant to the GUI implementation with respect to the diagrams that are produced.

Testing the added functionality of the GUI widgets is a difficult task due to the need to mock a large number of widget classes. This is not generally feasible or desirable, so frameworks exist to simulate clicks and typing on the application. However, no such framework was integrated for this project.

If an automated GUI testing technology was to be used for testing the DITAA GUI, there would be only a few main features to test. Since we assume Swing widgets work as advertised and the layout of the application is 'static', i.e. a single window, single view, no menus, etc., the testing is simplified and limited to the following verifications:

1. The GUI starts when
    a. DITAA is run with no arguments
    b. DITAA is run with the --gui or -g argument
2. When no file (or an unreadable file) is selected, the Save As button is disabled.
3. When the Load file button is selected, a file chooser is presented
4. When a file is chosen as an input,
    a. The file contents populate in the left pane
        i. If the file is not 'valid', an error is shown instead
    b. Assuming a 'valid' input, (within an appropriate amount of time) the output diagram populates in the right pane
    c. Assuming a 'valid' input, the Save As button is enabled
5. When the Save As button is selected, a file chooser is presented
6. Choosing a file in the Save As file chooser results in the file being saved to the specified location.

Test Cases that Fail

There are clearly additional bugs in the DITAA codebase as several of the tests currently fail. While it is unfortunate that the cases are failing, it demonstrates some success by our testing efforts in identifying broken functionality within the application.

- In test case 8, a point marker is not rendered.
- In test case 10, a line is not connected to the document it clearly connects to in the ASCII. The document has a special shape (document), which doubtless makes this difficult to ensure.
- In test case 13, there are barely-visible 'extra lines' continuing past where the line is supposed to curve into a corner.
- In test case 22, numerous point markers are not rendered.
- In test case 27, an incorrect break appears in a line joining two squares.
- In test case 28 the same incorrect break occurs.
- In case "Bug 10", one horizontal line is not aligned with other horizontal lines beginning in the same column of the ASCII.
- In "Bug 13", the same misalignment occurs.
- All of the cases named "OutOfMemoryError", except for the version named ".edit", appear to result in indefinite hanging. We are not so sure this is actually an issue of running out of memory, as the stress tests actually result in a complaint from the JVM but these cases do not.

- Case “Garbage” results in a RuntimeException.
- The “Stress Test” and “Less Stress Test” result in OutOfMemory errors.

Stress Testing

To perform stress testing on DITAA requires diagrams with a large number of entities, i.e. a large input file. To accomplish this stress testing, the acceptance test dataset includes three files considered to be abnormally large - tiny_stress_test.txt (30KB), less_stress_test.txt (300KB), and stress_test.txt (29,893KB).

We create the test case ‘Stress Test’ by tiling Test Case 1 to create a file 10,000 times the size of Test Case 1. Attempting to run DITAA with this test input results in an OutOfMemory error from the JVM. We progressively scale down this test, creating ‘Less Stress Test’, and ‘Tiny Stress Test’, the last of which is only 10 times the size of Test Case 1, and successfully runs to completion in 67 seconds on a powerful home workstation.

We conclude that DITAA is not architected around the possibility of large inputs. In particular, the creation of numerous intermediate AbstractionGrids is likely a source of great memory inefficiency, and the execution of repeated flood-fills makes the program inefficient in time on large ASCII art as well.

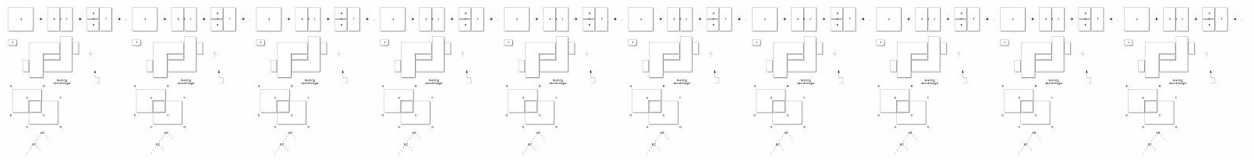


Figure 1: The (correct) rendered output of the Tiny Stress Test, the largest input we successfully process with DITAA

Unit Testing

The DITAA baseline was provided with very few existing unit tests. IDEA generated the following coverage report prior to our modifications and additions:

Package	**Class, %**	**Method, %**	**Line, %**
all classes	27.3% (12/ 44)	16.8% (107/ 637)	17.9% (777/ 4351)

Coverage Breakdown

Package	Class, %	Method, %	Line, %
org.stathissideris.ascii2image.core	5.9% (1/ 17)	0.9% (1/ 113)	1.7% (13/ 781)
org.stathissideris.ascii2image.graphics	0% (0/ 12)	0% (0/ 225)	0% (0/ 1688)
org.stathissideris.ascii2image.test	100% (3/ 3)	100% (21/ 21)	100% (169/ 169)
org.stathissideris.ascii2image.text	66.7% (8/ 12)	30.6% (85/ 278)	34.7% (595/ 1713)

Additions

The pre-existing implementation has some unit tests for the TextGrid, CellSet, and GridPattern classes. We write additional unit tests for:

- DebugUtils
- Pair
- ConversionOptions
- DiagramShape

Of these, DiagramShape is the most significant and complex class, and receives the most detailed unit tests. DiagramShape's non-trivial public methods are all assigned a test.

Results

With our additions to the unit tests, the code coverage was improved greatly. The overall coverage was increased by 2.7 times when counted by class, and more than tripled when counting by method and line. We were able to achieve 75% coverage in the graphics package which previously had no unit test coverage.

