**Chapter 4: Design**

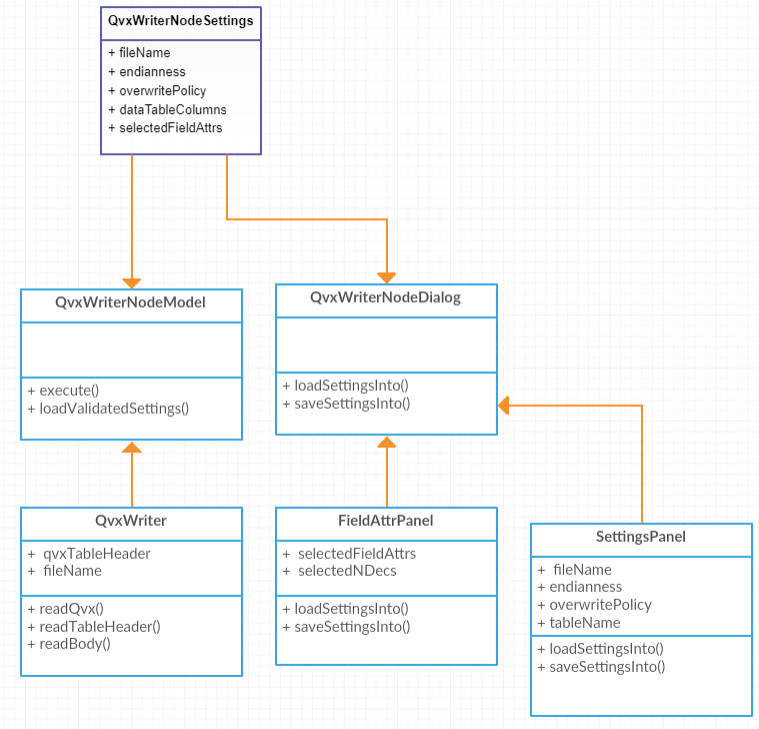
For both the Reader and Writer portions of our project, there were two major components: File IO with qvx files and Knime application code. We used Java to complete both of these components. To complete the Knime part, we had to install KNIME SDK. This sdk allowed us to save and load Knime settings, data types, and data, and it allowed us to create the actual Knime node itself. We did not need to use the KNIME sdk for the file IO part; this part was done mainly by using pure Java logic (i.e. reading and writing binary files byte-by-byte).

After setting up the sdk, we installed a Knime plug-in, then used the New Node Wizard to generate our Knime node classes. For the reader node, we had to add a file chooser. When the node is executed, the Qvx Reader gets the file name from this chooser, then opens the file. The Qvx Reader first saves all of the file contents into a byte buffer and identifies the zero-separator byte. Any contents before the zero-byte are the table header and any contents after are the table data. JAXB is used to parse the table header XML portion. Then, the Field Headers from the Table Header are saved. Next, each row of data from the data portion is read, in order to create the Knime data table. The length (in bytes) of each data item depends on the corresponding field header. The field header also lets the Qvx Reader know how the data should be stored in Knime (string, integer, double, etc.). In qvx format, dates are often stored as doubles that represent the number of days since 1900, so the Qvx Reader had to have code that converts these doubles to the corresponding Calendar object, so that dates could be stored in Knime correctly.

