Outliner Project

Dr. Owen Hildreth

Ally Smith
Chayada Somrit
Emily Hepperlen
Peter Hall

Colorado School of Mines

CSCI 370: Advanced Software Engineering

Summer 2021

Table of Contents

Introduction 1

Requirements 2

Non-Functional Requirements

Functional Requirements

System Architecture 3

Technical Design 5

Quality Assurance Plan 7

Results 9

Lessons Learned 10

Future Work 11

Appendix 12

Introduction

Dr. Owen Hildreth is a materials science professor at the Colorado School of Mines who uses macOS for all of his lab research and equipment. For many years he has used a program called Trees 2, a tree-based note-taking application, similar to an outline in Microsoft Word that uses a horizontal structure with subtopics displayed to the side of the overarching parent rather than points being beneath a topic. This makes it easy to quickly read notes and keep track of ideas. This program was held in high regard, but the developer (topoftrees.jp) abandoned their software and it became deprecated as APIs updated and Apple stopped supporting old functionality.

He has asked us to recreate the functionality of the Trees 2 program using Apple's most recent frameworks, particularly SwiftUI and Combine. These allow for us to design our program declaratively as opposed to the more standard imperative or event-driven programming paradigms.

The vision for our version of Trees 2, which is called Grendel, is to recreate its core functionality while modernizing it to use more recent frameworks as well as fixing the features that were broken. By using SwiftUI to recreate this program, the core functionality can be ported to iOS and iPadOS with only minor changes and adaptations.

Requirements

Because our team was tasked with recreating the behavior of existing software, our functional requirements focus heavily on the expected/desired functionality. Our non-functional requirements are the overarching design/architecture which guided our decisions when implementing the features described below.

Non-Functional Requirements

- Use SwiftUI and Combine wherever possible, avoid AppKit and UIKit
- Different elements must have good contrast and be visually distinct using color
    - Light mode specifications
        - Text should be dark
        - Connecting lines in the tree should be a lighter color than the default text
        - Handles to resize tree layers should be lighter, but darker than in Trees 2
- Uses object-oriented programming (OOP) to organize inputted data

Functional Requirements

- Create new documents from the Apple menu bar
- Multiple documents may be open simultaneously
- Maintain Trees 2 keyboard shortcuts as best as possible
    - Press Enter to insert new lines
    - Press Tab and move the current item horizontally in a lower level under the top item
    - Press shift+Tab to move the current item vertically under its current parent item
- Ribbon along top with buttons for adding children, adding items, etc.
- Copy and paste nodes and their children
- Has icons (bullet points) designating if a node has children or not
    - Circle: leaf
    - Arrow: has children
    - Minus: toggle minimize
- Collapse and expand subtrees (toggle family)
    - Can minimize and maximize children of nodes
    - Can maximize a node and all of its children
- Dotted gray lines used to show the connections between parent and child nodes
- Able to change the text color, bold, italic, highlight, etc. of an individual node
- Choose width of each tree level with a horizontal slider
- Able to use vertical and horizontal scroll functionality to pan around the document
- Can save and open documents
- Can export to a text file that is human-readable and uses indentation
- Stretch goals
    - Sync over iCloud
    - Runs on iOS/iPadOS

