### **Introduction**

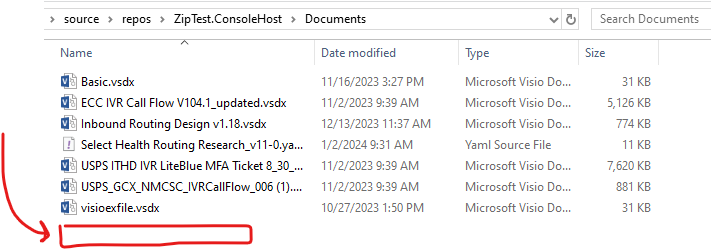
This report is intended to detail the usage of the QA Visio Parser program and enable it to best support a given Microsoft Visio file. It is intended to enhance the QA development experience by providing supporting information and assisting with test case creation.

### **Set-Up**

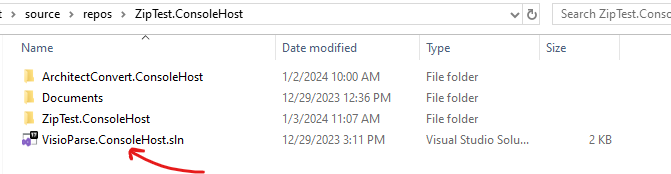
1. Ensure that Visual Studio with .NET support is already installed with a repository folder configured, then make sure that the folder for the project is contained inside of that repository folder.

Note: The default repository location for Visual Studio is C:\Users\{username}\source\repos, ensure that ZipTest.ConsoleHost is installed there.

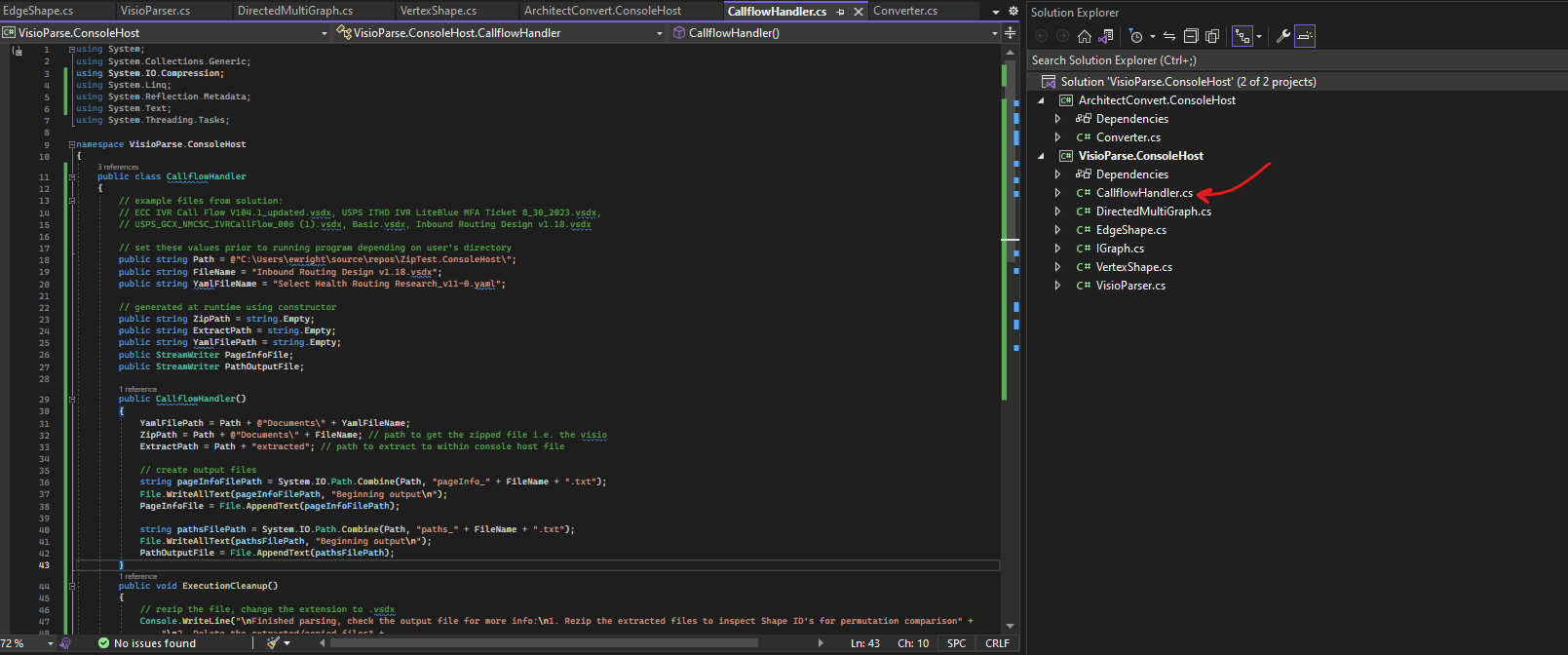
1. Open up the ZipTest.ConsoleHost project folder, the Documents folder is where supporting documents should be placed, for example the Visio file desired to be parsed. Multiple Visio files can be placed in here, but the program will only run one at a time.



1. Open up the ZipTest.ConsoleHost project folder and double click on the .sln file, it should open up the project in Visual Studio.