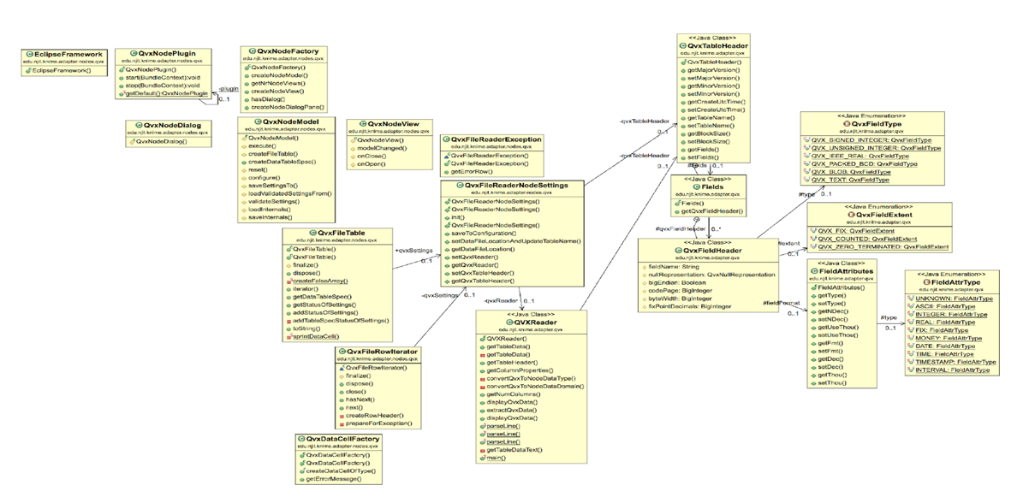
Our reader node can also detect if a qvx file is formatted with non-ideal types. For instance, in some of the test files that we received, certain qvx column were labeled as “double” even though they actually only contained integers. When our code sees a double column, it does an extra pass through the data to check for this case, and if all of the numbers are integers, the Knime data table will store the column as integer values (rather than double values). Similarly, some of the test qvx files contained columns with String values that were clearly supposed to be date values (such as “12-1-2016”). When our code sees a String column, it does an extra pass through the data to try to convert the Strings to dates; we did this conversion by using regular expressions. If all of the data could be converted successfully, the Knime data table will store the column as date-time values (rather than String values). Qvx files can also be read from the internet; this approach requires a different method of reading the input file, but after the initial file read is complete, the rest of the procedure is exactly the same as reading a file from the local machine.

For the Qvx Writer node, we had a file chooser where the user specifies what file to write to. The writer node must take a Knime data table as input. The writer node allows the user to decide how each Knime data column will be stored in the qvx file. Depending on how the data is stored in Knime, the available storage options are different. For example, if the data is an integer in Knime, it can only be stored in qvx as a numerical type (such as REAL, FIX, or INTEGER). These storage options are available to the user in drop-down menus for each column of Knime data. Our writer node also sets default qvx types, so that the user does not have to worry about configuring these settings if they do not want to.

The Qvx Writer first has to extract the Data Specifications of the input Knime data table. These specs are used to help our node decide what the default qvx type for each column should be (which can be overwritten by the user, as specified above). When the node is executed, the QvxWriter class first gets the selected qvx types from the Field Attributes Panel. The Qvx Table Header is constructed based on these qvx types. Then, we go through each value in the Knime data table, and we convert each value to the appropriate byte value depending on the Field Header. For example, if the corresponding Field Header specified that the value should be stored as an integer in the qvx file, then the value would take up 4 bytes in the qvx output file.



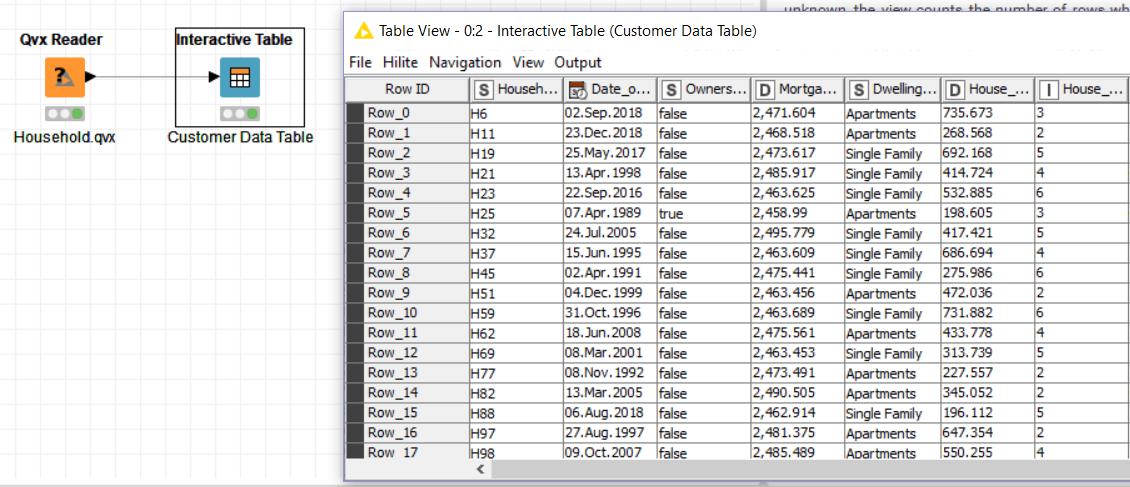
This is a class diagram for the QvxWriter project. The QvxWriterNodeDialog is the user interface. The Node Dialog contains two main panels: FieldAttrPanel and SettingsPanel. When the user clicks on the node, the QvxWriterNodeDialog node loads the saved settings from its QvxWriterNodeSettings object into both of the main panels. When the user clicks “Apply” (save settings), the values from each of the main panels are saved into the QvxWriterNodeSettings object. These QvxWriterNodeSettings are shared by the QvxWriterNodeModel and QvxWriterNodeDialog, so that when one of them updates the settings, the other one receives the same changes. The QvxWriterNodeModel implements the logic of writing the QVX files, based on the QvxWriterNodeSettings.



