Laser-Cut Box Designer Application

Team Members:
Audrey Horne
Nolan Donaldson
Scott Oelkers
Michael Berg

Client:
Dr. Owen Hildreth

Date:
December 6, 2020

Introduction

Dr. Owen Hildreth is an assistant professor and researcher at the Department of Mechanical Engineering in the Colorado School of Mines who works with small-scale additive manufacturing. He has a Glowforge laser cutter in his lab (model: Basic), which he and his students commonly use to create and engrave all manner of objects using PDF files as inputs. One particular project that is seen frequently by the laser cutter is designing customized boxes, primarily used for storing instrumentation and delicate lab equipment.

The necessity of a custom box designer stems from many factors; primarily, the lack of good existing software for this specific application. Online or open-source apps often have limited featuresets or design capabilities, lacking features such as saving/loading. On the flipside, fully-featured parametric CAD software such as SolidWorks or AutoCAD are not specifically designed for creating laser-cut parts. As a result, they require a mildly experienced user who correctly accounts for laser-specific design constraints when making boxes, which gets complicated. For example, a CAD user would be forced to model every feature of the box, such as tabs or slots, to mesh with each other properly, and they would need to manually account for design quirks such as the kerf of the laser they will cut the box with. Another factor necessitating the existence of the box designer is our client's desire to have the app developed for iOS and MacOS for use with MacBooks and iPads.

This project was started in the summer of 2020 by a field session group of the time. Our goal was to refactor their code, extensively document the project, and add essential features such as one-button PDF file exporting or slotted joins between box walls. Debugging issues that crippled the user experience of the app, such as the numerous issues plaguing the camera, were also essential. As such, the hope for this project was to bring the app closer to its intended goal: a fully-featured, open-source, freely available, simple, easy to use, and powerful app on the MacOS/iOS platform.

Requirements

We were handed an initial version of the Box Designer application that had been developed by a previous Field Session team in the summer of 2020. This version included a basic box design and the accompanying GUI, basic user controls to modify box parameters, and a simple method to write the design to a JSON or PDF file, which was capable of being read by the Glowforge laser cutter. Our client, Dr. Owen Hildreth, initially gave our team a list of requirements for our upcoming overhaul and expansion of the Box Designer application; for each requirement listed, there is a note that describes either an issue with the initial version's implementation or the lack of implementation. Due to the three priority levels that our client outlined ("must haves", "nice to haves", and "if you have time"), both the functional and nonfunctional requirements (below) are divided into three categories according to priority. Red priority goals were an absolute requirement of this project; they had to be done for the project to be designated as complete. Yellow priority goals were stretch goals that the client wanted the team to complete as soon as the red priority goals were finished. Finally, green priority goals were of the lowest necessity; the client only expected these if all other goals were adequately completed. Additionally, our client slightly adjusted our requirements throughout the semester -- the changes are noted if applicable.

Functional Requirements:

- Red priority goals:
  - Keep the current camera angle, position, and zoom after changing one of the box's properties
    - This was a bug present in the initial version of the box designer
  - Replace the PDF save features with Swift's built-in archiving capabilities
    - The initial version did not use Swift's capabilities (see "Technical Design" section for more details on this requirement)
  - Add button and associated code for exporting to PDF in GUI
    - There are Yellow Priority goals associated with this exporting button.
    - The initial version only had save options in the taskbar, not embedded in the GUI
  - Add the option for measurement units (inches or millimeters) in the GUI and display the chosen unit
    - The initial version only had inches as an option, and the labels did not display the unit.
- Yellow priority goals:
  - The 'Export to PDF' button pop-up should also include an option for inputting physical dimensions of the print area and adjusting the layout of box pieces accordingly (ensuring individual components are not cut off). Additionally, there should be an option to choose only one wall drawn on each PDF page.

        - The initial version did not have any of these options.
    - ➢ Add the capability to create additional cutouts for holes in varying shapes (circle, square, rectangle, triangle, etc.)
        - Note: Our client limited the shapes for cutouts to circle, rounded rectangles, and rectangles later in the semester. However, our cutout implementation allows the code to be easily extensible for adding additional shapes.
        - The initial version did not have this functionality.
    - ➢ Add capability to select and drag different box components with common snap points (center, midpoint, end, etc.)
        - The initial version did not have this functionality.
        - While we were able to implement wall selection and snap points, we did not have time to allow walls to move independently of each other, which was a prerequisite to "dragging" a wall. Thus, we did not fully complete this stretch goal (see "Results" section for more details).
    - ➢ Add capability to highlight a selected box component
        - The initial version did not have this functionality.
    - ➢ Include the capability to delete and add box components
        - The initial version did not have this functionality.
    - ➢ Add option for different lid joinings (tabbed, short slide, long slide, handle)
        - Note: Our client later limited this list to adding a handle to a wall.
            - The initial version did not have this option.
    - ➢ Add option for different side joinings (slotted, tabbed)
        - The initial version did not have the "slotted" join option.
    - ➢ Add user control of the location of internal separators, as well as adding multiple separators of different sizes
        - In the initial version, the user could add a maximum of two internal separators, and they could not choose the plane or the location of the added walls.
    - ➢ Changing the application's JSON saving and loading capabilities to Swift's native Encodable and Decodable protocols
        - The initial version could save as a JSON, but it did not utilize Swift's protocols. Additionally, the initial version could not load a box model back into the application.
- ❖ Green priority goals:
    - ➢ Add option to create text and specify placement on/in the box
        - The initial version did not have this functionality, and we were not able to complete this stretch goal (see "Results" section for more details).
    - ➢ Add optional settings for kerf, margin, stroke, and padding, taking into account the width and taper of the laser

        - The initial version did not have this functionality.

Nonfunctional Requirements:

- Red priority goals:
    - Documenting the code written by the Summer 2020 Field Session
        - The initial version had minimal documentation.
    - Professionally writing an installation manual and user guide
        - Note: This was originally a requirement from our advisor, not our client.
        - The initial version did not include these documents.
    - Ensuring GUI colors are sympathetic to color-blind users
        - The initial version did not address this.
    - Ensuring GUI is sympathetic to users with low vision
        - The initial version did not address this.
- Yellow priority goals:
    - Refactoring the code written by the Summer 2020 Field Session team
- Green priority goals:
    - None

System Architecture

As stated in the introduction, this semester's project was a continuation of CSCI370 Field Session Summer 2020. Because of this, we had an existing architecture to work on. Figure A.1, found in Appendix A, depicts the full UML diagram of the previous team's finalized product. When we analyzed the structure, we found a few major restructure and refactor tasks. Firstly, in the GitHub repository, there were many old, unused classes that were not deleted. Our first step to address the existing architecture was to thoroughly analyze which classes were unnecessary to keep; our team wanted to ensure there would be no confusion for next semester's (Spring 2021) field session group that Dr. Hildreth plans on enlisting. Figure 1 showcases all of the classes in the repository that we found to be obsolete. Once identified, they were removed from the repository and the architecture was restructured.

