Puneet Sandher (1174249)

Cobol Statistical Measure Reflection

This is a reflection of re-engineering a statistical Cobol program to improve functionality and remove legacy and archaic features. This process had many challenges, difficulties running an unfinished program, difficulty understanding the program, and reflecting on re-engineering larger programs.

The biggest challenge in re-engineering a program is understanding the logic. As a novice Cobol programmer, I was still learning the legacy and archaic features and understanding the code. This was a big learning curve to overcome to understand the logic. The logic itself was over-complicated and had many unnecessary loops and poorly written conditional statements, which took longer to understand. Moreover, attempting to re-engineer a component of the program without understanding it, would cause the program to crash as many poor structures were interwoven together. For example, the original program had many unnecessary loops such as multiple loops to calculate the standard deviation and mean. Understanding the logic and re–engineering this looping structure was difficult, as well as, it's difficult to make small incremental changes that did not cause the program to crash. I had to make large changes to the looping structures without running the program, which led to many errors and warnings to handle all at once. I adjusted the code so that instead, there was one loop that collected all the data in an array and any calculations would be done by iterating through the array. This eliminated the complex and repetitive loops of file reading, which significantly improved the program's structure. One complexity of the loops to read a file is that it had a terminator, which was removed, and now a file can iterate until the last data point as outlined in the program.

It was difficult getting unfinished programming running. The initial programming had many errors and warnings that needed to be resolved which required going over the program line by line. Compiler error messages are not always informative and require further understanding of the code. For example, there was an error for statmold.cob of a loop that goes until 999999.99, which at first glance makes it difficult to understand the error with the limited information from the compiler. A significant time was dedicated to understanding the program and identifying that the problem was that the loop could be a maximum of 999999.98. Moreover, after getting the program running without errors, I faced logical errors specifically with opening the files. The program's functionality does not work without the file opening, which requires spending time reading documentation and understanding the code. The code could have been having issues due to the changes I made or original changes in the program, it took a significant amount of time to narrow down the origin of the logic error. Overall, running unfinished code was difficult because it required a significant amount of time to understand the program, read the documentation and many logical errors to achieve a functional program.

The most challenging aspect of the Cobol program was understanding how the mean and standard deviation were calculated. There were many loops involved in reading the files, and unnecessary paragraphs made to do one calculation (such as sum-loop), which I learned was unnecessary as I traced the program. It took a significant amount of time to understand this program's logic. I incrementally separated each statistic calculation into one paragraph. It was difficult to understand because of minimal and poorly written paragraphs, which resulted in long paragraphs of code to read. To improve the readability of the program, each operation in the program has its paragraph that will be helpful for any future improvements and iterations of code. It was difficult to understand the input-loop paragraph as it had two functionalities, writing the data point to the output, as well as, calculating the sum of the dataset for the mean. This was a poorly written paragraph as it had two different operations, which is difficult for programmers to identify and understand which is why the change was made. Modularity will make

reengineering faster as the code is less likely to crash because paragraphs are less dependent on each other and it's easier to understand.

The re-engineering of the statistical measure program was not too complex because the program was short, however, a significant amount of time was spent understanding each line of the code. As a novice Cobol programmer, it may have been faster and easier to write the program from scratch because it was a shorter program, but it still had many issues that needed to be resolved. Re-engineering a large 10,000-line program would be extremely difficult as there are many intertwined pieces, which makes it difficult to incrementally improve as it can break the entire program. Moreover, a programmer will need to make detailed comments throughout the program to understand what is happening, as well as, notes of specific changes that need to be made and line numbers of the dependencies that must be adjusted to hold the program's integrity. The complexity and interdependencies of re-engineering a large program is a challenging and time-consuming task, requiring expertise and attention to detail. Moreover, a thorough knowledge of Cobol and its legacy features to understand why certain decisions were made in the original design, potential issues of the design and how to improve it. In addition, a large program requires a comprehensive testing strategy to ensure all program functionalities are successful and the program's integrity is intact. Re-engineering a smaller program made it easier to separate each calculation into a separate paragraph, update inaccurate loops and remove legacy features. A larger program requires additional attention to detail, or it may be best to divide larger programs into sections to reduce the complexity.