System Architecture

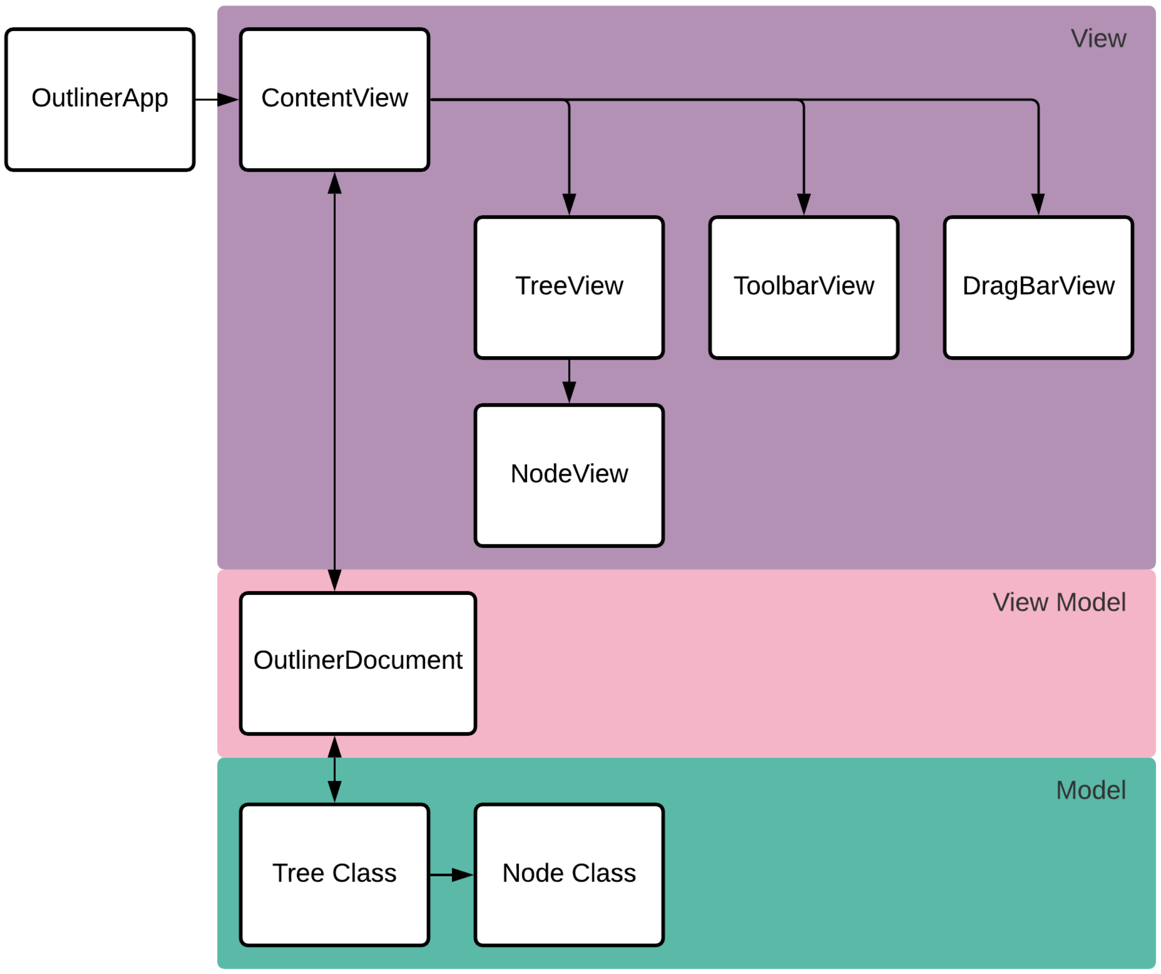


Figure 1: Diagram of our system architecture

The system architecture for Outliner is a MVVM (Model, View, View Model) architecture.

The Model, shown in green on Fig. 1, contains components responsible for holding the data for the application. Inside of the Model category there are two classes which build up and manage a tree data structure: `Tree` and `Node`. The `Tree` class holds a reference to the root node of the tree and contains functions for manipulating the tree structure. Within the tree, the `Node` class is what comprises the nodes, with each instance holding text to display to the screen, text rendering settings, and details such as whether its children are hidden or visible.

The View Model, shown in pink in Fig. 1, acts as an intermediary between the Model and View, receiving intents from the View to update the Model and updating the Model correspondingly as well as receiving notice from the Model that updates have occurred and telling the View to redraw. OutlinerDocument is a straightforward implementation of this functionality.