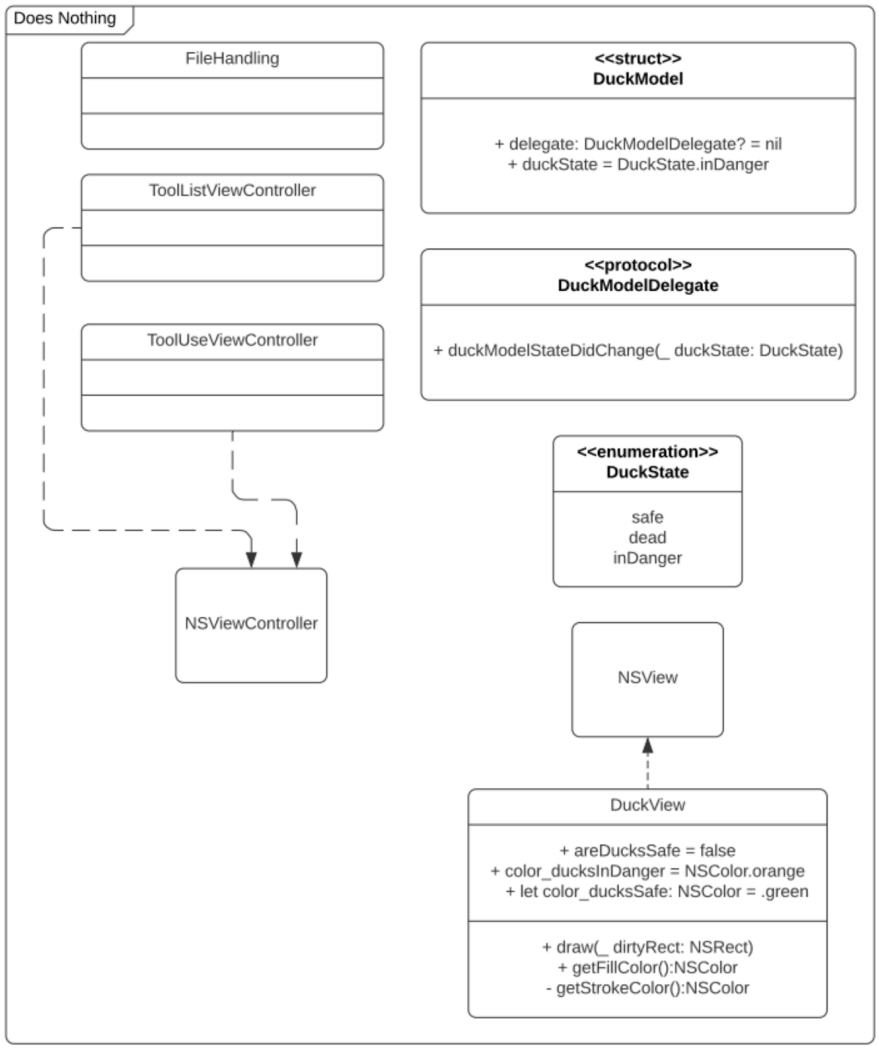

Figure 1: This is a subsection of the previous team's UML which shows a collection of classes that our team analyzed and found to be unused. The full UML is in Appendix A (Figure A.1).

Additionally, our team addressed refactoring the existing architecture to better suit the new requirements that needed to add functionality. For instance, one of our main requirements was to redesign how the application exported the box model template to PDF -- the client wanted Swift's native archiving capabilities to be used. Because of this, the collection of classes that addressed File Handling Control were completely refactored and, at times, deleted (Figure 2). Additionally, the `PathGeneration` class -- which is essentially how the walls of the box are constructed -- was bloated with functions that could be condensed once we understood the class' purpose. Comparing Figures A.1 (original UML) and A.2 (our finalized UML) in Appendix A will show the intense refactoring we completed on the `PathGeneration` class.

Along with refactoring, we added numerous classes that addressed the new requirements. We ensured that these new classes complied with the Single Responsibility Principle and were designed in such a way that next semester's field session team could easily extend their functionality, based on projected requirements from our client. Figure A.2, found in Appendix A, depicts the full UML that is the result of our described efforts.

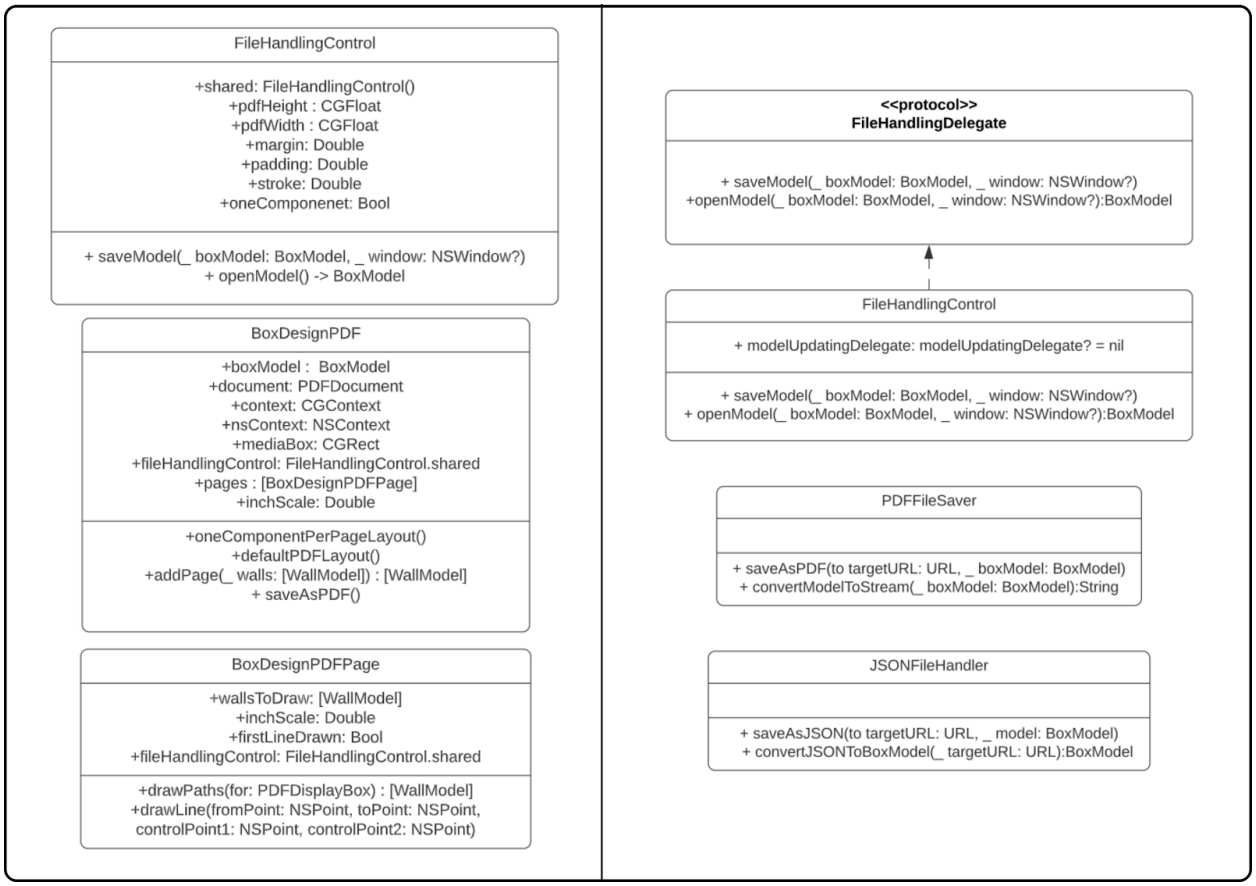

Figure 2: The left side of this figure is our team's finalized collection of classes that address File Handling Control. The right side of this figure is the original collection of classes, which we refactored or deleted to comply with a top-priority requirement from our client. This refactor is discussed in detail in our Technical Design section.