The following coverage metrics were generated by IDEA with our added unit tests:

Package	Class, %	Method, %	Line, %
all classes	73.5% (36/ 49)	60.3% (396/ 657)	61.8% (2780/ 4499)

Coverage Breakdown

Package	Class, %	Method, %	Line, %
org.stathissideris.ascii2image.core	47.1% (8/ 17)	20.2% (23/ 114)	9.7% (78/ 806)

org.stathissideris.ascii2image.graphics	75% (9/ 12)	61.7% (140/ 227)	67.1% (1179/ 1756)
org.stathissideris.ascii2image.test	100% (8/ 8)	100% (38/ 38)	100% (221/ 221)
org.stathissideris.ascii2image.text	91.7% (11/ 12)	70.1% (195/ 278)	75.9% (1302/ 1716)

The low coverage for the core package is due in part to unused classes, such as JavadocTaglet and VisualTester, and partially-implemented functionality such as custom-shape config files, which is also unused.

Bug Discovery and Repair

Mixed Shape Bug(s)

During the Design phase, we discovered a possible bug in the Diagram constructor--the handling of mixed shapes may fail under certain circumstances. We introduce new test cases to confirm the existence of this bug and clarify its nature, and in three rounds of testing and code modification, correct the behavior of DITAA by re-designing the aforementioned section of the method.

Our original understanding of the bug is that mixed shapes are decomposed using one of two strategies in the original implementation. They are decomposed using the logic ‘subtract all closed shapes from mixed shapes’, whenever any closed shapes are present, and using an ‘advanced’ algorithm that relies on tracing when there are not. But that may not always be appropriate. We introduce test case 23, which confirms that the ‘advanced algorithm’ logic works under the example circumstance given by the author:

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

And test case 24 confirms that, if a separate closed shape is introduced, the pre-existing implementation fails because the advanced algorithm is not used:

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

In this case, the larger shape in the diagram disappears entirely, because it is never successfully decomposed into a series of open and closed shapes.

We also develop test case 25, which shows the same error occurring even in one large connected component:

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

And the slightly different test case 26 demonstrates that when the ‘advanced’ algorithm is triggered by ensuring that no simple closed shapes exist, the shape is handled correctly. This test case passes in the pre-existing implementation:

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

Our first notion, then, is that the ‘advanced’ mixed-shape handler appears to work in all cases, and was added after the method of subtracting closed shapes was discovered not to work in the case where no closed shapes were present in the diagram. We believe we can simplify the code by simply removing the branch which subtracts closed shapes and always uses the advanced algorithm.

After making this change, we test the new version of the software. The new test cases that we have devised pass. However, we also run a regression test by running the older, existing test cases--and test case 3 (reproduced here) fails!

```
           +--+
 +-----+-->|  |
 |     |   +--+
 |     |
 |     |
+-+-+ +-+-+
|   | |   |
|   | |   |
+---+ +---+
```

Something about two closed shapes joined by a line appears to cause trouble! We introduce test cases 27, 28, and 29 to clarify this behavior:

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

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

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

These test cases all fail in the implementation current at this stage. Test case 29 causes significant confusion, as in the implementation at this stage, multiple copies of shape sections are drawn! This will eventually turn out to be caused by a closely-related bug that we will have to fix in the course of repairing our original bug. But initially, we are unaware of this, and create test case 30 to help clarify:

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

Test case 30 passes, as well it should. There is something problematic about the existence of extra closed shapes, it seems?

We also have to closely study the implementation of the ‘advanced’ method, CellSet.breakTrulyMixedBoundaries(). This is essentially a tracing method, which pulls out open shapes by starting to trace at ‘dead-ends’ and stopping when it reaches intersections. Test case 3, then, can be explained as a failure because there are multiple closed cells in a mixed shape (the one created by an outside-in fill in Step 2) but only one little dead-end, from which tracing stops before any significant part of the shape can be processed.

There is an immediate impulse to use the subtraction of closed shapes *followed* by the ‘advanced’ algorithm. Indeed, this would cause test case 3 to pass. But we should beware of ‘blindly’ making changes to ensure our existing test cases pass, without fully understanding why. Test cases may not--and most often do not--cover the entire possibility space of inputs to a program, and cannot substitute for careful reasoning.

Indeed, our understanding of the code indicates that this hotfix would not give correct behavior on test case 27 (above). There are no simple closed shapes to be subtracted there.

However, we reason, if the ‘advanced’ algorithm was run *first*, and the closed shapes found as a *result* were subtracted from the remaining cells...then the proper set of closed and open shapes could be found.

We want to do this without breaking cases like case 26 (above), where subtracting one closed shape could destroy another.

Based on this, we believe we can achieve fully-correct mixed shape processing with a modest re-design to this part of the algorithm. We will iterate over both steps of mixed-shape processing in a do-while loop--returning to the start of the loop as long as any mixed shapes remain that haven’t been broken down--and additionally, rather than subtracting all closed shapes willy-nilly, we will ‘subtract’ only closed shapes that are *subsets* of a larger mixed shape under consideration.

(This also allows us to remove the large copy-pasted block of ‘classify all shapes’ logic, so that it exists only in one place, making for cleaner code.)

We cannot think of any diagram that should be processed incorrectly by this design. Any set of cells must have either dead ends or simple closed shapes that can be identified via a flood fill. By continuing to pull out those shapes, we should always be able to break the diagram down into Closed and Open shapes.