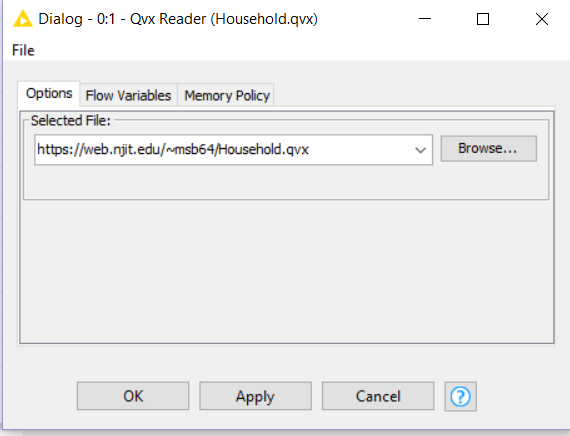
This is our ER Diagram for the Qvx Reader node. Our QvxReader class does the logic of reading the qvx file. This class contains a QvxFileReaderNodeSettings object, which contains a QvxTableHeader object. The QvxReader stores the information from the QvxTableHeader, so that the reader can understand the data format of the specific file, so that it can read the file correctly. To set the QvxFileReaderNodeSettings, the QvxNodeDialog (user interface) class is used. On execution, the QvxNodeModel instantiates a QvxReader object, which calls the “readQvx” function to parse the specified qvx file.

**Chapter 5: Development**

We successfully created a Qvx Reader node. This node allows the user to read from a qvx file on the local machine or from the Internet. This node produces a Knime data table on the output port. This means that the data read from the qvx file can then be passed into the Knime workflow to be further analyzed/transformed. After the qvx data is processed in Knime, the user can either create a data visualization or export the data to a csv/Excel file. The table generated by the Qvx Reader will always store the data in Knime in an appropriate way. In other words, the Knime data format should be based off the qvx format specified in the qvx table header. The exception to this is if it would make sense to store the data in a different way than specified (for instance, if the qvx type is String, but the values are clearly dates). Our solution supports the reading of all 10 Field Attribute Types mentioned in the public qvx specification, meaning it can support dates, money, integer, real numbers, etc.

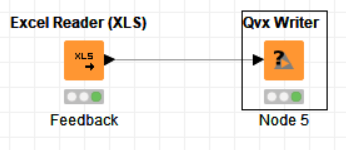


This screenshot shows an example Workflow that uses our Qvx Reader. Data is loaded from a file named “Household.qvx”. On the right is the table that is produced by our node. Various data types are successfully loaded, including String, Date Time, double, and integer. In this specific example, the date and integer columns were actually stored as Strings and doubles respectively in the qvx file, but they are stored correctly in the Knime data table. There are no null values shown here, but our code appropriately stores them as Missing (empty) Data Cells in Knime.