In addition to redesigning and adding to the original code, we altered the GUI of the original application. Figure 3 depicts the original application interface, whilst Figure 4 depicts the final GUI for the application. Because our client wants to extend this application's compatibility from strictly macOS to both macOS and iOS in the future, we designed the GUI with the thought that Apple phones and tablets will not have easy access to a menu taskbar, as they would on Apple computers. Therefore, we put as much functionality in the GUI panels below the box view as we could. This way, the future team that adds iOS-compatibility to the application can use most of the existing GUI.

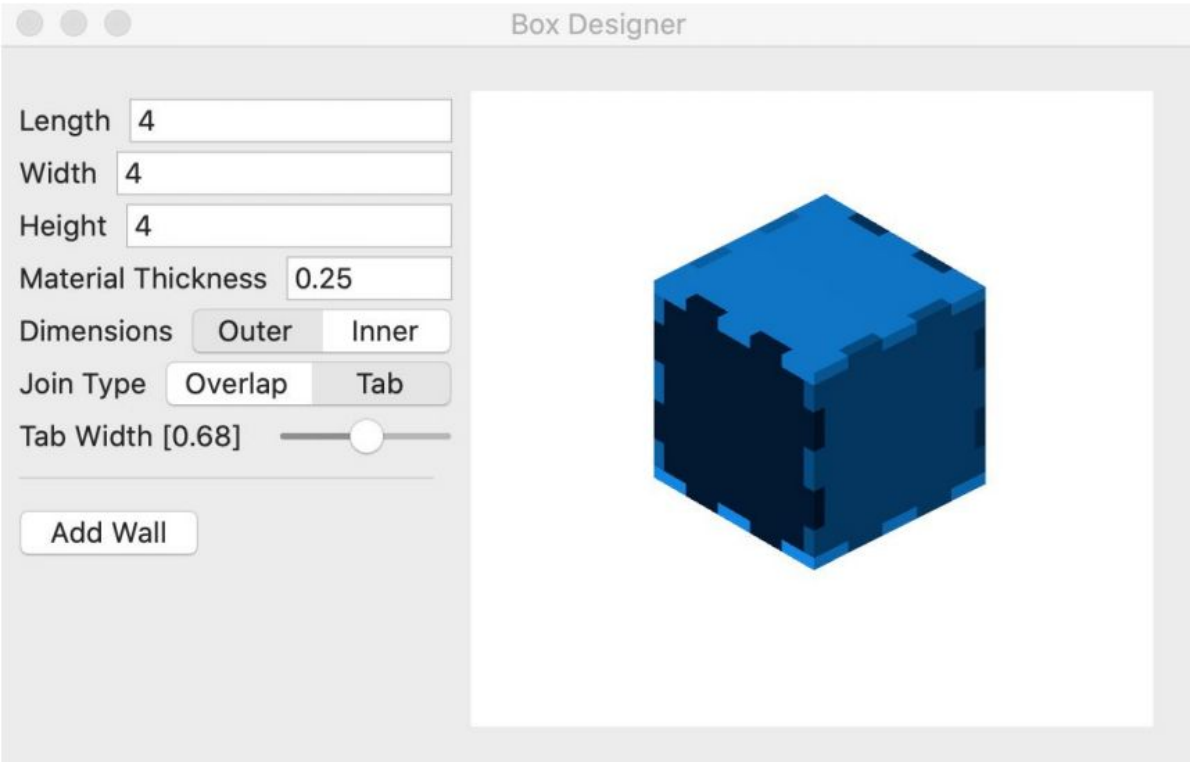

Figure 3: This is the finalized GUI from Field Session Summer 2020. This image was extracted from their final report, as we did not procure an image of the application when we first began to work.

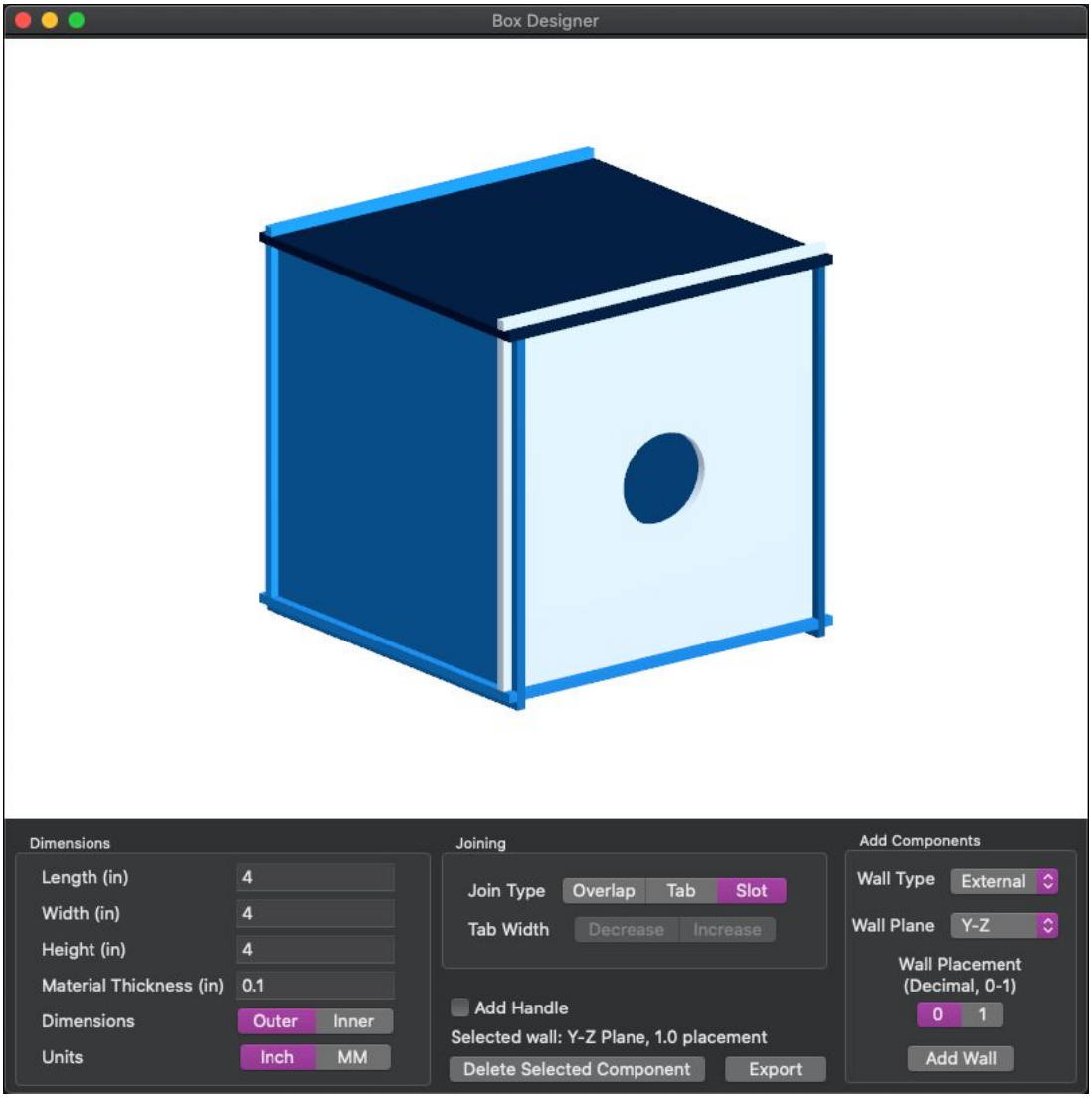

Figure 4: This is our finalized GUI. It showcases some of the new functionality that we added -- slot joins, wall selection, unit display and selection, and shape cutouts.