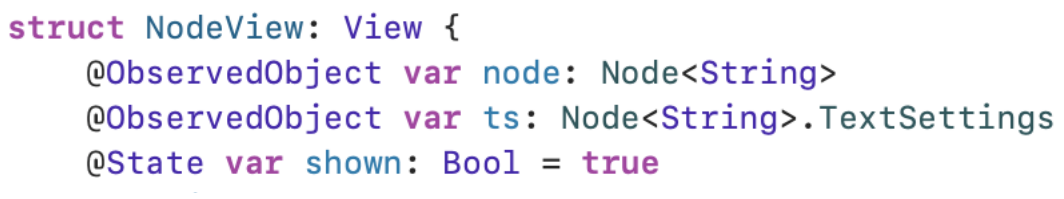
The View, shown in purple in Fig. 1, is what is drawn to the screen and handles listening for button presses and keyboard shortcuts. It consists of a number of custom views which recursively build up the final window. ContentView is a view which arranges all of the other elements onto the screen; it is the part that glues all of the pieces together. The TreeView builds up a tree diagram using NodeViews to draw all of the nodes in the Model. A ToolbarView is created to add buttons into the titlebar of the application, these buttons allow first time users access to the majority of interactions that are possible with the software without them needing to learn a bunch of keyboard shortcuts. DragBarView creates a bar similar to one that would be present in a table application like Excel for resizing columns, which allows us to resize all of the nodes in a column.

OutlinerApp exists outside of this MVVM model and allows for new instances of the application to be run. Multiple windows and theoretically multiple tabs in a window can be opened instead of only a single instance.

Technical Design

Because Swift is a relatively new programming language, and SwiftUI is even newer, it has many modern features for easily creating applications, but it also comes with its own quirks. The biggest change that SwiftUI uses is the *reactive programming paradigm.* Programming languages like Java and C++ have traditionally used an *imperative* paradigm, where objects are instructed to change in the code. However, SwiftUI allows for us to design our applications to react to these changes when they notice a change.

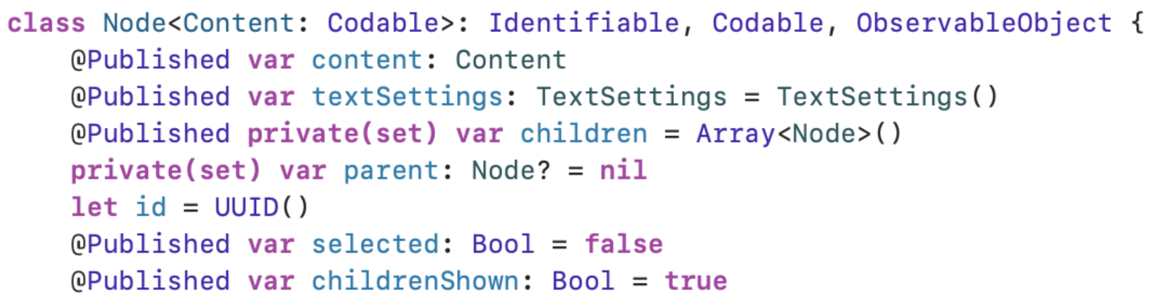
Objects inside of a class or struct can be given something called a *property wrapper* that changes how the parent behaves with regards to said object. Take a look at the code snippet in Fig. 2, which is responsible for displaying the contents of a Node in an appropriate way.



```
struct NodeView: View {
    @ObservedObject var node: Node<String>
    @ObservedObject var ts: Node<String>.TextSettings
    @State var shown: Bool = true
```

Figure 2: `NodeView` struct, showing our use of property wrappers in SwiftUI

Before the first two variables, you can see `@ObservedObject`, one of SwiftUI's property wrappers. This instructs the `NodeView` to subscribe to the `node` variable, causing the `NodeView` to be redrawn each time something in `node` changes. Inside of our `Node` class (Fig. 3), we added the `@Published` property wrapper to any variable that affects how it is drawn, which instructs Swift to listen for changes in these variables and tell the associated `NodeView` to redraw itself when changes are published. The `shown` variable in the `NodeView` also has the `@State` property wrapper, which designates the variable as a "source of truth," meaning it can be written to and read from by Swift in order to draw the UI appropriately.