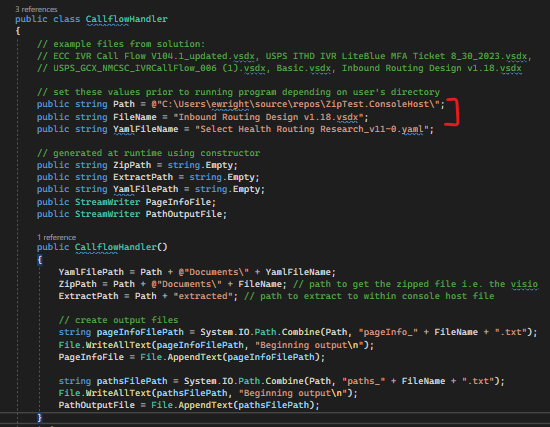
****

1. In the Solution Explorer tab on the right, double click on the file named “CallflowHandler.cs” to open it up.



1. To parse the given Visio file, the CallflowHandler needs to know where to find the file. So edit the following lines defining the Path and the FileName to be the path to the project file on the user’s computer and the name of the Visio file itself.

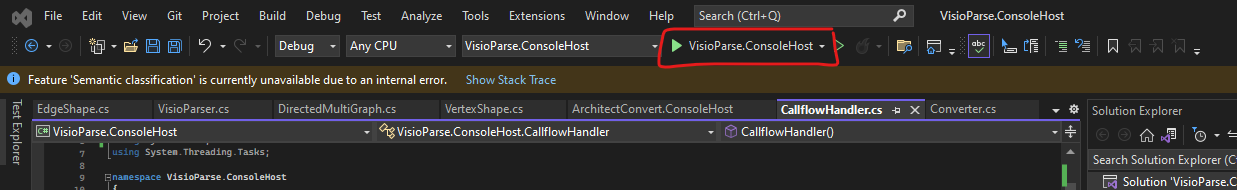
Note: These lines don’t need to be typed by hand, file paths can be copied (once inside the directory, right click on the ZipTest.ConsoleHost at the top and click “Copy address”, then paste it into the Path value) and the file name can be copied (right click on the desired Visio file, click rename, copy the text, then paste it into the FileName value)



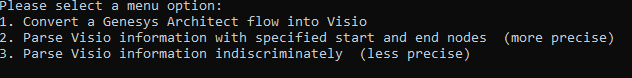
### **Parsing a Visio File**

The following instructions will guide a user through the program execution to select the right configuration settings and interpreting output. Note that the output will appear differently for each Visio due to the parsing of its information, so take caution when comparing the following steps as the output is produced from the provided example.

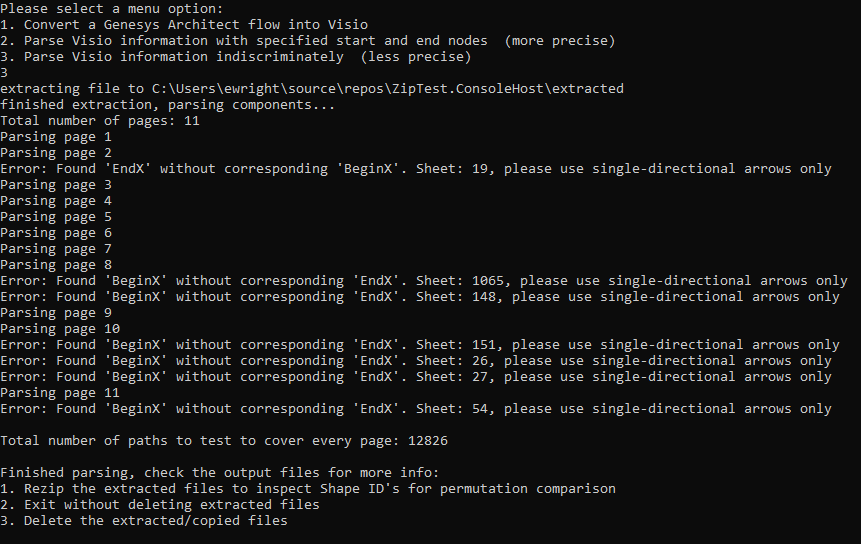
1. Click the execution button at the top to begin running the program



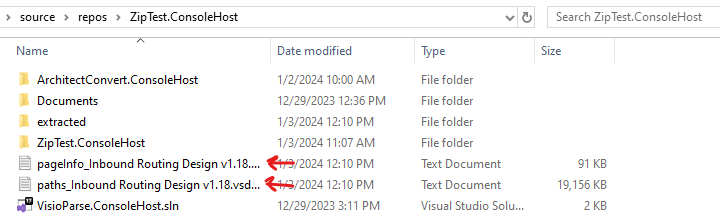
1. From the window popup, a list of menu choices will appear. To parse Visio files, we will focus primarily on options 2 and 3. With no knowledge of the contained Visio data, it is safe to pick option 3. Type “3” and press enter.