Technical Design

While we refactored, redesigned, and newly designed a lot of functionality in order to comply with our requirements, two designs were the most exciting to us. These included 1) the refactored design for how the box model was exported to PDF and JSON formats, as well as the new design for how a saved box model was loaded back into the application, and 2) the new design for shape cutout capabilities (see Figure 4 above for visualization on a circle cutout from a box model wall). We chose these topics because we think they are the epitome of extensibility and efficiency -- throughout our process, we continued to remind ourselves that developers would be working on this project next semester. Thus, we wanted to ensure that they could easily extend these designs.

I. Using Swift's Native Capabilities for Exporting and Loading

Our client wanted to completely overhaul the original design for exporting a box model to PDF and JSON, and for loading a saved box model template from a JSON file into the application. For the PDF exportation, the original code converted the wall paths to one long string and wrote that string to a .pdf file. This design was inflexible, and sometimes resulted in walls of the model getting cut off on the PDF; there was no way to add another page to the PDF, because it technically was not a PDF document. Thus, if the box model was too large for a single PDF page, the code could not mitigate walls getting cut off. Therefore, we redesigned the method of PDF exportation by incorporating custom classes that inherited from Swift's native objects and libraries (Figure 5).

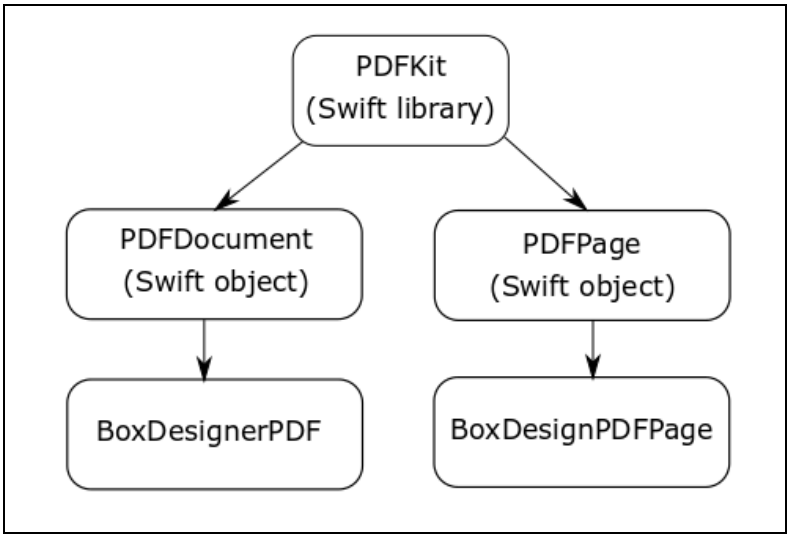

Figure 5: This image depicts the inheritance of our custom classes that guided our design for exporting box model's to the PDF.

This design allows a lot more flexibility and user choice; because Swift understands it is a PDF document, the user can now choose:

- PDF width
- PDF height
- PDF margin (how far from the edges of the document?)
- PDF padding (distance between individual wall drawings)
- PDF stroke (how thick should the line be?)
- One component per page (there is either the default layout, which ensures no walls are cutoff, but tries to put as many walls as possible on a page and only adds a page out of necessity, or the user can choose to put each wall component on separate pages of the PDF document)

One of the "happy byproducts" of this design is that the option for PDF stroke can address some small issues with the laser that had previously happened. The laser that cuts these boxes out has both a width (albeit small) and a taper (the laser loses small amounts of energy as it cuts further from the source, which leads to a slightly sloped wall edge). The ability to change the stroke of the walls' lines could help account for the width of the laser, ensuring the box walls fit together snugly when put together. Figure 6 is a sample PDF document from this new design.

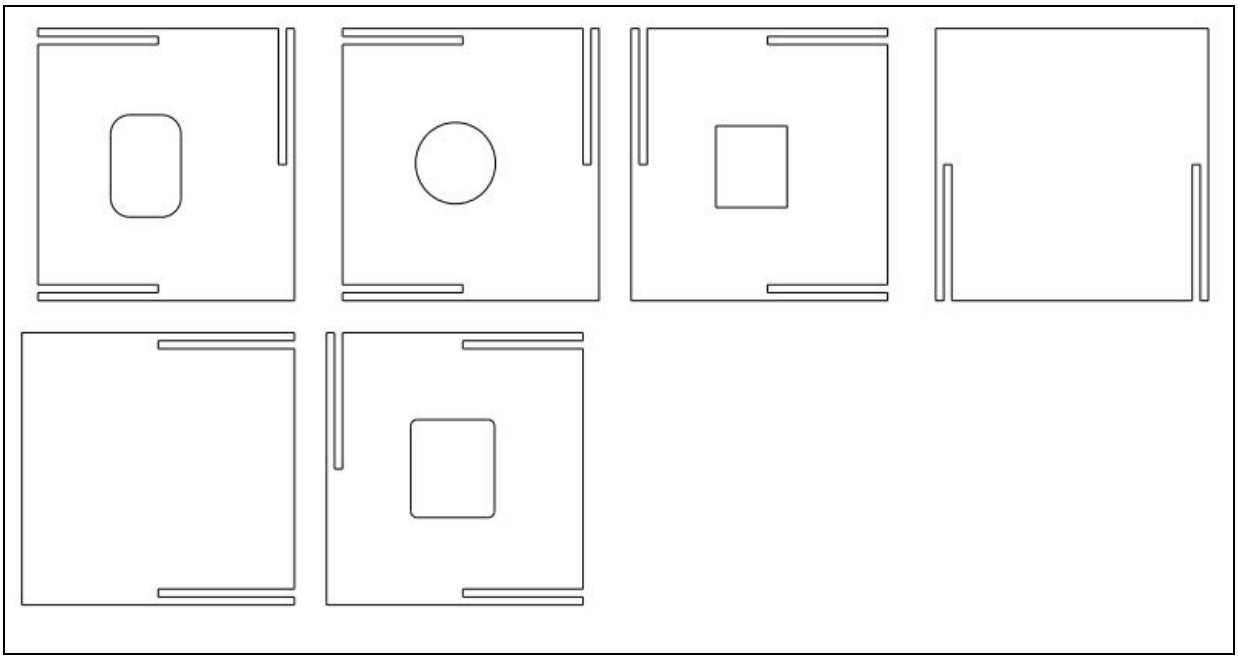

Figure 6: This is a sample PDF export of a slot-join box model with cutouts. For this sample, we adjusted the size of the PDF page so that all walls would fit and the stroke so that the lines were finer. This adjustment would not be possible without the new design. We also updated the drawing to accommodate curved lines, as originally the code could only draw straight lines.