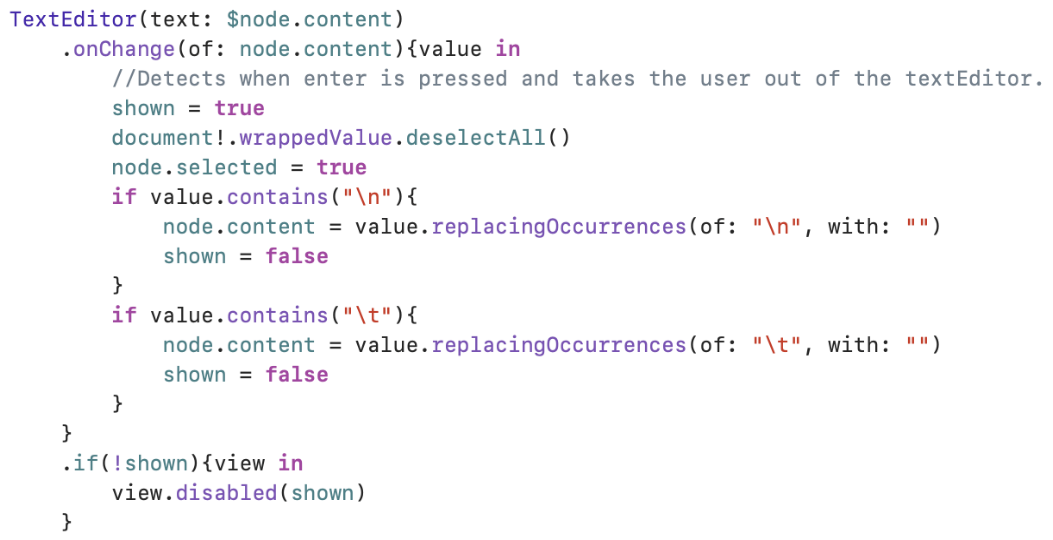


```
class Node<Content: Codable>: Identifiable, Codable, ObservableObject {
    @Published var content: Content
    @Published var textSettings: TextSettings = TextSettings()
    @Published private(set) var children = Array<Node>()
    private(set) var parent: Node? = nil
    let id = UUID()
    @Published var selected: Bool = false
    @Published var childrenShown: Bool = true
```

Figure 3: `Node` class showing the use of `@Published` on variables combined with the `ObservableObject` protocol

These property wrappers allow for us to create fast, flexible applications without being responsible for redrawing components manually, and are only available in the SwiftUI framework. However, because SwiftUI is so new, there are still features that are missing from it, which can lead to some unique workarounds when trying to build applications.

One such missing feature in SwiftUI 2 is the inability to directly manage the focused elements on screen[1]. This led to some issues in our application, as we intended to have editable text fields on screen that you could finish editing by pressing return on your keyboard. Instead, we came up with a workaround to test for a newline character at the end of a node each time the text is changed. This code can be seen below in Fig. 4.



```
TextEditor(text: $node.content)
    .onChange(of: node.content){value in
        //Detects when enter is pressed and takes the user out of the textEditor.
        shown = true
        document!.wrappedValue.deselectAll()
        node.selected = true
        if value.contains("\n"){
            node.content = value.replacingOccurrences(of: "\n", with: "")
            shown = false
        }
        if value.contains("\t"){
            node.content = value.replacingOccurrences(of: "\t", with: "")
            shown = false
        }
    }
    .if(!shown){view in
        view.disabled(shown)
    }
```

Figure 4: The body of our `NodeView` struct. The `TextEditor` is where the user can type in and edit their nodes, and will be deselected and disabled in order to unselect it.