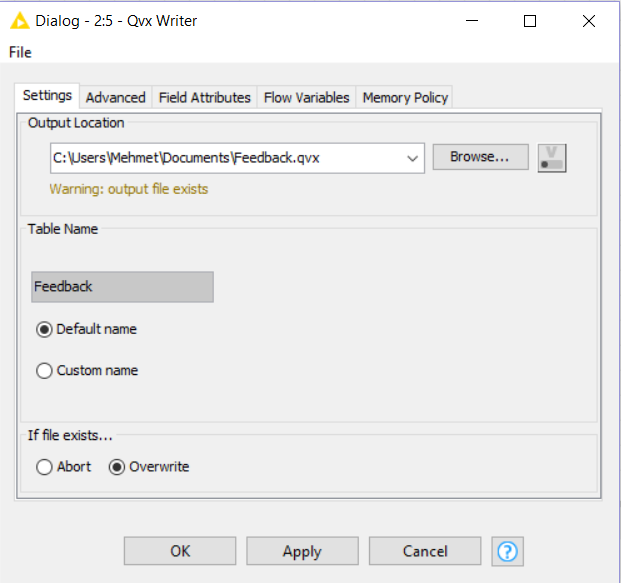
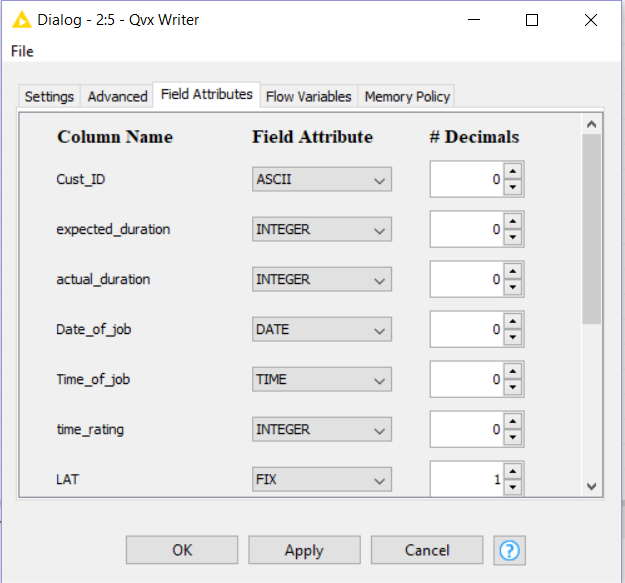


This is our Qvx Reader user-interface.

We also made a Qvx Writer node. This node takes a Knime table on the in-port (for example, a table generated by a csv reader), and then generates a qvx file. This qvx file can be loaded into QlikView. In QlikView, the data can be used to create a data visualization, as shown at the bottom of this page. In our Knime node, the user first chooses an output file location for the qvx file.

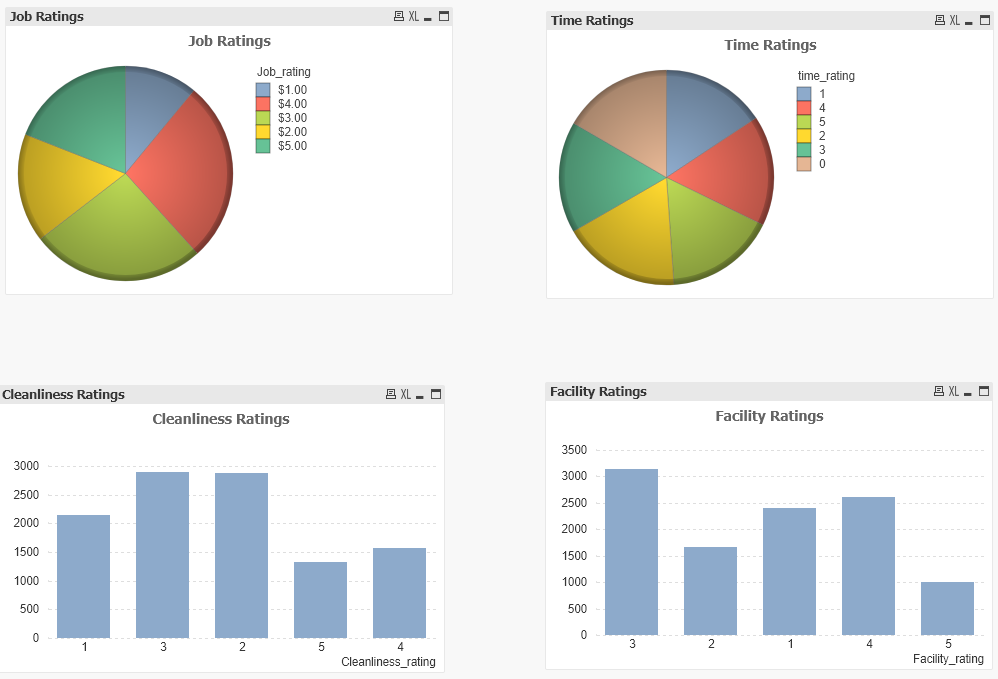


This screenshot shows an example Workflow for our Qvx Writer. Data is loaded from an Excel file, and this generated data table is passed to our Writer.