In addition to redesigning PDF exportation, we redesigned saving to a JSON file and loading a saved JSON box model back into the application. Originally, the code manually converted all

necessary box model information into a long string and saved it as a JSON, and while there were functions addressing opening a box model from a JSON, they were skeleton code. To redesign this, we used a built-in Swift protocol called Codable that can encode custom classes into a JSON format, and subsequently decode a JSON file to initialize the saved box model. Figure 7 is an example of an encoded JSON file that can be loaded into the application to revert to the saved box model.


```
{"materialThickness":0.125,"joinType":"slot",
"boxHeight":4,"innerDimensions":false,"numberTabs":3,
"walls":{
"3":{"numberTabs":3,"position":[0,0,0.0625],
"wallType":"smallCorner","innerPlane":"smallCorner",
"width":4,"wallNumber":3,"length":4,"joinType":"slot",
"innerWall":false,"handle":false,"wallShapes":[],
"materialThickness":0.125},
"1":{"numberTabs":3,"position":[0.0625,0,0],
"wallType":"longCorner","innerPlane":"smallCorner",
"width":4,"wallNumber":1,"length":4,"joinType":"slot",
"innerWall":false,"handle":false,
"wallShapes":[],"materialThickness":0.125},
"4":{"numberTabs":3,"position":[0,0,3.9375],
"wallType":"smallCorner","innerPlane":"smallCorner",
"width":4,"wallNumber":4,"length":4,"joinType":"slot",
"innerWall":false,"handle":false,
"wallShapes":[],"materialThickness":0.125},
"2":{"numberTabs":3,"position":[3.9375,0,0],
"wallType":"longCorner","innerPlane":"smallCorner",
"width":4,"wallNumber":2,"length":4,"joinType":"slot",
"innerWall":false,"handle":false,"wallShapes":[],
"materialThickness":0.125},
"0":{"numberTabs":3,"position":[0,0.0625,0],
"wallType":"bottomSide","innerPlane":"smallCorner",
"width":4,"wallNumber":0,"length":4,"joinType":"slot",
"innerWall":false,"handle":false,"wallShapes":[],
"materialThickness":0.125},
"5":{"numberTabs":3,"position":[0,3.9375,0],
"wallType":"topSide","innerPlane":"smallCorner",
"width":4,"wallNumber":5,"length":4,"joinType":"slot",
"innerWall":false,"handle":false,"wallShapes":[],
"materialThickness":0.125}},
"intersectingWalls":[],"boxWidth":4,"wallNumberStatic":6,"boxLength":4}
```


Figure 7: This is a sample JSON file that was encoded in an application session and can be decoded to revert the application to this saved box model's specifications.

In order to use Swift's protocol, we had to enable numerous custom classes to conform to Codable. If a variable type did not conform to Codable in the custom classes, we had to either save the necessary information for that variable to be instantiated after decoding, or we had to extend the type so that it could be encoded. As an example: integer and double types already conformed to Codable, and no alterations were needed for variables of that type. The SCNVector3 type did not conform to Codable, so we had to extend its functionality. Finally, NSBezierPaths did not conform to Codable, and we could not extend its functionality, so we had to save specific data that allowed the wall's paths to be generated after decoding. Overall, this

design was more efficient and simpler than the original design, and its outcome added functionality to the application. Furthermore, it was implemented with extensibility in mind -- if future developers add variables to the `BoxModel` or `WallModel` classes, they simply need to add them to the Codable functions in order to encode and decode the new variables.

II. Cutout Technical Design

Because our client originally wanted more cutout shapes than we implemented, we wanted to ensure that adding new shape types was easy for future developers. In addition, we had to design the cut outs in a way that allowed them to be encoded and decoded via Codable (see above). As previously mentioned, NSBezierPaths (which are essentially the paths that define the walls) cannot conform to Codable. Because of that, the shapes needed to be defined by their attributes that do conform to Codable, and they also needed to be associated with the wall they are cut out from. Figure 8 shows the finalized design for cutouts that addressed these necessities.

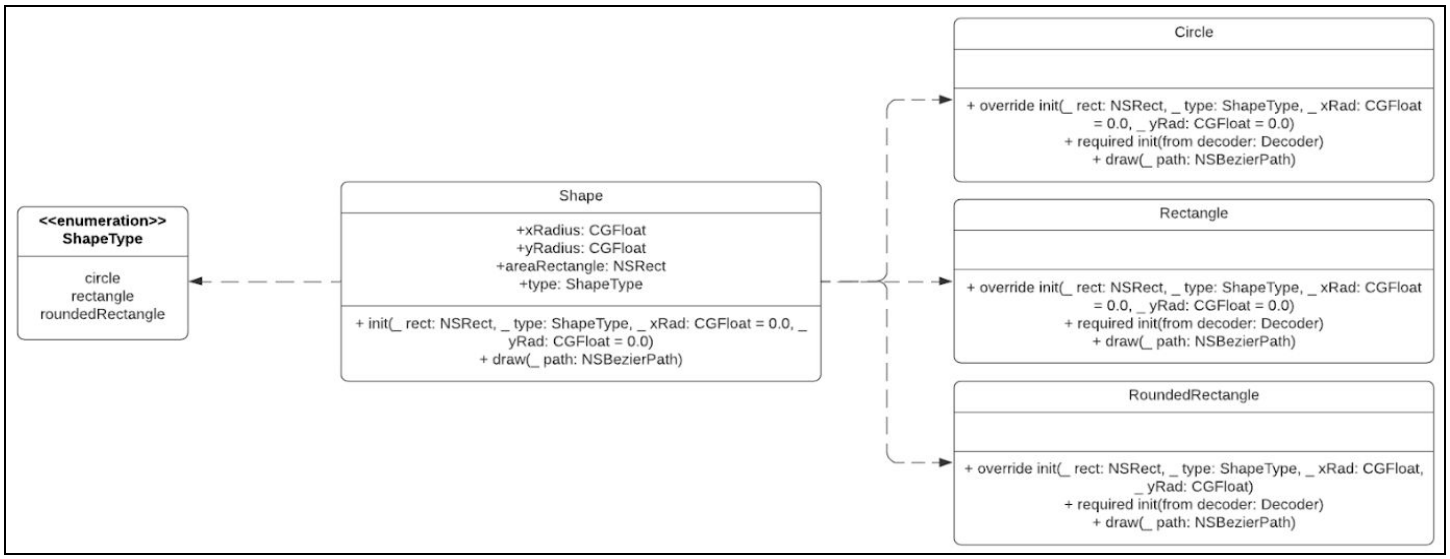

Figure 8: This UML depicts the abstract class "Shape," which any cutout shape must inherit from, and the enumerated "ShapeType," which is a list of allowed shapes.

As Figure 8 shows, the Shape class is essentially an abstract class (we say "essentially" because Swift does not have an "abstract" keyword). The classes that inherit from Shape must implement the draw() function -- all shapes will draw themselves differently, and it is up to them to know how to draw themselves. Additionally, the enumerated ShapeType is simply a list of allowed shapes. This design allows extremely easy extensibility of the application's allowed shape cutouts; as long as the shape knows how to draw itself, future developers can simply add the enumerated shape type, implement the new shape's class that inherits from Shape, and tell it how to draw itself. In addition, all of the Shape class variables conform to Codable. Future developers need not worry about addressing new shapes' saving and loading capabilities. However, we still

need a way to associate the cutouts on a wall with the wall itself. Due to the design of cutouts, this is extremely easy: the instantiated wall simply has an array of Shapes ([Shape]) that can be drawn with it. This means that the wall-shapes data structure will not need to be updated if future developers add possible cut out shape types. We believe that this design is the most efficient and extensible way to add and associate cutouts with their walls.

