DITAA Design Document

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
010

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:

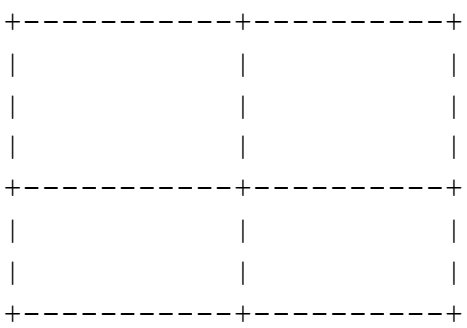
```
_________
/------\
|       |
\------/
```

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.

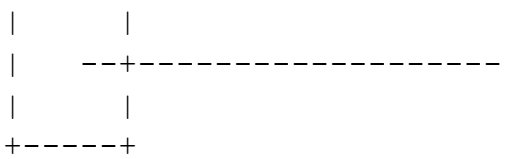
```
<Diagram constructor, part 3>:
    ...
    For each CellSet boundary in boundarySetsStep2:
        startCell := an arbitrary cell from boundary
```

```
        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.

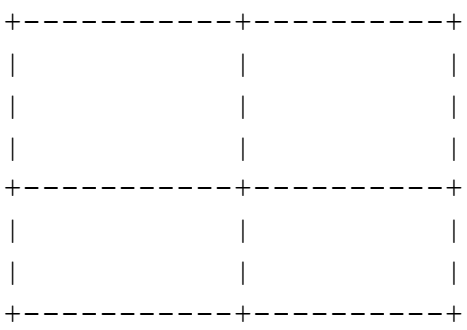
A Mixed shape is one that is neither an open shape, nor a single non-self-intersecting curve defining a closed shape. An example is given in the comments and repeated here:

```
+-----+
```



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.
- extractText(grid)
    - 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.
- makeArrowheads(grid)
    - 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
- makePointMarkers(grid)
    - 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

- renderToImage(diagram, options)
    - Precondition: No arguments are null.
    - Postcondition: Makes a BufferedImage image with the same size as diagram and returns render(diagram, image, options)
- render(diagram, image, options)
    - Precondition: No arguments are null.

    - Postcondition: draws a visual representation of diagram onto image. Overwrites any existing contents in image.
- renderToPNG(diagram, filename, options)
    - If this returns true, it writes renderToImage(diagram, options) to filename as a PNG.
- isColorDark(color)
    - Precondition: Argument is not null
    - Postcondition: Returns true if no color channel in color is greater than 200.

Private Methods

- renderTextLayer(textObjects, width, height)
    - 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).
- renderCustomShape(shape, g2)
    - 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.
- renderCustomSVGShape(shape, g2)
    - 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.
- renderCustomPNGShape(shape, g2)
    - 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:

- dropShadows(diagram, image, performAntialias)

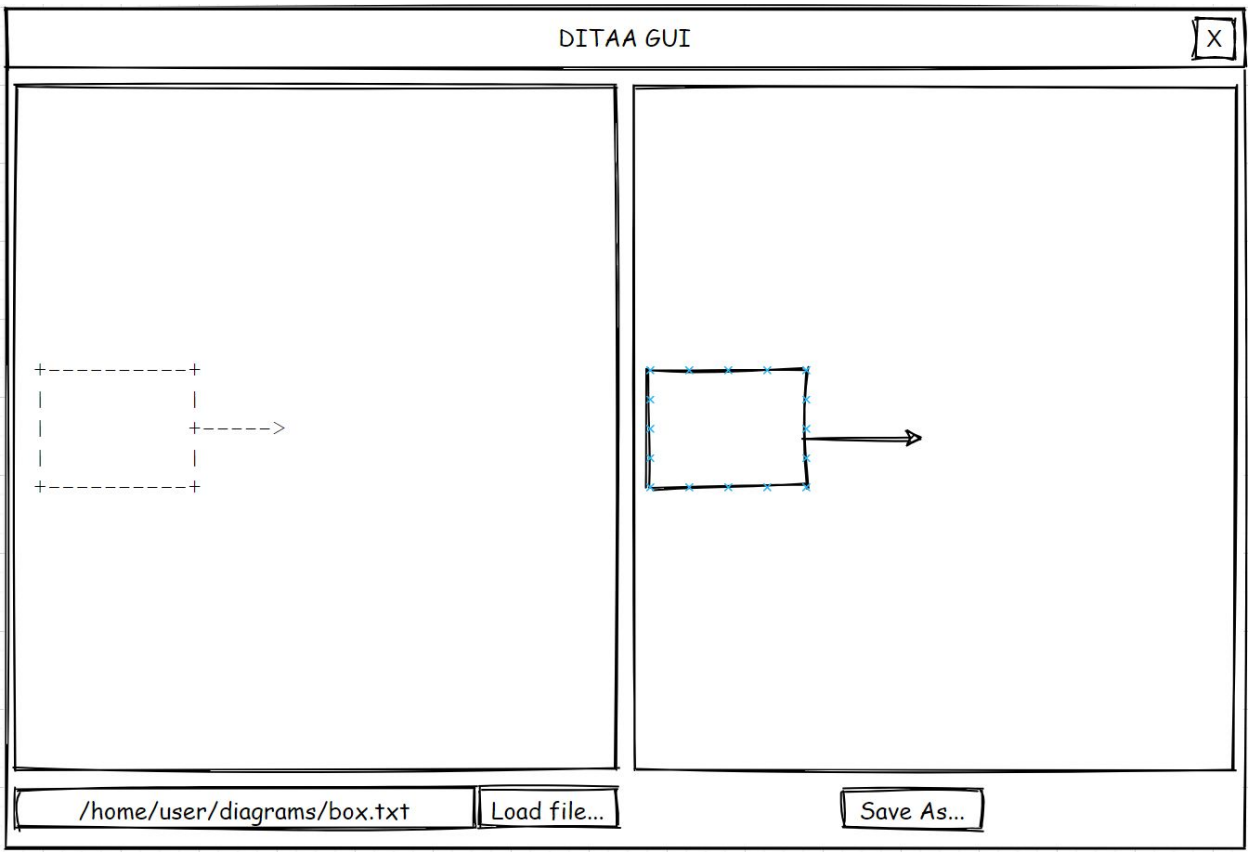
    - Precondition: No arguments are null. Image contains a blank white background and nothing else.
    - Postcondition: A shadow of every shape in diagram.getAllDiagramShapes() for which shape.dropsShadow() is rendered onto a copy of image, which is returned as the result. The shadows are blurry dark-grey 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.

Team Member Journals

Christopher Menart

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 work 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 constructor
- 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

Brad Schneider

11/13/20

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

11/19/20

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

11/29/20

- Sketched out design for new feature (GUI) based on the specs from Team 3 (this was the agreed-upon additions on the discussion board)

11/30/20

- Another pass at design doc review for structure/formatting and formalizing language. Consider sections to possibly move between project documents.

12/1/20

- Update doc to include GUI design, further proofreading/edits

12/2/20

- Added journals to design doc