Modifying the code as described results in most test cases passing--but, again, not all. Cases 27 and 29, for example, still fail. Upon seeing the output, we develop another intuition and create test case 31 to test it:

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

Case 31 passes despite the failure of case 29. We finally realize what’s happening--there is a (separate) bug in the logic of breakTrulyMixedBoundaries() that destroys closed shapes by removing intersection characters--but only if the intersection is attached to a stub exactly 1 character long. By adding an extra check, this is prevented, and at last, all test cases pass!

The Wasteful Flood Fill Bug

In the process of refactoring Diagram by decomposing its large constructor into smaller methods, we began regression testing to ensure that behavior was unchanged and found that test case 1 (the very first!) appeared to result in hanging for an excessive period of time.

This behavior was tracked to findInteriorExteriorBoundaries(), where a comment by the original author indicated that there may be a previously-unfixed bug resulting in findBoundariesExpandFrom() returning empty lists when it was not expected to.

Further inspection revealed that this was, indeed, the source of the problem. The outer loops of findInteriorExteriorBoundaries() are intended to loop over every cell of an AbstractionGrid representing part of the input art, in order to try flood-filling a given shape from all possible locations. However, the bounds of this loop originally used this.width and this.height, which do NOT represent the height and width of that AbstractionGrid in ‘cells’, but the size of the entire Diagram in *pixels*. These loop bounds were vastly higher than they were supposed to be, and leading to huge numbers of unnecessary loop iterations. This is why variable names should give some indication of units!

We replaced these with the correct loop bounds, and the excessive execution time of findInteriorExteriorBoundaries() was no more.

Discussion and Conclusion

Many clear and fruitful avenues exist for future work. In terms of features, the incomplete functionality for custom shape config files would ideally be either fully implemented or deleted from the code. The class hierarchy of large classes such as CellSet would be examined with greater scrutiny. There are additional miscellaneous refactors we would like to perform given time, such as the replacement of integer constants by actual enumerated constants, or a more sensible organization to the static factory methods for DiagramComponents (currently, the factories for DiagramShape are located in DiagramComponent, for reasons unknown).

JUnit tests would be written for all of DITAA's classes, and would more effectively exercise edge cases in the software while achieving better code coverage. Similarly, documentation and contracts would be extended to cover the entire codebase.

Our existing contracts could easily be improved further, given ample time; they are currently given in English prose, prose which, for that matter, is not as precise as it could be. Even better would be contracts given using unambiguous mathematical notation.

Due to time constraints, we were unable to execute the planned reduction in the size of the codebase. After the bug repairs and features added to the code, our SLOC analyzer reports 10,127 lines of code, an increase of approximately 500 lines. These extra lines of code come primarily from the new DitaaGUI class. (Although we believe the analyzer is also counting our JUnit tests, adding a 'spurious' 100-200 lines in this count. At any rate, we left the program larger than we found it, to our chagrin!)

We were also unable to implement the features originally planned as seen in the project Requirements and Specifications. The features implemented instead (which are the ones discussed from 'Design' onward) are the ones specified by this alternative specification document: <Specification-Doc/CS 7140 - Specification Document Group 3.pdf at main · CS-7140-Abhishek-Daniel-Griffin-Lance/Specification-Doc (github.com)>

Still, we believe that we have managed to make a noticeable improvement to DITAA and eased the way for any hypothetical future maintainers wishing to expand, alter, or finish its incomplete features. Our collective documentation and refactoring efforts make the core of DITAA--its most complex and interconnected classes--vastly easier to understand. Our bugfixes are in some of the most complex sections of the code, and a test case like our stress test never would have been processed successfully without both of them repaired.

Ultimately, we believe we have successfully demonstrated how, with sufficient time, increased rigor can be brought to the design of this example software.