There are other issues that we have run into, such as not being able to click on things that have a `Color.clear` background, specific keybinds not working in certain contexts, etc. That being said, these problems have been part of the fun of learning this new programming paradigm and have helped us learn a lot more detail of how SwiftUI works.

[1] This feature is being added in SwiftUI 3, which will be released later in 2021

Quality Assurance Plan

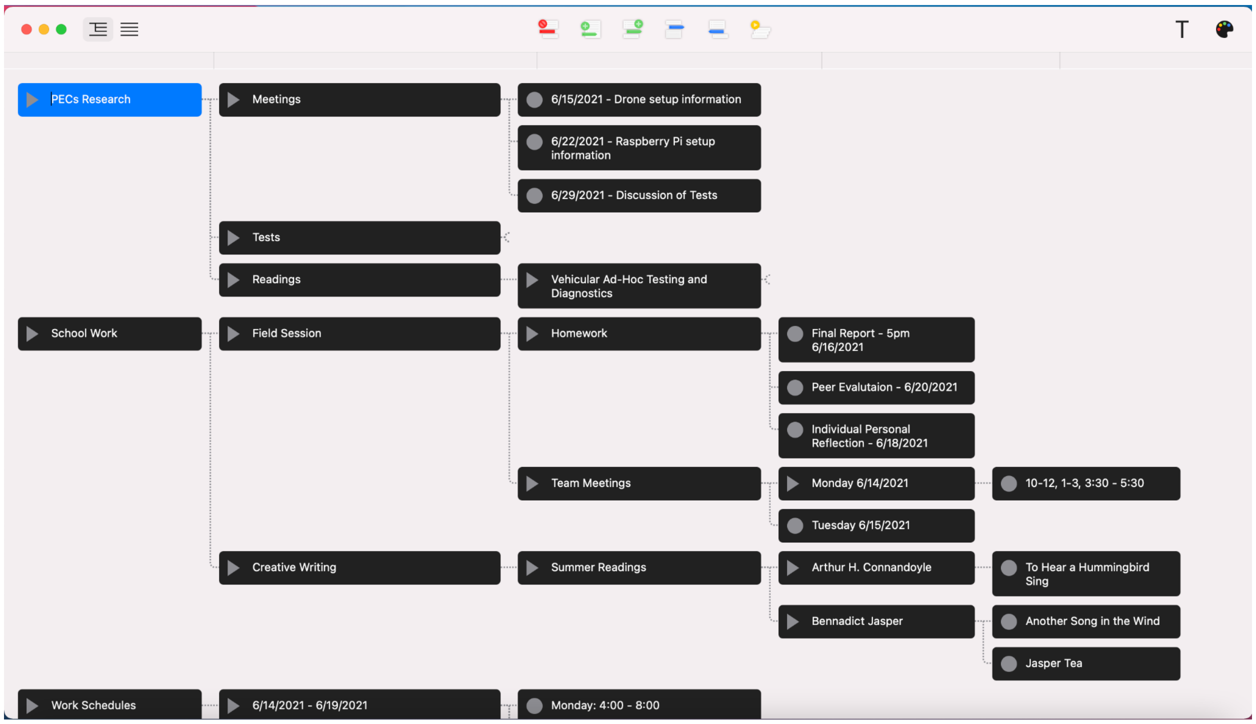
Our quality assurance is divided into two major categories: the code and testing. The coding aspect of our plan revolves around creating and maintaining code that is easy to read, edit, and understand. This process uses coding standards, refactoring, architecture, and design to make the code less complicated and better able to be understood by teammates and by third parties. Testing focuses on making sure that the code works as intended. This process included code reviews, UI testing, unit testing, and integration testing to ensure that each portion of the code works correctly.