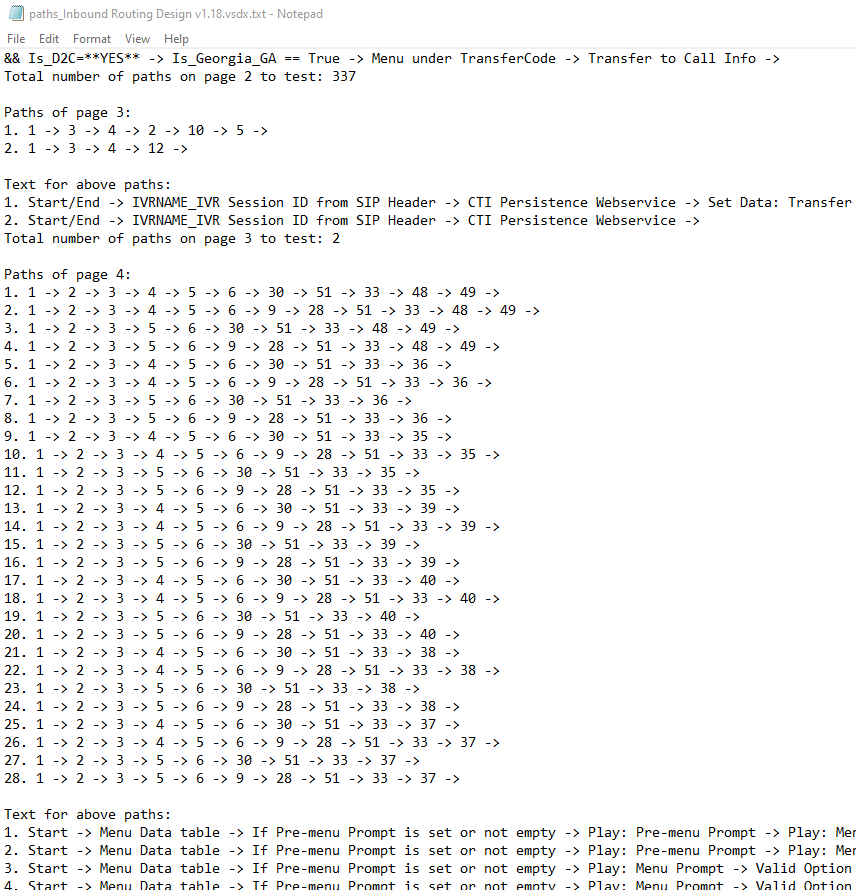


1. The program will parse the Visio and the console window will provide some benchmarks and logs. At the end of the parsing, we can see that the test for this example Visio produced 12826 total paths for every page.



1. Next, reopen our project folder and notice that there have been a few additions.
   1. In the paths file, we can see the details of each path generated by finding permutations between all possible start nodes to all possible end nodes. The paths are listed as the IDs for each shape, but are printed in text as well to be more readable and quickly interpretable.
   2. In the pageInfo file, we can see a breakdown of the internal information for each page, which can help interpret some of the paths as well as assist in extra configuration in later steps
   3. The folder named “extracted” contains the XML data that was extracted from the provided Visio and parsed, it can potentially be used in the subsequent step