Appendix A: File Listing

```
.:
total 56K
drwxr-xr-x. 2 bradschneider 4.0K Nov 29 20:57 build
drwxr-xr-x. 3 bradschneider 4.0K Nov 29 10:45 classes
-rw-r--r--. 1 bradschneider  18K Nov 29 10:45 COPYING
-rw-r--r--. 1 bradschneider 6.6K Nov 29 10:45 ditaa.iml
-rw-r--r--. 1 bradschneider  448 Nov 29 10:45 HISTORY
drwxr-xr-x. 2 bradschneider 4.0K Nov 29 10:45 lib
drwxr-xr-x. 2 bradschneider 4.0K Nov 29 20:43 releases
drwxr-xr-x. 5 bradschneider 4.0K Nov 29 10:45 src
drwxr-xr-x. 3 bradschneider 4.0K Nov 29 10:45 tests

./build:
total 12K
-rw-r--r--. 1 bradschneider   54 Nov 29 10:45 ejp-filter
-rw-r--r--. 1 bradschneider 4.1K Nov 29 17:01 release.xml

./classes:
total 4.0K
drwxr-xr-x. 3 bradschneider 4.0K Nov 29 10:45 artifacts

./classes/artifacts:
total 4.0K
drwxr-xr-x. 2 bradschneider 4.0K Nov 29 15:54 ditaa_jar

./classes/artifacts/ditaa_jar:
total 22M
-rw-r--r--. 1 bradschneider 22M Nov 29 15:54 ditaa.jar

./lib:
total 4.5M
-rw-r--r--. 1 bradschneider 361K Nov 29 10:45 batik-awt-util.jar
-rw-r--r--. 1 bradschneider 272K Nov 29 10:45 batik-bridge.jar
-rw-r--r--. 1 bradschneider 233K Nov 29 10:45 batik-css.jar
-rw-r--r--. 1 bradschneider  86K Nov 29 10:45 batik-dom.jar
-rw-r--r--. 1 bradschneider  46K Nov 29 10:45 batik-extension.jar
```

```
-rw-r--r--. 1 bradschneider  75K Nov 29 10:45 batik-ext.jar
-rw-r--r--. 1 bradschneider 135K Nov 29 10:45 batik-gui-util.jar
-rw-r--r--. 1 bradschneider 167K Nov 29 10:45 batik-gvt.jar
-rw-r--r--. 1 bradschneider  37K Nov 29 10:45 batik-parser.jar
-rw-r--r--. 1 bradschneider  37K Nov 29 10:45 batik-script.jar
-rw-r--r--. 1 bradschneider 417K Nov 29 10:45 batik-svg-dom.jar
-rw-r--r--. 1 bradschneider 161K Nov 29 10:45 batik-svggen.jar
-rw-r--r--. 1 bradschneider 136K Nov 29 10:45 batik-swing.jar
-rw-r--r--. 1 bradschneider  62K Nov 29 10:45 batik-transcoder.jar
-rw-r--r--. 1 bradschneider  80K Nov 29 10:45 batik-util.jar
-rw-r--r--. 1 bradschneider  20K Nov 29 10:45 batik-xml.jar
-rw-r--r--. 1 bradschneider  30K Nov 29 10:45 commons-cli-1.0.jar
-rw-r--r--. 1 bradschneider  41K Nov 29 10:45 commons-cli-1.2.jar
-rw-r--r--. 1 bradschneider  57K Nov 29 10:45 jericho-html-1.4.jar
-rw-r--r--. 1 bradschneider 584K Nov 29 10:45 js.jar
-rw-r--r--. 1 bradschneider 473K Nov 29 10:45 pdf-transcoder.jar
-rw-r--r--. 1 bradschneider 913K Nov 29 10:45 xerces_2_5_0.jar
-rw-r--r--. 1 bradschneider 105K Nov 29 10:45 xml-apis.jar

./releases:
total 0

./src:
total 12K
drwxr-xr-x. 3 bradschneider 4.0K Nov 29 10:45 au
drwxr-xr-x. 2 bradschneider 4.0K Nov 29 10:45 META-INF
drwxr-xr-x. 3 bradschneider 4.0K Nov 29 10:45 org

./src/au:
total 4.0K
drwxr-xr-x. 3 bradschneider 4.0K Nov 29 10:45 id

./src/au/id:
total 4.0K
drwxr-xr-x. 3 bradschneider 4.0K Nov 29 10:45 jericho

./src/au/id/jericho:
total 4.0K
drwxr-xr-x. 3 bradschneider 4.0K Nov 29 10:45 lib
```

```
./src/au/id/jericho/lib:
total 4.0K
drwxr-xr-x. 2 bradschneider 4.0K Nov 29 10:45 html

./src/au/id/jericho/lib/html:
total 336K
-rw-r--r--. 1 bradschneider 5.7K Nov 29 10:45 Attribute.java
-rw-r--r--. 1 bradschneider  16K Nov 29 10:45 Attributes.java
-rw-r--r--. 1 bradschneider 2.4K Nov 29 10:45 BlankOutputSegment.java
-rw-r--r--. 1 bradschneider  64K Nov 29 10:45 CharacterEntityReference.java
-rw-r--r--. 1 bradschneider  17K Nov 29 10:45 CharacterReference.java
-rw-r--r--. 1 bradschneider 2.8K Nov 29 10:45 CharOutputSegment.java
-rw-r--r--. 1 bradschneider 7.0K Nov 29 10:45 Element.java
-rw-r--r--. 1 bradschneider 6.4K Nov 29 10:45 EndTag.java
-rw-r--r--. 1 bradschneider  16K Nov 29 10:45 FormControlType.java
-rw-r--r--. 1 bradschneider  11K Nov 29 10:45 FormField.java
-rw-r--r--. 1 bradschneider  11K Nov 29 10:45 FormFields.java
-rw-r--r--. 1 bradschneider 6.0K Nov 29 10:45 IntStringHashMap.java
-rw-r--r--. 1 bradschneider 2.3K Nov 29 10:45 IOutputSegment.java
-rw-r--r--. 1 bradschneider  11K Nov 29 10:45 NumericCharacterReference.java
-rw-r--r--. 1 bradschneider 5.4K Nov 29 10:45 OutputDocument.java
-rw-r--r--. 1 bradschneider 1.6K Nov 29 10:45 OutputSegmentComparator.java
-rw-r--r--. 1 bradschneider 2.2K Nov 29 10:45 package.html
-rw-r--r--. 1 bradschneider 3.0K Nov 29 10:45 SearchCache.java
-rw-r--r--. 1 bradschneider  14K Nov 29 10:45 Segment.java
-rw-r--r--. 1 bradschneider  21K Nov 29 10:45 Source.java
-rw-r--r--. 1 bradschneider 4.9K Nov 29 10:45 SpecialTag.java
-rw-r--r--. 1 bradschneider  36K Nov 29 10:45 StartTag.java
-rw-r--r--. 1 bradschneider 2.7K Nov 29 10:45 StringOutputSegment.java
-rw-r--r--. 1 bradschneider   20 Nov 29 10:45 stylesheet.css
-rw-r--r--. 1 bradschneider  26K Nov 29 10:45 Tag.java