Quality Assurance

Because our client initially expressed his desire to release this application as open-source software in the future, we knew that Quality Assurance was of the utmost importance. Whenever our team implemented new functionality, we extensively manually tested the application to ensure that previous features were still working and new features were bug-free. If we ran out of time to fix a minor bug (for our purposes, "minor bug" means it won't crash the application), we made sure to note the bug in our GitHub repository's "Issues" tab for future developers. Furthermore, we completed a user guide that is up-to-date with all functionality in the application (Appendix B). If our client releases this application as open-source software, new users will be able to quickly learn the controls and featureset via the user guide.

As we believe quality assurance should also be applicable to future developers of the project, we wanted to note that we thoroughly documented all of the classes in the current system architecture. Additionally, we set up GitHub Pages (https://hildrethresearchgroup.github.io/Box-Designer/index.html) for our repository and a GitHub workflow that updates the documentation any time a developer pushes commits to the remote master branch. The documentation will be current and accessible for next semester's team, and they will be able to update GitHub Pages automatically -- with the stipulation that they document their code using Apple MarkUp syntax.

Unit Testing

Because the main purpose of this application is to be easy to use, we put most of our efforts in user interface testing (see below). As long as the application was visually responding correctly, the only testing outside of user interface testing we felt the need to do was verifying that the dimensions of the box template on the PDF correspond correctly to the user-inputted dimensions in the application. After all, there is no point for the user to adjust the dimensions if the resulting PDF (which the box is laser-cut from) is not accurate. To test this, we printed out the PDF of a box that, according to the application, should be 4 x 4 x 4 inches. After measuring with a ruler, the printed box passed this test.

User Interface Testing

Our group decided to continually test any new functionality that we added in order to 1) make sure that the original functionality continued to work correctly and 2) make sure that new implementations did not have obvious, app-crashing bugs. These are the behaviors and questions that we addressed while conducting our tests:

- Ensuring that changing UI elements does not break expected behavior, such as box dimensions or the information displayed by UI elements
- Box dimensions - does resizing the box cause visual errors or errors with the PDFs created by the program?
- Changing the features of the box under certain conditions:

- Exporting PDFs with extreme box features or dimensions
- Tabs - varying tab count, tab width(s), etc.
- Camera: testing varying camera angles (box rotation), the effect of changing box features and dimensions on the camera, any possible camera clipping through the box, keeping the box in frame, testing for any lighting glitches that make the box seem 2D instead of 3D, etc.
- Identify any edge cases or outliers, test them, and address if the test does not visually pass
- Bad/invalid inputs handled? How does the program react?
- Do all functions work as expected with all combinations of settings?

Integration Testing:

While integration testing will be more prominent for the field session team that expands this application to iOS, we did find a few tests to perform. They are:

- Testing trackpad/touch gestures to ensure iOS compatibility
  - Although we were only working on the macOS application, we wanted to ensure that app users on laptops had intuitive camera controls with their trackpad. We manually tested this, as some of our members had trackpads. Because the trackpad is similar to touch gestures on a smartphone or tablet, we believe the motions for camera control will translate easily to iOS.
- Resizing the window of the app to ensure compatibility on varying screen sizes, resolutions, and aspect ratios
- Importing and exporting files that are supported by devices that will run the program

Results

Incomplete Requirements:
Our group was able to address and adequately complete almost all of the requirements requested by our client -- from all priority levels. The only two requirements we did not complete are:

- Dragging to snap points: while we did address snap points (there is foundation code that will make a snap point appear at edges and midpoints if the user hovers their cursor over the snap point area), we were not able to allow the user to drag a specific wall and snap it into a location. As of now, adding walls is a slightly more in-depth process for the user: they have to specify the location in the GUI panels, instead of being able to just drag to a snap point.
- Adding text to walls: we were not able to address this requirement at all, at least with something concrete. We did a lot of research about the best way to do this, but each avenue we attempted turned out to be a dead end. In the future work, we list a couple of ideas that we had, but did not have time to implement.

Future Work:
Firstly, if we had knowledge of minor (non-app-crashing) bugs in the application that we did not have time to fix, we made sure to post these issues in the GitHub repository "Issues" tab for future developers. Secondly, we have quite a few ideas for future refactors, implementations, and functionality. They are:

- Text: To address adding text to the box, we have two ideas. The first, which we explored the least, is to use CGGlyphs to display the text on the SCNNode of the text-added wall. The NSBezierPath of CGGlyphs can apparently be extracted, which is necessary for drawing on the PDF. Our second idea was to simply implement the viewing and PDF-drawing of the text independently. We were able to view text on walls in the application via SCNText, but there is no way that you can extract an NSBezierPath from an SCNText object. Therefore, if future developers kept track of the "where and what" of strings the user wants, they could simply use the NSString.drawInRect() method when exporting the text to PDF.
- Cutouts: There are a few ideas for future work with the cutouts. While there is currently code that removes a cutout if the user decreases the dimensions such that the cutout will not fit, the cutouts do not translate logically if the user increases the dimensions of the box. This should be addressed after considering how the user would want their cutout placements to change when altering dimensions. Additionally, as of now, the rounded rectangle cutout's "roundness" can only be changed symmetrically -- it would be easy to split this up so that the user could cut out an asymmetrical rounded rectangle.
- Adding walls: The major issue with how the "adding components" is currently designed is that there cannot be four equal compartments; the code deals with intersections of internal walls by chronological order (older walls do not change, while newly added

walls adjust their dimensions according to an imminent intersection). Additionally, all walls are drawn relative to the origin. Due to this combination, there is no way to "continue" a wall after an intersection on the other side of the intersected wall. One idea we had to address this was for users to input a negative number for the placement of an internal wall, which would indicate they want to draw from the walls that are opposite the local coordinate system's origin.

- Apple Help: while there is an accompanying user guide for the application, we attempted to incorporate the user guide into the in-built Apple Help, which would enable the user to use Spotlight to search for their desired topic. This feature could easily be found under the "Help" menu in the taskbar. However, it was absurdly strange to implement this feature, and we decided to focus on more important requirements at the end of the semester.
- Conversion to SwiftUI: Our client has expressed a desire to rewrite the application in SwiftUI to help commercialize the product and improve device intercompatibility. Proper documentation will be essential moving forward.

Learned Lessons:

We think the most important, career-pertinent lesson that we learned for this project was the increased difficulty of working on pre-existing code -- especially pre-existing code that is not documented well. It is easier to address issues if you have worked on the code from the beginning and intricately understand the design. However, all of our members had to get caught up on Swift and XCode before we could adequately understand all of the existing code. This project (even with the initial learning curve) was a great experience though, as most of us will have to be the "future developer" for a project in our career. We also learned that Swift is an extremely capable and fun language with fantastic GUI support, and XCode is a capable IDE as well -- although there are some issues with its Git capabilities. Finally, we have all learned that procrastinating merges with the master branch in Git will lead to even bigger headaches later on.