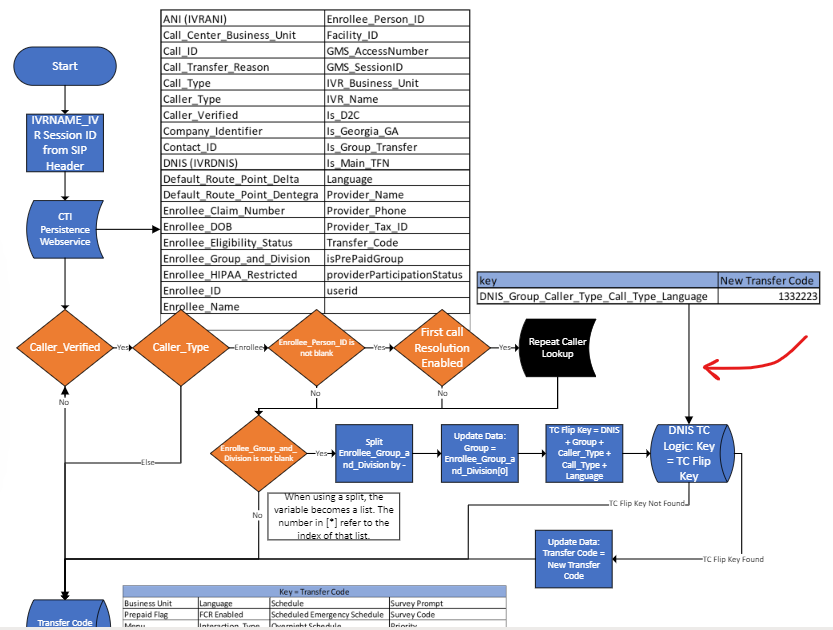


****



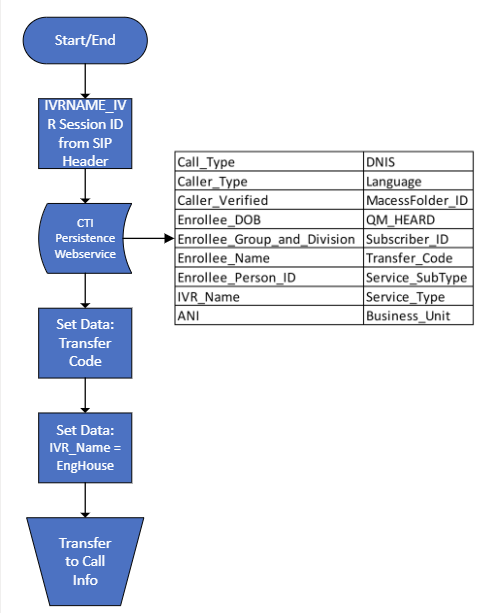
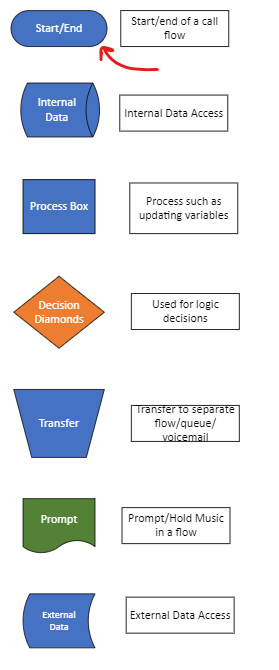
1. Back in the console window, more options are provided which deal with the extracted files. Note that these files are copied from the Visio, not converted. This means that they can be safely deleted and the original Visio is still contained in the Documents folder with no modification
   1. The first option will allow for zipping the extracted files back up into a modified Visio, but with all text replaced as the Shape ID to help with additional visualization of the permutations. To do so, press “1” and then enter.
      1. Note that there is a subsequent option to delete this modified file as well, just press “d” and then enter. Anything else and program execution will stop and the file will be retained
   2. The second option leaves the system as-is and exits to allow the user to keep the XML content. Note that the XML text fields have been overwritten with the ID. Furthermore, the “extracted” file must either be removed from the directory or deleted manually to allow for the program to run again, it won’t overwrite previously extracted contents. To do so, press “2” and then enter.
   3. The third option handles the deletion for the user and is the default choice. It simply gets rid of the “extracted” file, the output files remain there for the user to view even after program execution is done, and those files are overwritten on each run as well so they don’t need to be cleared. To do so, press “3” or anything except “1” or “2”, then press enter.

### **Modifying Parsing Settings**

For indiscriminate parsing, start nodes are chosen from nodes on the page which have outgoing edges but no incoming edges (with end nodes being the opposite). This works for most cases, but can cause some troubles with unintended nodes such as the following case in the used example showing how the table would be counted as a start node and leading to a large overestimation of paths in more complex files:  
  


#### **Note on Design Pattern Concerns**

As a result, we can be more specific in our parsing by permitting user-defined start and end nodes. Because of this, users are given some design freedom in their Visio choices, but try to keep starting and ending locations uniform across every page so that the configuration settings for the parsing can be as precise as possible. For example, the Visio used by this document uses a uniform shape for each start/end node. However, sometimes the author defines start nodes as “Start/End”, and there is no end node. In cases like this, there should be only one path, but instead we complicate things into one of two possibilities:

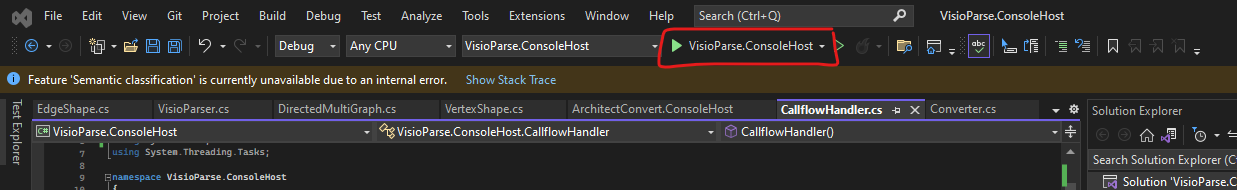
1. The user chooses to parse indiscriminately, whereupon the example page would return two paths (one from Start/End to the table, and one from Start/End to the bottom block labeled “Transfer to Call Info”).
2. The user chooses to parse precisely, either by providing the Start/End Master ID (which will be explained momentarily) or by providing the text “Start” and “End”. In either case, no paths are found in this page because no end node can be found for the options, and in the case of the text option there isn’t even a start node found as neither text will match “Start/End”.  
   

In either case, neither option will find the correct paths, because the Visio does not follow uniform constraints. Because of this, users must be careful in their design choices if they wish to use an automation tool. The choice to be more precise also requires an element of precision when designing to ensure the closest results, which means that users must clearly define where their flows begin and end and stick to that determination for the remainder of the file.

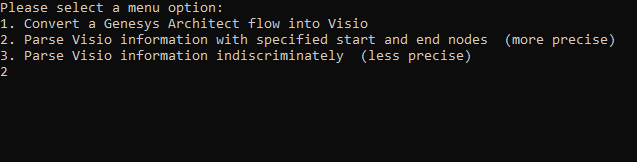
#### **How to Parse Precisely**

To enable the parser to find paths with constraints, we can use the following steps

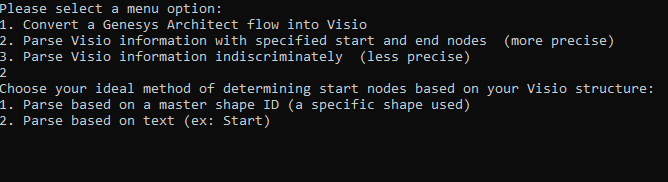
1. Click the execution button at the top to begin running the program



1. From the window popup, a list of menu choices will appear. To parse Visios with specified constraints, select option 2. Type “2” and press enter.