./src/META-INF:
total 4.0K
-rw-r--r--. 1 bradschneider 92 Nov 29 10:45 MANIFEST.MF

./src/org:
total 4.0K
drwxr-xr-x. 3 bradschneider 4.0K Nov 29 10:45 stathissideris
```

```
./src/org/stathissideris:
total 4.0K
drwxr-xr-x. 6 bradschneider 4.0K Nov 29 10:45 ascii2image

./src/org/stathissideris/ascii2image:
total 16K
drwxr-xr-x. 2 bradschneider 4.0K Nov 29 17:03 core
drwxr-xr-x. 2 bradschneider 4.0K Nov 29 10:45 graphics
drwxr-xr-x. 2 bradschneider 4.0K Nov 29 10:45 test
drwxr-xr-x. 2 bradschneider 4.0K Nov 29 10:45 text

./src/org/stathissideris/ascii2image/core:
total 108K
-rw-r--r--. 1 bradschneider 9.2K Nov 29 15:54 CommandLineConverter.java
-rw-r--r--. 1 bradschneider 6.2K Nov 29 10:45 ConfigurationParser.java
-rw-r--r--. 1 bradschneider 5.8K Nov 29 10:45 ConversionOptions.java
-rw-r--r--. 1 bradschneider  182 Nov 29 10:45 DebugUtils.java
-rw-r--r--. 1 bradschneider 9.1K Nov 29 17:03 DitaaGUI.java
-rw-r--r--. 1 bradschneider 2.0K Nov 29 10:45 DocBookConverter.java
-rw-r--r--. 1 bradschneider 4.4K Nov 29 10:45 FileUtils.java
-rw-r--r--. 1 bradschneider 6.4K Nov 29 10:45 HTMLConverter.java
-rw-r--r--. 1 bradschneider 9.7K Nov 29 10:45 JavadocTaglet.java
-rw-r--r--. 1 bradschneider  193 Nov 29 10:45 Pair.java
-rw-r--r--. 1 bradschneider 1.9K Nov 29 10:45 PerformanceTester.java
-rw-r--r--. 1 bradschneider 4.7K Nov 29 10:45 ProcessingOptions.java
-rw-r--r--. 1 bradschneider 2.2K Nov 29 10:45 RenderingOptions.java
-rw-r--r--. 1 bradschneider 1.7K Nov 29 10:45 Shape3DOrderingComparator.java
-rw-r--r--. 1 bradschneider 5.4K Nov 29 10:45 VisualTester.java

./src/org/stathissideris/ascii2image/graphics:
total 144K
-rw-r--r--. 1 bradschneider  15K Nov 29 10:45 BitmapRenderer.java
-rw-r--r--. 1 bradschneider 9.1K Nov 29 10:45 CompositeDiagramShape.java
-rw-r--r--. 1 bradschneider 2.1K Nov 29 10:45 CustomShapeDefinition.java
-rw-r--r--. 1 bradschneider 4.1K Nov 29 10:45 DiagramComponent.java
-rw-r--r--. 1 bradschneider  30K Nov 29 10:45 Diagram.java
-rw-r--r--. 1 bradschneider  29K Nov 29 10:45 DiagramShape.java
-rw-r--r--. 1 bradschneider 4.0K Nov 29 10:45 DiagramText.java
-rw-r--r--. 1 bradschneider 6.1K Nov 29 10:45 FontMeasurer.java
-rw-r--r--. 1 bradschneider 4.6K Nov 29 10:45 ImageHandler.java
```

```
-rw-r--r--. 1 bradschneider 5.2K Nov 29 10:45 OffScreenSVGRenderer.java
-rw-r--r--. 1 bradschneider 6.2K Nov 29 10:45 ShapeEdge.java
-rw-r--r--. 1 bradschneider 3.1K Nov 29 10:45 ShapePoint.java

./src/org/stathissideris/ascii2image/test:
total 20K
-rw-r--r--. 1 bradschneider 688 Nov 29 10:45 CellSetTest.java
-rw-r--r--. 1 bradschneider 707 Nov 29 10:45 GridPatternTest.java
-rw-r--r--. 1 bradschneider 11K Nov 29 10:45 TextGridTest.java

./src/org/stathissideris/ascii2image/text:
total 112K
-rw-r--r--. 1 bradschneider 3.0K Nov 29 10:45 AbstractCell.java
-rw-r--r--. 1 bradschneider 5.3K Nov 29 10:45 AbstractionGrid.java
-rw-r--r--. 1 bradschneider  18K Nov 29 10:45 CellSet.java
-rw-r--r--. 1 bradschneider  12K Nov 29 10:45 GridPatternGroup.java
-rw-r--r--. 1 bradschneider 8.9K Nov 29 10:45 GridPattern.java
-rw-r--r--. 1 bradschneider 4.8K Nov 29 10:45 StringUtils.java
-rw-r--r--. 1 bradschneider  47K Nov 29 10:45 TextGrid.java

./tests:
total 4.0K
drwxr-xr-x. 2 bradschneider 4.0K Nov 29 10:45 text

./tests/text:
total 432K
-rw-r--r--. 1 bradschneider  16K Nov 29 10:45 art10.png
-rw-r--r--. 1 bradschneider  554 Nov 29 10:45 art10.txt
-rw-r--r--. 1 bradschneider 1.4K Nov 29 10:45 art11.png
-rw-r--r--. 1 bradschneider  148 Nov 29 10:45 art11.txt
-rw-r--r--. 1 bradschneider  132 Nov 29 10:45 art12.txt
-rw-r--r--. 1 bradschneider  492 Nov 29 10:45 art13.png
-rw-r--r--. 1 bradschneider   44 Nov 29 10:45 art13.txt
-rw-r--r--. 1 bradschneider 1.9K Nov 29 10:45 art14.png
-rw-r--r--. 1 bradschneider   73 Nov 29 10:45 art14.txt
-rw-r--r--. 1 bradschneider   42 Nov 29 10:45 art15.txt
-rw-r--r--. 1 bradschneider  176 Nov 29 10:45 art16.txt
-rw-r--r--. 1 bradschneider  280 Nov 29 10:45 art17.txt
-rw-r--r--. 1 bradschneider 6.0K Nov 29 10:45 art18.png
-rw-r--r--. 1 bradschneider  410 Nov 29 10:45 art18.txt
```

```
-rw-r--r--. 1 bradschneider    7 Nov 29 10:45 art19.txt
-rw-r--r--. 1 bradschneider  12K Nov 29 10:45 art1.png
-rw-r--r--. 1 bradschneider 1.7K Nov 29 10:45 art1.txt
-rw-r--r--. 1 bradschneider  795 Nov 29 10:45 art20.txt
-rw-r--r--. 1 bradschneider   12 Nov 29 10:45 art21.txt
-rw-r--r--. 1 bradschneider 2.8K Nov 29 10:45 art22_2.png
-rw-r--r--. 1 bradschneider 2.6K Nov 29 10:45 art22_3.png
-rw-r--r--. 1 bradschneider 2.9K Nov 29 10:45 art22_4.png
-rw-r--r--. 1 bradschneider 2.4K Nov 29 10:45 art22.png