Appendix A - UMLs

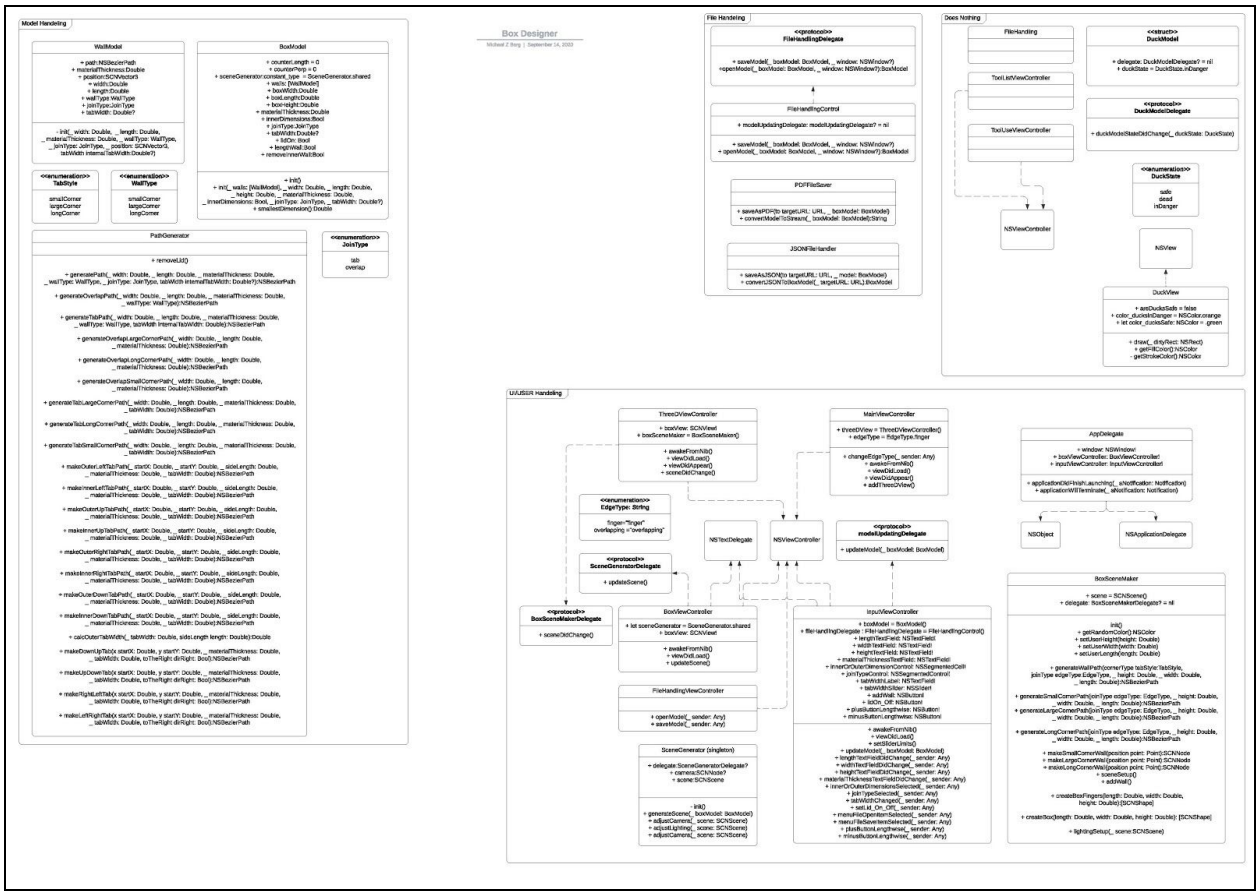

Figure A.1: This UML describes the previous team's finalized product; it was the initial architecture when we started working on the project.

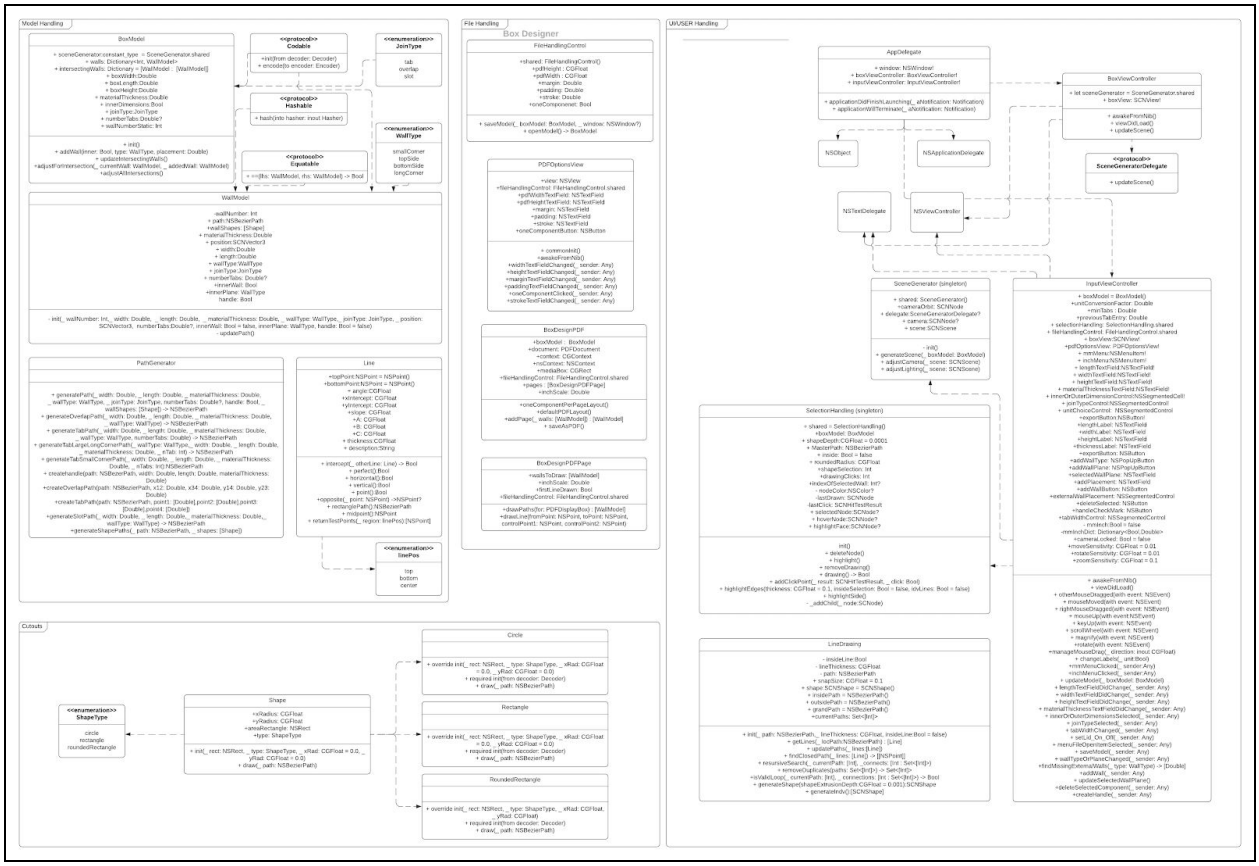

Figure A.2: This UML describes our finalized system architecture that addresses most of the requirements from our client (see "Results" for requirements that were not completed).

Appendix B - User Guide

Box Designer User Guide

This document explains the camera and panel controls for the Box Designer macOS application.