In the “Settings Panel”, the user chooses the output location. The file overwrite policy (bottom of panel) is set to “Overwrite”, so when the node is executed, the file will be overwritten; if the overwrite policy is “Abort”, there would be an error message (because the file already exists).

In the “Field Attributes Panel”, the user selects how the data will be stored in the qvx file, and sets the number of decimal places (which is applicable to certain types of number formats, and ignored for other formats). Our node makes a decision about how the data will be formatted in the qvx file (based on the Knime data type of each column), but the user can overwrite this default qvx format by using drop-down menus to select an appropriate format.

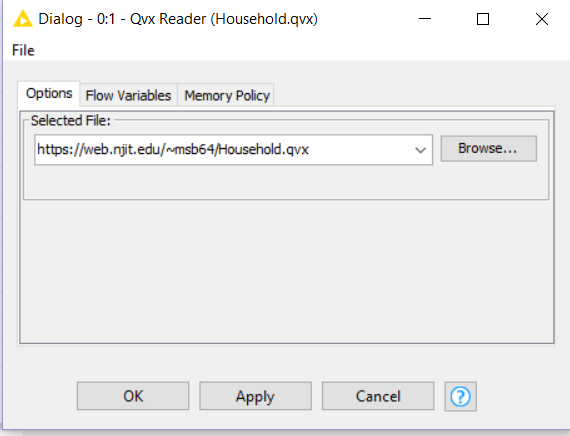


This is an example of the QlikView data visualizations that can be done based on the qvx files that are generated by our Qvx Writer.

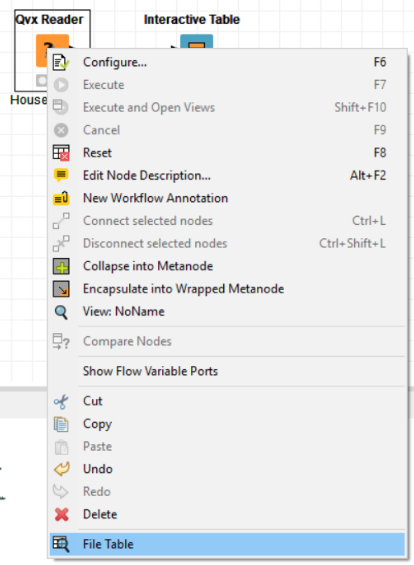
**Chapter 5: User Manual**

Qvx Reader:

In the settings panel, choose a file location to read from. This file can either be loaded from the local file system or the internet.



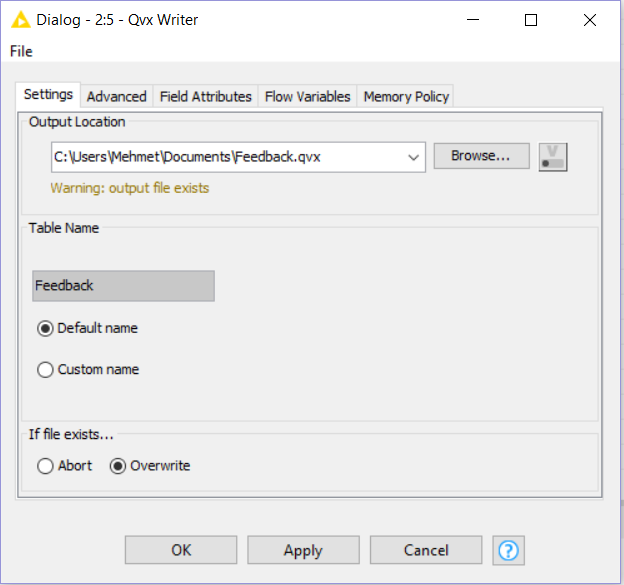
Execute the node. To see the data table that was generated by the node, right click the node, then choose “File Table”. This file table can be passed to the next node in the Workflow.