-rw-r--r--. 1 bradschneider  108 Nov 29 10:45 art22.txt
-rw-r--r--. 1 bradschneider  978 Nov 29 10:45 art23.png
-rw-r--r--. 1 bradschneider  130 Nov 29 10:45 art23.txt
-rw-r--r--. 1 bradschneider 1.2K Nov 29 10:45 art24.png
-rw-r--r--. 1 bradschneider  226 Nov 29 10:45 art24.txt
-rw-r--r--. 1 bradschneider 1.2K Nov 29 10:45 art25.png
-rw-r--r--. 1 bradschneider  138 Nov 29 10:45 art2_5.txt
-rw-r--r--. 1 bradschneider  184 Nov 29 10:45 art25.txt
-rw-r--r--. 1 bradschneider 1.3K Nov 29 10:45 art26.png
-rw-r--r--. 1 bradschneider  184 Nov 29 10:45 art26.txt
-rw-r--r--. 1 bradschneider  828 Nov 29 10:45 art27.png
-rw-r--r--. 1 bradschneider  116 Nov 29 10:45 art27.txt
-rw-r--r--. 1 bradschneider  811 Nov 29 10:45 art28.png
-rw-r--r--. 1 bradschneider  116 Nov 29 10:45 art28.txt
-rw-r--r--. 1 bradschneider  646 Nov 29 10:45 art29.png
-rw-r--r--. 1 bradschneider   75 Nov 29 10:45 art29.txt
-rw-r--r--. 1 bradschneider  11K Nov 29 10:45 art2.png
-rw-r--r--. 1 bradschneider  708 Nov 29 10:45 art2.txt
-rw-r--r--. 1 bradschneider  530 Nov 29 10:45 art30.png
-rw-r--r--. 1 bradschneider   75 Nov 29 10:45 art30.txt
-rw-r--r--. 1 bradschneider  748 Nov 29 10:45 art31.png
-rw-r--r--. 1 bradschneider  113 Nov 29 10:45 art31.txt
-rw-r--r--. 1 bradschneider 4.4K Nov 29 10:45 art32.png
-rw-r--r--. 1 bradschneider  226 Nov 29 10:45 art32.txt
-rw-r--r--. 1 bradschneider 1.1K Nov 29 10:45 art3_5.png
-rw-r--r--. 1 bradschneider  133 Nov 29 10:45 art3_5.txt
-rw-r--r--. 1 bradschneider 1.1K Nov 29 10:45 art3.png
-rw-r--r--. 1 bradschneider  130 Nov 29 10:45 art3.txt
-rw-r--r--. 1 bradschneider  357 Nov 29 10:45 art4.png
-rw-r--r--. 1 bradschneider   11 Nov 29 10:45 art4.txt
-rw-r--r--. 1 bradschneider 3.1K Nov 29 10:45 art5.png
```

```
-rw-r--r--. 1 bradschneider  154 Nov 29 10:45 art5.txt
-rw-r--r--. 1 bradschneider  104 Nov 29 10:45 art6.txt
-rw-r--r--. 1 bradschneider  101 Nov 29 10:45 art7.txt
-rw-r--r--. 1 bradschneider 1.2K Nov 29 10:45 art8.png
-rw-r--r--. 1 bradschneider   86 Nov 29 10:45 art8.txt
-rw-r--r--. 1 bradschneider  666 Nov 29 10:45 art_text.txt
-rw-r--r--. 1 bradschneider   38 Nov 29 10:45 bug10.txt
-rw-r--r--. 1 bradschneider    7 Nov 29 10:45 bug11.txt
-rw-r--r--. 1 bradschneider   52 Nov 29 10:45 bug12.txt
-rw-r--r--. 1 bradschneider    5 Nov 29 10:45 bug13.txt
-rw-r--r--. 1 bradschneider  138 Nov 29 10:45 bug14.txt
-rw-r--r--. 1 bradschneider   54 Nov 29 10:45 bug15.txt
-rw-r--r--. 1 bradschneider   38 Nov 29 10:45 bug1.txt
-rw-r--r--. 1 bradschneider  134 Nov 29 10:45 bug2.txt
-rw-r--r--. 1 bradschneider  214 Nov 29 10:45 bug3.txt
-rw-r--r--. 1 bradschneider 1.8K Nov 29 10:45 bug4.png
-rw-r--r--. 1 bradschneider   64 Nov 29 10:45 bug4.txt
-rw-r--r--. 1 bradschneider  318 Nov 29 10:45 bug5.txt
-rw-r--r--. 1 bradschneider   46 Nov 29 10:45 bug6.txt
-rw-r--r--. 1 bradschneider 3.0K Nov 29 10:45 bug7.png
-rw-r--r--. 1 bradschneider  204 Nov 29 10:45 bug7.txt
-rw-r--r--. 1 bradschneider  228 Nov 29 10:45 bug8.txt
-rw-r--r--. 1 bradschneider 1.8K Nov 29 10:45 bug9_5.png
-rw-r--r--. 1 bradschneider  108 Nov 29 10:45 bug9_5.txt
-rw-r--r--. 1 bradschneider 2.0K Nov 29 10:45 bug9.png
-rw-r--r--. 1 bradschneider  168 Nov 29 10:45 bug9.txt
-rw-r--r--. 1 bradschneider 3.8K Nov 29 10:45 color_codes.png
-rw-r--r--. 1 bradschneider  222 Nov 29 10:45 color_codes.txt
-rw-r--r--. 1 bradschneider  866 Nov 29 10:45 corner_case01.png
-rw-r--r--. 1 bradschneider  128 Nov 29 10:45 corner_case01.txt
-rw-r--r--. 1 bradschneider  914 Nov 29 10:45 corner_case02.png
-rw-r--r--. 1 bradschneider  127 Nov 29 10:45 corner_case02.txt
-rw-r--r--. 1 bradschneider  512 Nov 29 10:45
dak_orgstruktur_vs_be.ditaa.OutOfMemoryError.2.txt
-rw-r--r--. 1 bradschneider  512 Nov 29 10:45
dak_orgstruktur_vs_be.ditaa.OutOfMemoryError.3.txt
-rw-r--r--. 1 bradschneider  512 Nov 29 10:45
dak_orgstruktur_vs_be.ditaa.OutOfMemoryError.4.txt
-rw-r--r--. 1 bradschneider  140 Nov 29 10:45
dak_orgstruktur_vs_be.ditaa.OutOfMemoryError.edit.txt
```

```
-rw-r--r--. 1 bradschneider  512 Nov 29 10:45
dak_orgstruktur_vs_be.ditaa.OutOfMemoryError.txt
-rw-r--r--. 1 bradschneider  512 Nov 29 10:45 dak_orgstruktur_vs_be.ditaa.txt
-rw-r--r--. 1 bradschneider  610 Nov 29 10:45 ditaa_bug2.txt
-rw-r--r--. 1 bradschneider 2.3K Nov 29 10:45 ditaa_bug.txt
-rw-r--r--. 1 bradschneider   65 Nov 29 10:45 garbage.txt
-rw-r--r--. 1 bradschneider  359 Nov 29 10:45 logo.txt
-rw-r--r--. 1 bradschneider  183 Nov 29 10:45 simple_S01.txt
-rw-r--r--. 1 bradschneider   40 Nov 29 10:45 simple_square01.txt
-rw-r--r--. 1 bradschneider  133 Nov 29 10:45 simple_U01.txt
```