- **Code Reviews**
    - We did code reviews during pull requests.
        - Ensures that the code that gets merged into the main branch is of a high quality and doesn't have major errors.
    - Code reviews looked at all of the changes made in the branch and tested each of these changes.
- **User Interface Testing**
    - Because a lot of our program relies heavily on the UI and how the data is displayed, it was important that we made sure our UI is displaying correctly.
    - This was done through running the program and using the interface, using each of the new features.
        - The new features in the User Interface that had underlying code that had not been implemented yet were tested by having simple code output text indicating that the feature has been successfully activated.
- **Unit Testing**
    - Writing tests to ensure basic functionality like copying, moving nodes, etc. were working was helpful to ensure that future features don't break.
    - Unit tests helped ensure that the underlying data structures worked as expected and stored and organized the data in the correct fashion.
    - Can be automated in Xcode to ensure new code doesn't break old features.
- **Integration Testing**
    - Writing tests ensured that features were working when combined together is important.
    - Necessary because the behavior of a function/module must be consistent even when new features are added.
- **Comments**
    - Code was commented to increase readability and to provide a deeper understanding of functionality of code components.
        - Ensured that the project was able to be maintained in the future and helped with code reviews.
    - Comments also communicated the limitations of any code components.
- **Coding Standards**
    - Everyone tried their best to follow the standard coding conventions for Swift which will make future edits easier.

  - Code was cleaner and any developers outside of our team will be able to read and understand our design choices and documentation.
  - Updated the code to conform to the modern APIs allows for easier program maintenance.
- **Refactoring**
  - Refactoring kept the code simple and easy to continue development and maintenance.
  - Made code more resilient and less difficult to understand when reducing complicated functionality to simpler equivalents.
- **Architecture and Design**
  - Diagrams provided a visual representation of how the different elements of our program interacted.
  - As our code became more complex, diagrams were a concise way to portray and explain implementation.
  - Well-designed code is easier to implement and to expand upon.

Results

Team Illus-tree-ous is very satisfied with the work that we did during this field session. We were successfully able to recreate the core behavior of the Trees 2 software during the session, but we were not able to implement everything that we and the client had hoped to. That being said, we have learned a significant amount about reactive programming, UI design, and other software engineering skills. Below is a picture of our final product as well as a list of our results for this semester:



- **Implemented Features**
  - Custom buttons in Apple menu bar for main actions
    - New document, save/open, export to text file, etc.
    - Add item, add child, indent/outdent, etc.
  - Keybinds and toolbar with buttons for main actions
  - Multiple documents open in separate windows
  - Copy/cut and paste nodes and their children
  - Node icons denoting if it has children and if they are hidden or visible
  - Collapsible subtrees
  - Lines connecting parent and child nodes
  - Ability to change font settings per node
    - Bold, italics, foreground and background color
  - Adjustable width for each level of the tree
  - Horizontally and vertically scroll around the document
  - Export as a human-readable text file

- **Unimplemented Features**
  - The additional indented tree view (list view)
  - Drag and drop
  - Exporting to PDF
  - Multiple documents open simultaneously in tabs
  - Modifying node font, font size, and border color
  - Stretch goals
    - iCloud synchronization
    - iOS application
- **Performance Testing Results**
  - The application responds quickly to user input and no delays are apparent between clicking a result and the result appearing on screen.
  - The UI displays well on macOS screens and the window is resizable and responsive.
  - All unit tests pass in the program's current state.

Lessons Learned

Throughout the field session, our group has learned a lot about reactive programming in SwiftUI and software development in general. Swift was a new language to us, and so was the MVVM pattern. Over the past five weeks, we have learned how to utilize the unique features that this pattern and language provide for us in order to make better, faster, more flexible software. We have also learned a great deal about software development in general, particularly with regards to our "soft skills." We feel that we have a far greater understanding of working with a client and working in a team, problem solving skills, and how to delegate work to cater to an individual's skills.