Qvx Writer:

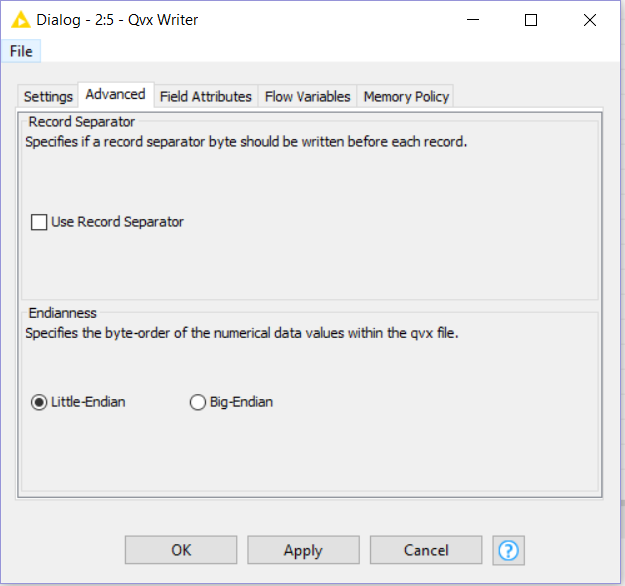
In the “Settings” panel, there are three options:

* Output Location – The user should enter a valid file name
* Table Name - Table name of the Qvx file. By default, it is set to have the same name as the output location.
* If file exists – Choosing “Abort” will cause the node to fail on execution, in order to prevent accidental overwriting of an existing file. This option is checked by default. If “Overwrite” is selected, the existing file will be overwritten.



In the “Advanced” panel, there are two options:

* Record Separator - Specifies if a record separator byte should be written before each record. By default, there is no record separator used.
* Endianness - Specifies the byte-order of the numerical data values within the Qvx file. By default, little-endian is used.

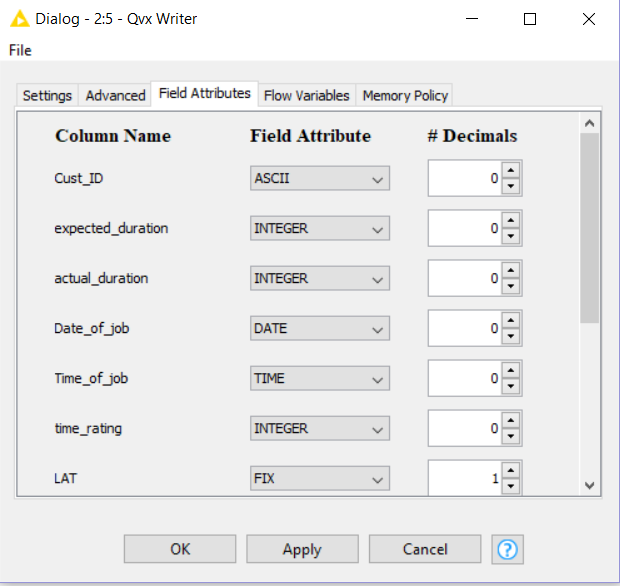


In the “Field Attributes” panel, each column has its own dropdown menu that specifies how that column will be stored.

The numerical types are REAL, FIX, INTEGER, and MONEY. For REAL and FIX, the “# of Decimals” field specifies how many decimal places will be stored in the qvx file.

The String types are ASCII and UNKNOWN

The Date-Time types are DATE, TIME, INTERVAL, and TIMESTAMP. INTERVAl is the same thing as TIME. TIMESTAMP includes both the date and the time.

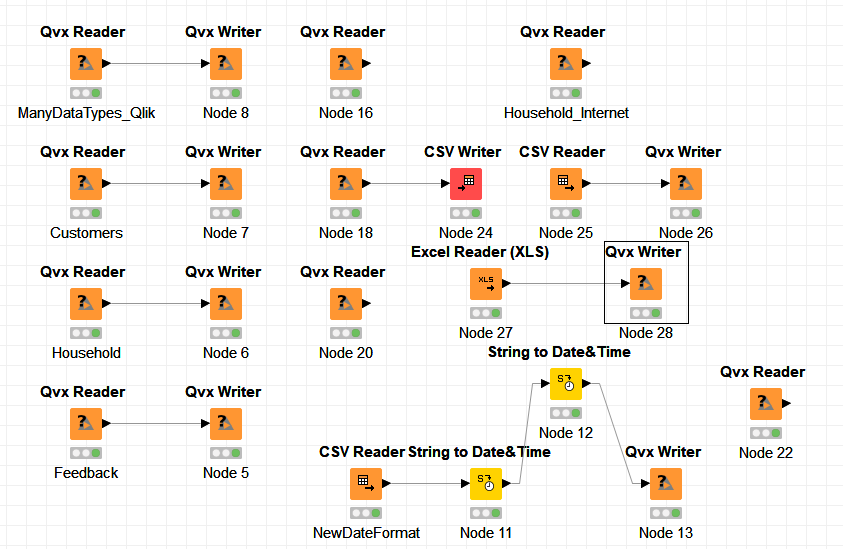


**Chapter 6: Evaluation and Conclusion**

Our final product met the sponsors’ requirements, which were to create a Qvx Reader node and a Qvx Writer node. Our sponsor had specific requirements about the qvx and Knime data types that needed to be handled in the project, and we met these requirements. The final project can be found at: <https://github.com/monicasangam/capstone-project>