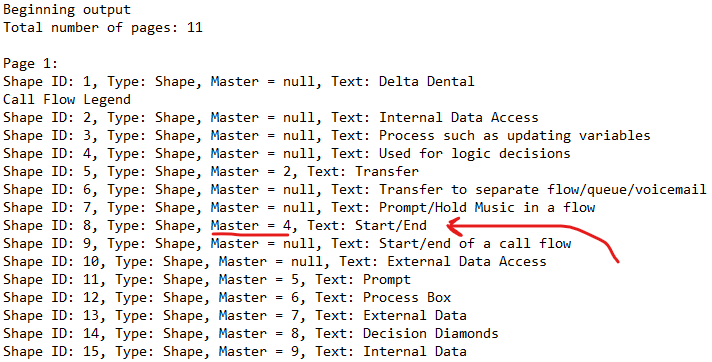


1. From the following menu, we can choose between defining nodes by Text or defining nodes by Master IDs. Enter “1” or “2” depending on which option is more consistent in the given Visio file.

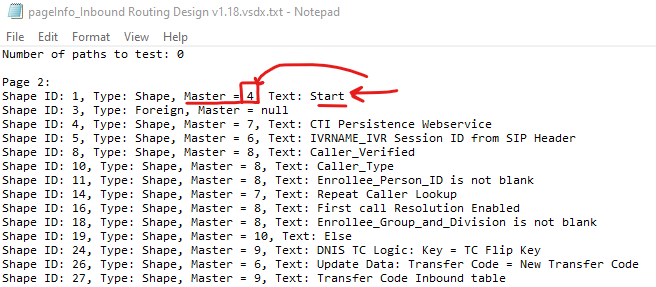


* 1. To parse by Master ID, reopen the pageInfo file from the indiscriminate parsing. If the given Visio includes a legend on the first page, then simply find the Master of the desired shapes to use. Then enter the start node Master ID followed by the end node Master ID

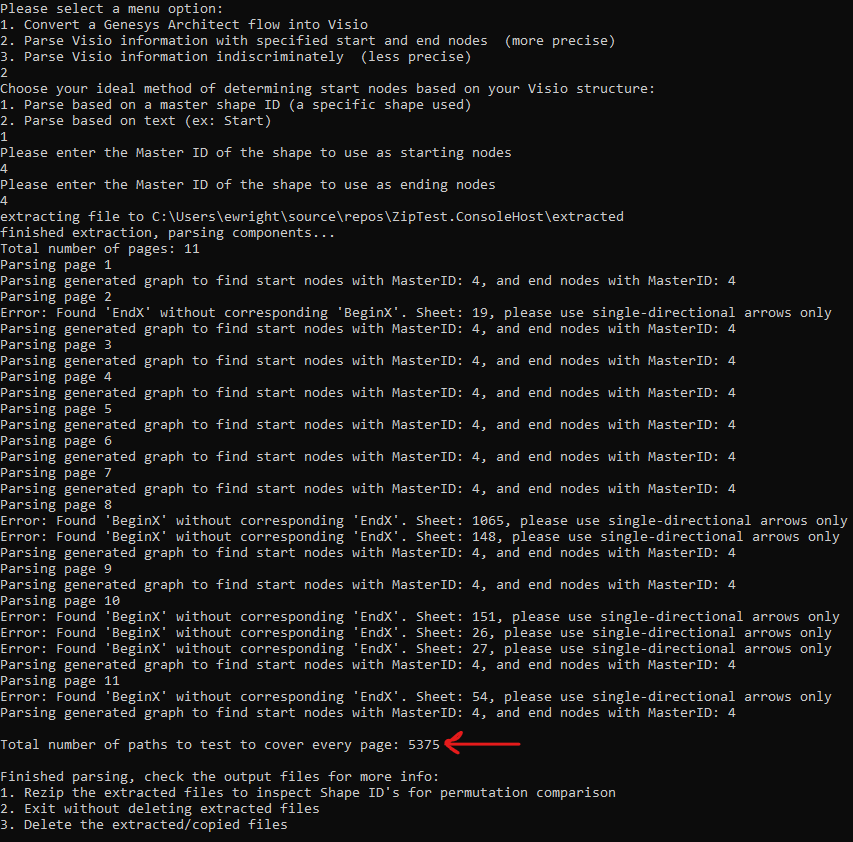
Note: if start and end nodes have the same Master ID, the program will still find the correct paths by interpreting their categories from their directed edges, so entering “4” and then “4” again is acceptable for flows which start and end using the same shape:



* + 1. Alternatively, any instance of a start or end node that can be found in a page can be used, as long as the Master is consistent and all desired nodes adhere to the same ID:

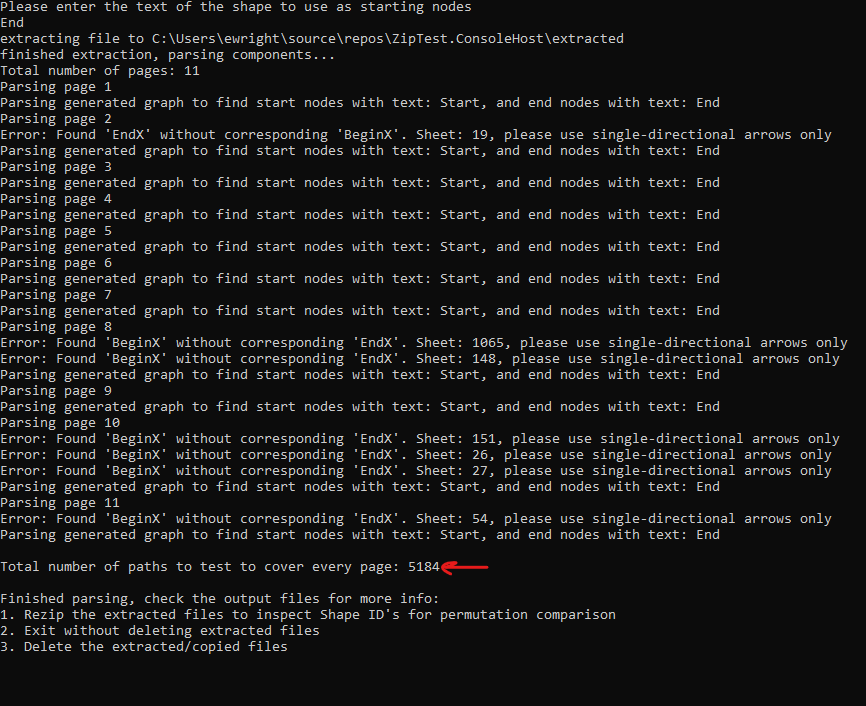


Example output (5375 test cases):



* 1. To parse by text, enter the text used by the desired start nodes, then enter the text used by the desired end nodes (ex: “Start”, then enter, then “End”, then enter)

Example output (5184):

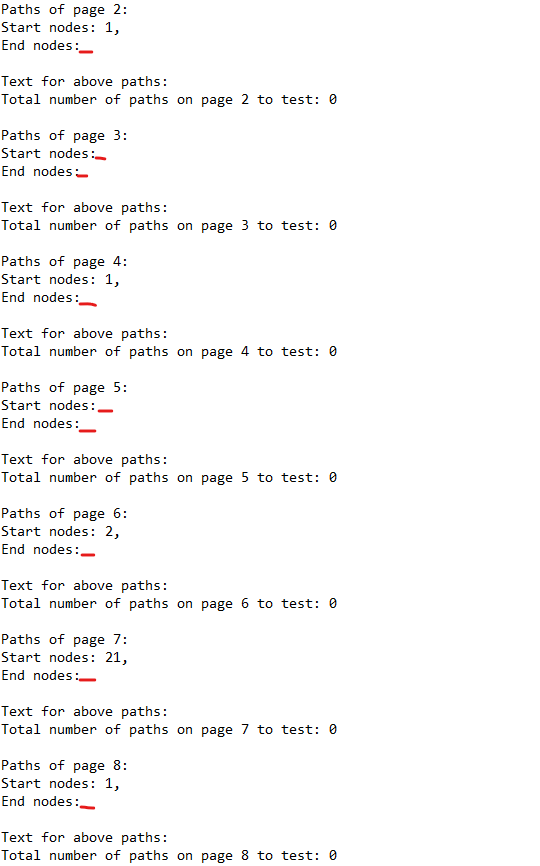


1. Handle final execution logic by referring to Step 5 of “Parsing a Visio File”.

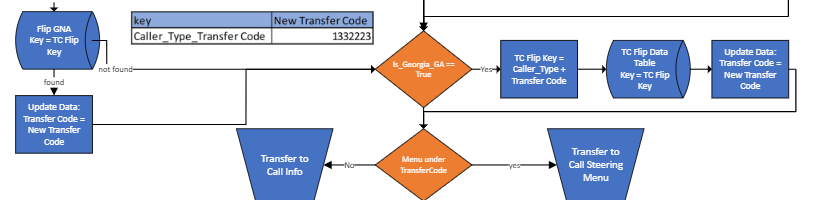
### **Interpreting Results and Identifying Issues**

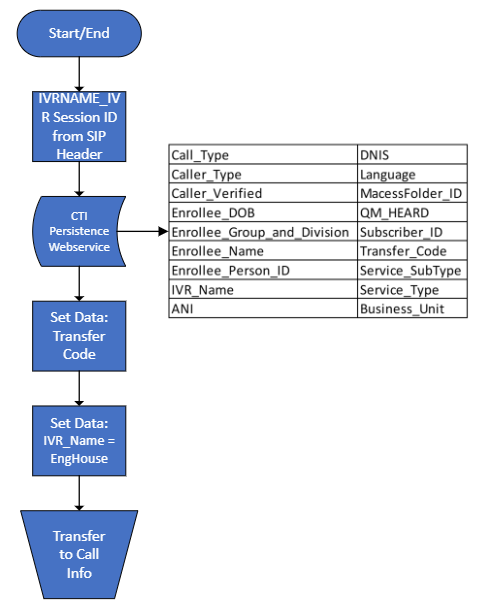
As can be seen, the total path count is much lower with added specification. However, this is primarily due to design styles being too inconsistent in the example Visio to accurately determine specified node locations, with end nodes especially. Here are a few provided examples to help document the importance of sticking to a design:

* On several pages, the specified design from the legend isn’t followed in some way. When parsing by text, we get the following output (where each red mark is a value that is missing from specified parsing which is preventing paths from being calculated):

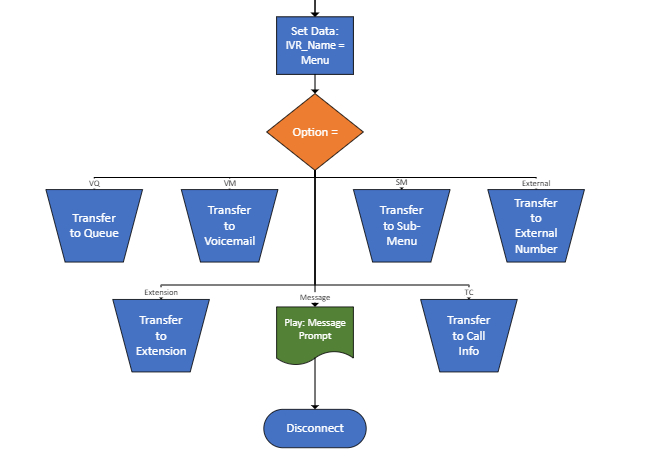


Here are there reasons:

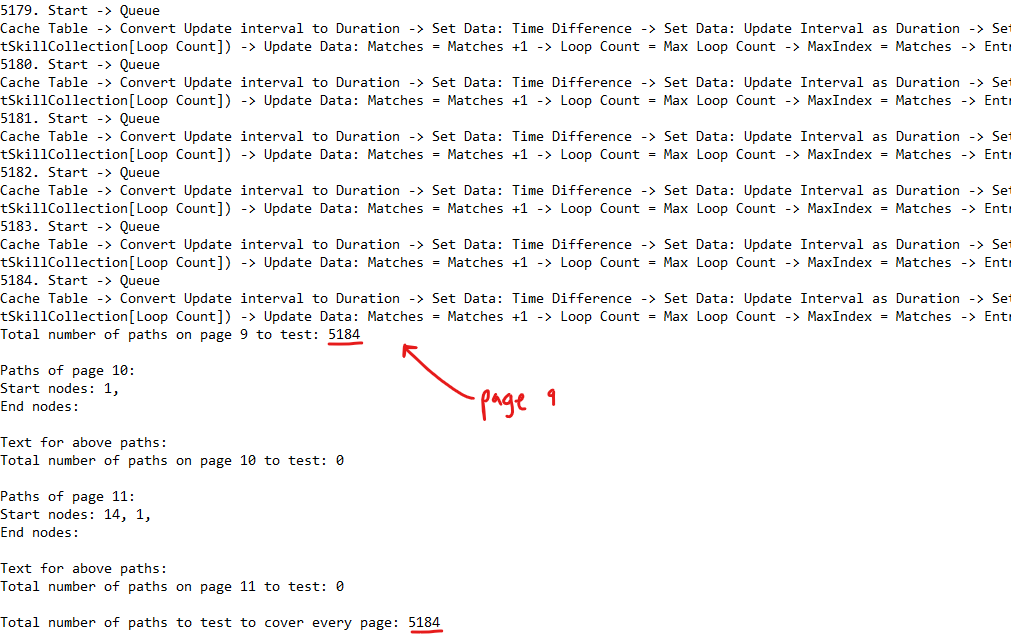
* + On Pages 2 and 3, the end node shape and “End” text is not used, which means that neither specified parsing method will be able to detect any paths in the entirety of both flows:  
     



* + On Pages 4 and 11, the end node is called “Disconnect” or something else instead of “End”. Fortunately, it is still using the same Master shape as the defined end node, but it means that parsing by text would not work for this flow because the end node would not be located:

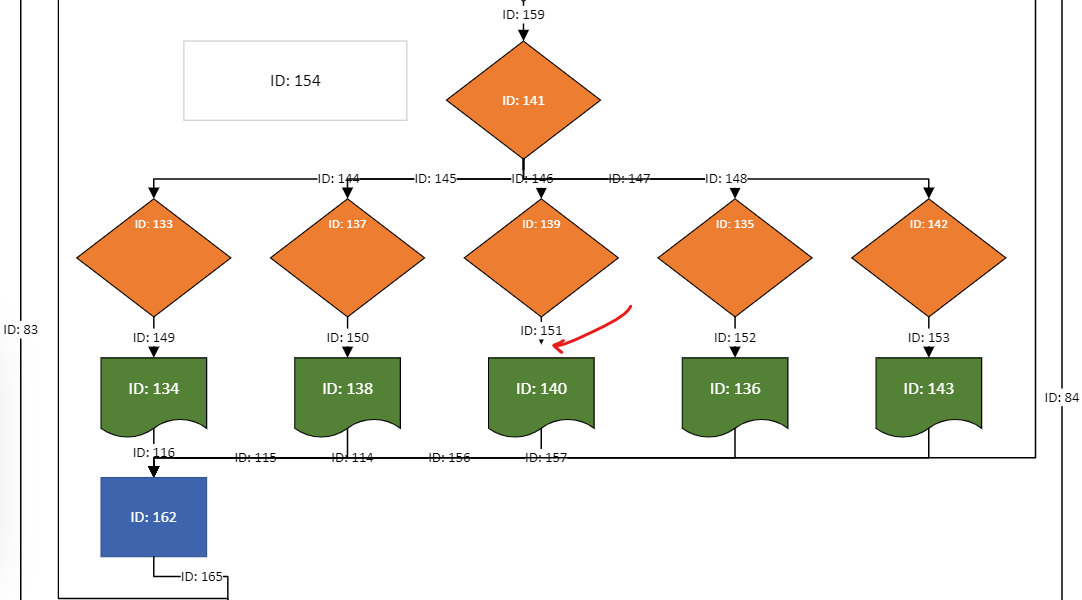


* So at a first glance, the 5184 test cases seem accurate. However, the reality is that every page (save for page 9, which happens to be the largest) has a design formatting issue, causing the true number of paths to be incalculable when parsing by text because the legend is not followed - which can be seen by the empty paths and missing nodes:

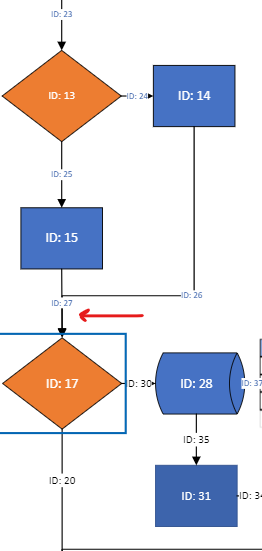


#### **Design Debugging Techniques**

Another useful technique to use while debugging the design of a flowchart is the ID mapping functionality. For example, after we extract the files and remap it back into a Visio file, we can observe the paths of page 10 which should have only one start node (Shape 1) and one end node (Shape 171):  
  
After rezipping up our modified Visio, we can see that this page has some faulty connections which cause unintended start nodes when using indiscriminate parsing and also prevents the path from the true start node from ever reaching the true end node. In the following image, it’s clear to see how Shape 140 incidentally becomes a start node, while also cutting off a traversal path from the actual start node preceding it:

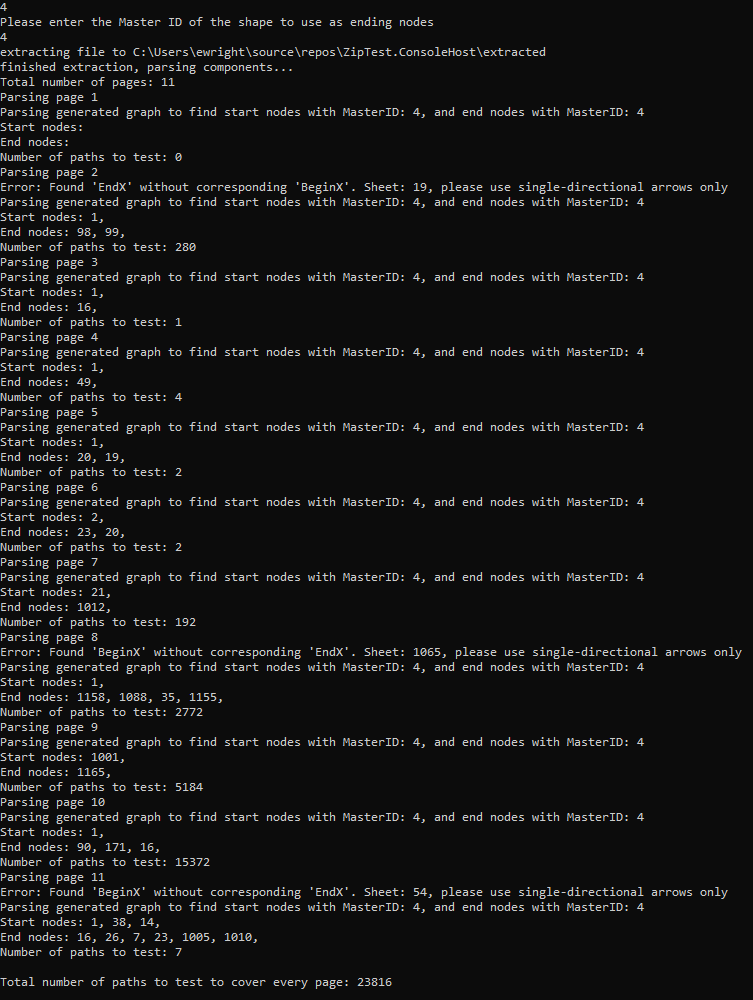


Furthermore, because Shape 17 is labeled as a start node in the indiscriminate parsing, we know that this arrow design isn’t actually connecting to the node. If the edge is not anchored to the node, then it will not be considered connected, which causes the parsing tool to discard the corresponding edge entirely.



This technique can assist with formatting the Visio in such a way that transfers it from being just human-readable into machine-readable. Be sure to analyze the results of the parsing to ensure that the Visio is designed in an optimal way using best practices.

After editing the Visio to patch up design inconsistencies and adding explicit start and end nodes, we reach this output:



23816 total tests using the specified Master ID parsing, far from the original number of 5375, proving the importance of ensuring that the Visio designs are completed before they are used for test generation.

### **Conclusion**

In conclusion, the Visio parser can help determine test cases for QA purposes and provide various useful information, but it requires careful consideration of design to ensure that the information is accurate.