Future Work

As noted in the results section, there are several features that were unimplemented at the end of our time working on the project. These features, in addition to several bugs and glitches in our final product are what will encompass the future work on this project. These items are, in general, either ease of life improvements, or additional features that do not directly influence the base functionality of the software. Additionally, some accessibility issues persist in the software that will need to be addressed. As SwiftUI continues to receive updates, these features will become easier—or possible—to implement.

- Unimplemented Features mentioned above
- Interface glitches
    - Hotkeys cease to work in full screen
        - SwiftUI glitch
            - Other sources online indicate that other programmers have come across the same issue
        - No known workarounds
    - DragBar Issues
        - Dividers are spaced too wide and do not correspond correctly to the nodes
            - Likely due to the use of .padding modifiers on the NodeViews but is difficult to confirm
        - When nodes reach the edge of the screen, they stop expanding
            - When there is a very deep tree, if a middle or left node's corresponding drag divider is selected, if it is dragged to the right until the rightmost nodes touch the side of the screen, the rightmost nodes cease to expand, but the drag divider continues to move.
            - Ideally, when the rightmost nodes reach the edge of the screen, they should move off the edge of it and allow the user to scroll in order to see them.
    - Replace TextEditor in NodeView with the new TextField in SwiftUI 3
        - Able to remove background from NodeView
        - Able to manage focus better using the new FocusState property wrapper
- Accessibility issues
    - High contrast appearance
    - Larger toolbar and buttons
    - VoiceOver integration
        - Requires text to speech and shortcuts to activate it

Appendix

Using the app:

- Selecting nodes
    - Nodes can be clicked on anywhere outside of the text area, which includes the icon to the left of the text or on a small border around the text where the cursor is a standard pointer.
    - Multiple nodes can be selected and deselected by using Command+Click to click on each node individually toggling the node being selected or not.
    - All nodes can be deselected by clicking outside of any node in the tree area.
- Modifying node text
    - The text in nodes can be modified by clicking into the text on a node and beginning to type.
    - To exit editing the node, Enter or Tab can be pressed.
- Keyboard shortcuts
    - Standard shortcuts for open, new, and save are implemented.
    - Selecting one or more nodes and pressing Tab will indent all selected nodes a single level.
    - Selecting one or more nodes and pressing Shift+Tab will unindent all selected nodes a single level.
    - Selecting one node and pressing Enter will create a new blank node under the current node.
    - Selecting one node and pressing Shift+Enter will create a new blank node as a child of the selected node
    - Selecting one or more nodes and pressing Command+C will copy the selected nodes and all of their children
    - Selecting one or more nodes and pressing Command+V will paste the copied nodes and all of their children below all selected nodes
    - Selecting one or more nodes and pressing Command+X will cut the selected nodes and all of their children
    - Selecting one or more nodes and pressing Command+D will duplicate the selected nodes
    - Pressing Command+LeftArrow will select the parent of all selected nodes
    - Pressing Command+RightArrow will select the children of all selected nodes
    - Pressing Command+Shift+E will export the file as a text document of the same name in the same folder as your working document
- Hiding nodes
    - Selecting one or more nodes and clicking the Toggle Family button will hide the children nodes of all selected nodes if any of them are currently visible and otherwise make all of the selected nodes' children hidden.
    - When a node has hidden children, the line connecting it to its children nodes changes into a line which terminates in a half circle in addition to having a triangle icon which denotes that this node has children:

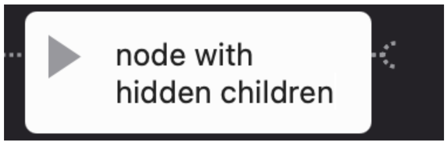


Figure 5: Screenshot of our node view with the hidden children indicator

- Changing colors
    - To change the text or background color of a node:
        1. Select the node.
        2. Click the “Choose text color” button in the top right corner.
        3. Select a color and then click on the A button to apply the color to the text or the Highlighter to apply the color to the node’s background.