Our approach to testing the project was based on incremental testing. The first step was to write code that converts String arrays to data tables and vice versa; by doing this, we verified that our steps for reading and writing tables were valid. Then, we worked on the user interface; we made sure that the node settings were being saved/loaded correctly. Next, we made sure that the qvx files were read/written correctly (without worrying about the specific qvx data types and Knime data types). Once that part was working, we fine-tuned our project to make the data formatting exactly as the sponsors specified

The sponsors gave us various test files. We tested them from our local machine, and we also uploaded them to our public NJIT webpages, so that we can test the reading of files from the internet. We developed a suite of tests which included IO of many different qvx files. We created a “Final Testing” Knime Workflow in order to run this test suite. We had to make sure various combinations of Knime nodes worked, including “Qvx Reader -> Csv Writer”, “Csv Reader -> Qvx Writer”, “Qvx Reader (from Internet file) -> Csv Writer”, “Excel Reader -> Qvx Writer”, “Csv Reader -> Create Date Time Field -> Qvx Writer”, etc (“->” is shorthand for “connected to”). Knime supports multiple different date-time formats, so we had to make sure that all of them were supported by our Qvx nodes. Whenever we made an update to our product (towards the end of the semester), we ran the “Final Testing” Workflow to make sure nothing got ruined by the update. Our “Final Testing” Workflow is shown here.



For the Qvx Writer, to test that the qvx files were saved with the specified Field Attributes, we loaded them into QlikView. We came up with a specific test case. We saved one of the column values as money and two of them as fix point decimals with exactly one decimal place. There were three date-time columns in the dataset (one date, one time, and one timestamp). We did three tests on these columns: convert all of them to date, convert all of them to time, and convert all of them to timestamp. This ensured that date-time could successfully be converted from one form to another. We also had to make sure that both of our nodes could correctly handle null values (in other words, for the Writer, put a null flag into the qvx file, and for the Reader, put a Missing Cell into the Knime data table).

For the Qvx Reader, we had to learn how to generate qvx files in QlikView. We first used a load query to load an Excel file, then we used a store query to store the table in a qvx file.

Analytiq will be selling our final product to Knime. Knime nodes are packaged in .jar files, so we created one .jar file for the Qvx Reader and one .jar file for the Qvx Writer. Currently, if someone has our .jar files, they can add both of our nodes to their Knime installation by dragging and dropping them into the appropriate folder on their machine. Once Knime has reviewed our project, they will integrate the .jars into a future Knime release, so that the Knime user has immediate access to our nodes.

We got a deeper understanding of Java by doing this project. We learned a completely new Knime API. There were various Java technologies that we picked up, such as JAXB and Maven. For us to get started on the project, we had to read official documentation about the qvx file format, so we got better at technical reading. The public qvx file specification has many different valid formats, so we had to carefully plan our code so that it would generalize to each one of these formats. Prior to this project, we had minimal experience with the entire software development cycle (including the packaging steps), so with the help of our Analytiq sponsor, we were able to learn how this process works.

Aside from specific technologies, we improved our ability to interact with stakeholders. From April onwards, we had weekly meetings with our sponsor at Analytiq. Throughout the entire semester, we met with Professor Vaish (who introduced us to this project) at least 3 times per month. At these meetings, the sponsors gave us advice about what needed to be done next as well as suggestions for things that we needed to change. Monica and I met with each other frequently as well, and we learned to better communicate with each other and help each other out with areas of struggle.

There are some things that we would change if we had to do this project all over again. Our code that does the qvx file parsing could possibly be made more clear by using inheritance (in order to eliminate some of our nested if statements). From the beginning, we should have had more of a focus on proper documentation; for various parts of the project, documentation was not added until mid/end of the semester. Regarding the “Final Testing” Workflow mentioned earlier, we should have created this Workflow earlier on so that we could have adopted a more test-driven approach to this problem, which would have helped us accurately gauge what was done and what needed to be done throughout the entire project.