Appendix B: Team Member Journals

Daniel Longendelpher

2020-09-02

- 12:00-12:30
  - Created Team Discord Channel

2020-09-13

- 13:00-14:30
  - Team meeting on Discord
  - Took action items to ask requirements questions on discussion forums and to look over DITAA code

2020-09-14

- 10:00-13:00
  - Installed Intellij IDEA
  - Downloaded and compiled DITAA source code

2020-09-17

- 19:00-20:00
  - Reviewing requirements document
- 21:30
  - Submitted initial requirements doc draft

Christopher Menart

2020-08-30

- Installed Intellij IDEA
    - Initial impressions are positive; this IDE has a similar feel to PyCharm and does not obstruct editing
        - Later realize that PyCharm and IDEA are in fact closely related

2020-09-12

- Downloaded DITAA source code
- Installed and ran Doxygen on DITAA source code

2020-09-13

- Compiled DITAA source code
    - Need to link com.sun.javadoc in order to compile successfully
- Held team meeting on Discord--see meeting minutes

2020-09-15

- Initial read of DITAA source code
    - Code is already quite clean by real-world standards. Code is sufficiently commented, classes are generally clear.
    - No obvious cases of egregious design such as lots of global variables
    - Possibly one demi-god class? ‘Diagram’ is rather long for one class, mainly constructor.
        - Possible target for refactoring

2020-09-17

- Reviewed initial functional requirements from Brad, added 1 new Req
- Added non-functional requirements (as dictated by project givens)
- Add initial discussion of possible maintenance improvements

2020-9-21

- Added initial proposed new functional requirements

10-2-20

- Discussing possibility of (context-free?) grammar for specifying inputs?
- Inspecting DITAA test cases for better view of intended program behavior
    - What forms one unbroken line/shape?

10-3-20

- Running all pre-existing DITAA test cases to see program behavior
    - Best to specify in terms of connected components of lines/corners
- Creating additional test cases for behavior with lots of corner characters
    - Behavior with ‘*’ corners not quite consistent with ‘+’ corners.
- Initial draft of specs of core behavior
    - May change
    - Teammates suggest possible 2-dimensional grammar?