Camera Controls

Mouse Camera Controls

- Zooming: The scroll bar zooms in and out.
- Translation: Right-clck-and-drag moves the box around the view.
- Rotation: The page-back or page-forward buttons on the (typically) left side of the mouse rotate the box.
- Selection: A single left-click on a wall highlights that wall.
- Focus: A double left-click on a wall focuses the camera only on that wall (for drawing cutouts).
- Abort Focus: The escape key gets the user out of “focus mode” and enables the user to highlight walls again.
- Abort Selection: The escape key will also abort selection mode — if a wall is highlighted, the escape key will return the box to an un-highlighted state.

Touchpad Camera Controls

- Zooming: Similar to iPhones, use two fingers and pinch inward to zoom out, or pinch inward to zoom in.
- Translation: Using two fingers, drag the box around the view (no pinching).
- Rotation: Similar to iPhones, using two fingers, move in circular motion to rotate the box. If using a trackpad, two finger click and drag also works.
- Selection: One-finger single click selects a wall to highlight it.
- Focus: One-finger double click focuses the camera on the double-clicked wall (for drawing cutouts and adding text).
- Abort Focus: The escape key gets the user out of “focus mode” and enables the user to highlight walls again.
- Abort Selection: The escape key will also abort selection mode — if a wall is highlighted, the escape key will return the box to an un-highlighted state.

Cutout Controls

- To cut out shapes, the wall must be in focus mode (double-clicking on the desired wall).
- After the wall is in focus mode, click once and move the mouse around to see the projected shape to cut out (this will be viewed as a pink overlayed shape on the wall). The '[' key filters through the allowed shapes (circle, rectangle, and rounded rectangle) backwards, while the ']' filters through the shapes forward. For rounded rectangles, the "rounded radius" can be changed by using the '+' and '-' keys. If the starting click was not where the user wanted the shape, press 'Esc' once to remove it from the view, and click in the new spot to start drawing again. If the pink shape looks like the desired cutout, click once more where the cursor currently is to cut out the shape on the wall. To be able to select walls again, press the 'Esc' key twice. The user will know they are in "Selection" mode again when the "Selected wall: " output reads "None" in the panel area. Final note: as of now, if the user wants to draw two shapes on the same wall, they must get out of Drawing and Focus mode (double-press 'Esc' key), and then re-focus on that wall. This is a bug that we have yet to find the issue for. Technically, the user can still cut-out a second shape in the same "Focus" session, but the second cut-out does not highlight where/what the shape looks like -- it just cuts it out after two clicks.

Application Menu:

File

- Open: This options allows users to load a previously-saved box template into the application. As of now, this is only allowed for JSON files.
- Save: This options allows users to save their current box template as a PDF or JSON file, at the path destination of their choosing. The user can also use the “Export” button in the GUI.

- Close: This option closes the Box Designer application.

Format

- Units: This option allows users to change the displayed units from the menu; it is also an option in the GUI.
- Color Change: This option is not enabled right now. The current shading and colors comply with WCAG.

View

- Enter Full Screen: This option allows user to expand the application to fit the whole screen. As of now, the panels at the bottom do not adjust. Press escape to exit full screen mode.
- Low Vision: This option is not enabled right now.

Window:

- Minimize: This option allows users to minimize the application and store it on the Dock.
- Zoom: This option allows users to maximize the area of the application in the screen, without going to full screen.
- Tile Window to Left of Screen: This option allows users to split their screen into two sections, with the Box Designer application on the left section.
- Tile Window to Right of Screen: his option allows users to split their screen into two sections, with the Box Designer application on the right section.
- Bring All to Front: This option allows users to bring any open sessions to the front of the screen — in front of any other applications that are open.

Help

- Hopefully, this taskbar item will hold the user guide and enable Spotlight Searching for specific topics in the user guide.

Panel Menu

Dimensions

- Length: This textfield takes input from the user, specifying their desired length of the box. The length is the dimension along the local coordinate system's z-axis.
- Width: This textfield takes input from the user, specifying their desired width of the box. The width is the dimension along the local coordinate system's x-axis.
- Height: This textfield takes input from the user, specifying their desired height of the box. The height is the dimension along the local coordinate system's y-axis.
- Material Thickness: This textfield takes input from the user, specifying their desired material thickness of the box. It is important this variable matches the material thickness that will be cut, otherwise the box will not fit together correctly.
- Dimensions: This option allows user to choose whether the length, width, and height dimensions they input refer to the space between the inner sides of the walls, or the total dimensions of the box (including the walls). If the inner option is selected, the true dimensions of the box will be the dimensions that are in the text fields, along with 2*(Material Thickness).
- Units: This option allows the user to choose inches or millimeters as the displayed unit in the application. This can also be changed in Menu Taskbar -> Format -> Units.

Joining

- Join Type: This option allows users to choose whether the walls are joined via overlap, tab, or slot style. Overlapping and tab joins require adhesive for the box to stay together, while the slot join has an interlocking nature that does not require adhesive.
- Number of Tabs: If the user selects the Tab join style, they can also choose the number of tabs for each wall. Due to how the tabs fit together, the "Number of Tabs" entry indicates the maximum number of tabs a wall can have, not the number of tabs every wall has.

Add Components

- Wall Type: This option allows users to choose whether they want to add an internal separator or an external wall. If neither "External" nor "Internal" is selected, then the "Add Wall" button is disabled.
- Wall Plane: This option allows users to choose the orientation of the wall they want to add — the plane that the wall will be on. However, due to the lack of visual coordinate system, this field changes as users select on walls. It will reflect the plane that the currently-selected wall is on, so that users can understand the orientation better. If a user does not wish to try and select the correct plane, they can choose a wall that is on the plane they want to add on, and the software will automatically adjust to adding a wall on that plane.
- Wall Placement:
    - Internal Wall Type Selected: This option allows users to choose where they add the wall in the box model. For internal walls, the user can input a decimal that is between 0 and 1; the decimal indicates where the internal wall should be placed between the two external walls on that plane, where the external walls are at exactly 0 and 1 placements. For instance, if a user wants an internal separator that's halfway between two external walls, they would input 0.5. If they want an internal separator that is a quarter of the way between two walls, they would input 0.25. Any decimal between 0 and 1 will work, but if the material thickness is large, inputting decimals close to 0 and 1 (like 0.01 or .99) would cause the walls to be "too close for comfort."
    - External Wall Type Selected: The wall placement for external walls is constrained to either 0 or 1, as these indicate the edges of the boundaries of the box (normalized). The 0 selection indicates the wall at the origin, whilst the 1 selection indicates the wall that is not at the origin. Due to the slightly confusing nature of this, the selection for the missing external wall for the specified plane is automatically chosen by the software. If both external walls for that plane are missing, then the selection defaults to 0. This is for user-friendliness of adding external walls.
- Add Wall: This button adds the wall specified by the other options into the box model. This button is disabled if neither "External" nor "Internal" are selected in the Wall Type drop-down menu.

Extraneous Buttons and Outputs

- Add Handle: This option allows a users to add a handle to any selected wall. As of now, it is a fixed-size, double-rectangle handle that is separated by a fixed distance. It will allow a wall to be grabbed in real life, after the box has been cut.
- Selected Wall Text Output: For user-friendliness, this text outputs 1) the plane of the selected wall and 2) the placement of the selected wall. If no wall is selected, it defaults to "None selected"
- Delete Selected Component: This button allows users to delete the wall that is currently selected. It does nothing if no wall is selected. Deleting the selected wall can also be accomplished by the "Back-space" key.
- Export: This button allows users to save their current box model as a PDF or JSON. The user can also save via the menu taskbar at File -> Save.