10-6-20

- Re-writing core DITAA specs (line/line group specs) to better match behavior
    - Behavior with directly-adjacent corners encourages specifying connected groups of corners instead of connected groups of lines

10-7-20

- Writing specifications for most additional features (styles, decorations, ‘Other’ features, etc.)

10-9-20

- Writing specs for non-functional requirements
- Summarizing journal for technical report

11/17/20

- Beginning Design document.
- Mostly reading code

11/18/20

- Finished reading all DITAA code
    - Notes on each class

11/19/20

- A comprehensive description of Diagram constructor should enable clear understanding of more or less the whole program. Beginning wokr on this

11/20/20

- Continuing to summarize/understand the Diagram constructor

11/21/20

- Continuing to summarize/understand the Diagram constructor

11/22/20

- Summarizing/understanding Diagram constructo
- AND summarizing/understand BitmapRenderer.render()

11/23/20

- Summarizing/understanding BitmapRenderer.render()

11/24/20

- Writing contracts for Diagram and its methods

11/25/20

- Began writing contracts for methods of TextGrid
    - Realized how many methods TextGrid has, forced to abandon this effort
- Wrote contracts for BitmapRenderer and its methods

11/27/20

- Pursuing suspected bug found when describing Diagram constructor
    - It seems like Mixed shapes wouldn't be handled correctly in all cases
    - Writing test cases to handle this
    - End up going through multiple cycles of tests and code changes documented in Testing

11/28/20

- Need to begin testing! Start test document
- Only thing in it so far is documenting bugfix from the 27th
- Implemented new color codes

11/29/20

- Looking into refactors
    - Start planning to decompose TextGrid class, but closer inspection casts doubt on the wisdom of this
    - Start breaking up diagram constructor instead
    - This triggers an (existing?) bug, which we spend the rest of the day fixing

11/30/20

- Writing JUnit tests
- Writing stress tests

- Formatting notes on DITAA classes for inclusion in Implementation document

12/1/20

- Finishing JUnit tests
- Starting Final Report Document
- Various report writing

12/2/20

- Re-running acceptance tests

Brad Schneider

9/10/20

- 17:00-18:00
    - Created journal for tracking project work
    - Set up some infrastructure for project/homework development
        - Create new instance of Linux VM
        - Checked out DITAA code
        - Downloaded Intellij IDEA
- 18:00-18:30
    - Read through DITAA usage docs
    - Began skimming DITAA source code

9/13/20

- 13:00-14:30
    - Held team meeting on Discord - see meeting minutes

9/14/20

- 16:30-17:00
    - Read deeper into the Requirements Specification example in the appendices of the Marsic book

9/16/20

- 19:00-21:00
    - Produced initial draft requirements table based on the DITAA documentation of features
    - Took a second pass over the initial draft to consolidate/re-number based on related requirements (e.g. refactor REQX, REQY, REQZ into REQX, REQX.1, REQX.2)
    - Created draft Use Case diagram

        - Identified very few use cases for DITAA; it is simple software
    - Started draft document for Requirements deliverable
        - Included both artifacts mentioned above

<u>10/2/20</u>

- Began looking at spec document requirements
- Researched ways to describe 2d grammars
    - Sent PDF link to teammates

<u>10/6/20</u>

- More work on interpreting inputs of DITAA into grammars
- Still some internal debate over how specific to get or how far to take this. Developed grammars for lines, arrows, closed shapes. Began to explore grammars around more combinations, e.g. ‘connected arrows’ which begin with a ‘+’ character that intersects another element, but seems that this can go down a rabbit hole and produce a spec that is actually more confusing to read.

<u>10/7/20</u>

- Decided to eliminate the more specific structures from the grammar.
- Deeper review of draft thus far from teammates

<u>10/9/20</u>

- Transformed notes on the input language into grammar rules using LaTeX for formatting.

<u>10/11/20</u>

- Updated spec draft with some narrative and screenshots of grammars for lines, arrows, shapes.
- Researched conformance and acceptance testing. Determined that these are focused on exercising the input grammar of the elements based on the single function of DITAA.

<u>11/13/20</u>

- Began reading through DITAA code to understand the structure and logical flow of the implementation, got far enough to understand major classes

<u>11/19/20</u>

- Reviewed initial work on design draft from teammate.
- Reviewed course lectures/documents for examples of capturing design

<u>11/29/20</u>

- Sketched out design for new feature (GUI) based on the specs from Team 3 (this was the agreed-upon additions on the discussion board)
- Considering how to test, generated some ideas, e.g. hashing two images to compare graphic output from DITAA
- Initial implementation of DITAA GUI code
- Documented structure of implementation in implementation doc
- Identify smoke test steps

<u>11/30/20</u>

- Identified class notes in design doc that were good candidates for moving to implementation.
- Configure/document ant build steps
- Another pass at design doc review for structure/formatting and formalizing language. Consider sections to possibly move between project documents.
- Discussed possibility of using junit to implement automated regression/acceptance/integration test suite
- Implemented test suite on existing (30+) test files; pass/fail determined by whether the graphic output from test matches the 'truth' output in the tests directory
- Ran code coverage reports before/after the addition of all tests our team implemented

<u>12/1/20</u>

- Update design doc to include GUI design, further proofreading/edits
- Update test document (proofreading, formatting, etc.)
- Add test steps for new GUI feature
- Add code coverage reports to testing doc
- Review existing implementation draft against requirements for the deliverable
- Proofreading, editing of implementation doc draft
- Creation of File listing index

<u>12/2/20</u>

- Added journals to design doc

<u>12/3/20</u>

- Added journals to other docs