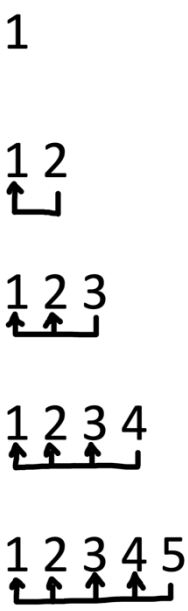
Software Engineering Concepts COMP3003 Assignment 1 Report

By George Aziz – 19765453

Chapter 1: Design

The implementation design of my overall program consists of three classes that are written in Java. One class contains the main program that is responsible to start the user interface (U.I) class where the second class is the user interface class itself that handles all button actions and the setting up of the graphical user interface (G.U.I), and finally the third class handles my multithreading solution to find files and calculate similarities between them as well as starting and stopping the threads. The GUI is written in JavaFX and is only responsible for the user interface where everything else related to threads is handled by the FileChecker class.

The design of my multithreading implementation is made up of one main producer thread and one main consumer thread. The relationship between the threads is currently a one-to-one relationship with the consumer thread executing tasks to a thread pool and the producer communicating with the consumer through a blocking queue. The producer thread is responsible for going through the user specified directory tree from the U.I and inputting all found nonempty files that end with a $.txt$, $.md$, $.java$, $.cs$ or $.csv$ extension into an array blocking queue of size 50 while also incrementing a job counter that will be later used by the consumers to check the progress of the entire program. The blocking queue is used by the main consumer thread to be able to see the next file name and continue with the comparisons. For each new file from the queue, it will be taken out and added to a list that is only used by the main consumer thread. This list is what is mainly used for the comparisons since for any new file that gets taken from the queue, it will then be compared to all other files within the list. Additionally, using the list as can be seen in the image to the right, ensures that each file only gets compared once to every other file as the specification requires which means each file gets compared $n - 1$ times since it doesn't get compared to itself. As of this point there has been no need for any form of synchronization due to there being only one of each thread total interacting with each other.



After the file is taken out of the queue and added to the list within the main consumer thread, a for loop will begin where each iteration of the loop will be executed in a thread pool. Within each iteration, a check will first occur to make sure the comparison isn't being made on the same file by checking the name of the two files. Once the check succeeds, the two file name strings are used to get the bytes of each file which then gets passed onto the LCS algorithm, provided to us where the similarity value of the two files is calculated. Once the similarity value is obtained, a new ComparisonResult object is made with the two file names and the

similarity value which is then used for the result output which is file writing and/or GUI output. After the result has been outputted, the current processed jobs count increments as well and then a progressChecker() method is called which calculates the current progress of the program, updates the progress bar on the GUI and finally does a final check to see if the program has completed by using the current progress value to see if it is equal to 1, which is calculated by using the job counter from the producer thread as well as another current processed job count. The only other way the program ends is if an interrupted exception is caught in either threads which then calls then endThreads() function that is used by every other approach to end. Both the result output into file and/or GUI, processed job counter increment and progress checks are done within a synchronization block on a mutex object since without a synchronization block there would be race conditions such as the incrementing of the count not being done properly as it is a shared resource among the thread-pool and so the program will never end. Furthermore, without a synchronization block file writing may result in issues due to multiple threads trying to write at the same time and for the GUI output, the GUI becomes a lot less responsive as there is a thread pool that constantly gets the GUI thread congested.

Chapter 2: Scalability

In a scenario that would require this program to be used for large number of files and by multiple people, non-functional requirements such as the program must be able to handle large files, the program must be able to handle large directories with many files, the program must be fast such that 300 files would finish at a maximum of 2 minutes and the program must be able to accurately compare any file to another file, are needed to ultimately have a scalable program.

The first requirement is that the program must be able to handle large files and that is a requirement that requires a bit of testing and debugging to identify the areas responsible in fulfilling it. The current program unfortunately is not able to handle large files or even small files. During my testing, the program was unable to handle large files or else the JVM crashes since it runs out of heap memory. Initially, I thought this issue had something to do with my implementation of multi-threading which led me to experimenting with different approaches and checking the memory usage of the program. Almost every different approach I took, led to the same result that the program crashes during its execution on large files. The program was only able to handle a maximum of 16KB on my laptop for the files or else JVM crashes with a “OutOfMemoryError : Java heap space” error. The issue was coming from the consumer thread, within the thread pools and after further testing and debugging, the core issue appears to be the LCS algorithm which is used to calculate the similarity value between files and more specifically the issue is when we first create the arrays that will be used within the algorithm. If two files being compared are two large files that are 150KB in size, then that means the sub-solutions array would be $150\text{KB} \times 150\text{KB} = 22\,500\,000\,000\ \text{bytes}^2$ in size and that is only for one array out of all the other comparisons which in my test case scenario was out of 300 total files, that is 44850 total comparisons (This number excludes same file checks). Therefore, the first issue and solution to making this program more scalable would be to improve the similarity check algorithm by reducing the sizes of the array or finding an alternative way that is more

effective and efficient to perform comparisons which not only would allow the program to handle large files but also provide better performance and help fulfill the requirement that the program must be able to handle large directories with many files.

Another requirement mentioned, is for the program ultimately to be fast. To fulfill this requirement, there are two main areas for speed up – the producer and the consumer. For the producer, one approach that can help improve the performance would be to submit each sub-directory into its own thread-pool which would process more files concurrently and provide them to the blocking queue. As a result, this also helps fulfill the requirement that the program must be able to handle large directories with many files. Regarding the consumer, one approach is ensuring the program itself is efficient which would mean to change the similarity check algorithm as previously mentioned such that it doesn't use as many resources and also provide better performance since that is the bottleneck of the program. Additionally, another approach for the consumer is to use multi-threading even further. Currently, there is only one thread pool being used for the consumer thread and that is for the execution of the similarity check and result output to file and/or GUI. The current performance can be seen in Figure 1 below, where there is a benefit to having that thread-pool used for the consumers as it does speed up the overall program execution. but having more than 5 threads in the pool becomes too expensive for little to no performance boost at the expense of more resources. Therefore, simply having more threads in the consumer thread pool won't suffice. The solution to this problem would be by having an additional thread pool being used for the "main consumer" since currently there is a one-to-one relationship with the producer and consumer, and so even if there were many producers as previously suggested, the bottleneck would be the consumer since it would be a many-to-one relationship. To further enhance the performance, the program would have to be a many-to-many relationship and so the current "main consumer" thread would be running concurrently with multiple instances of itself and ultimately benefiting the performance of the program by having more processes running at the same time as well as again, fulfilling the requirement of being able to handle large directories with many files.

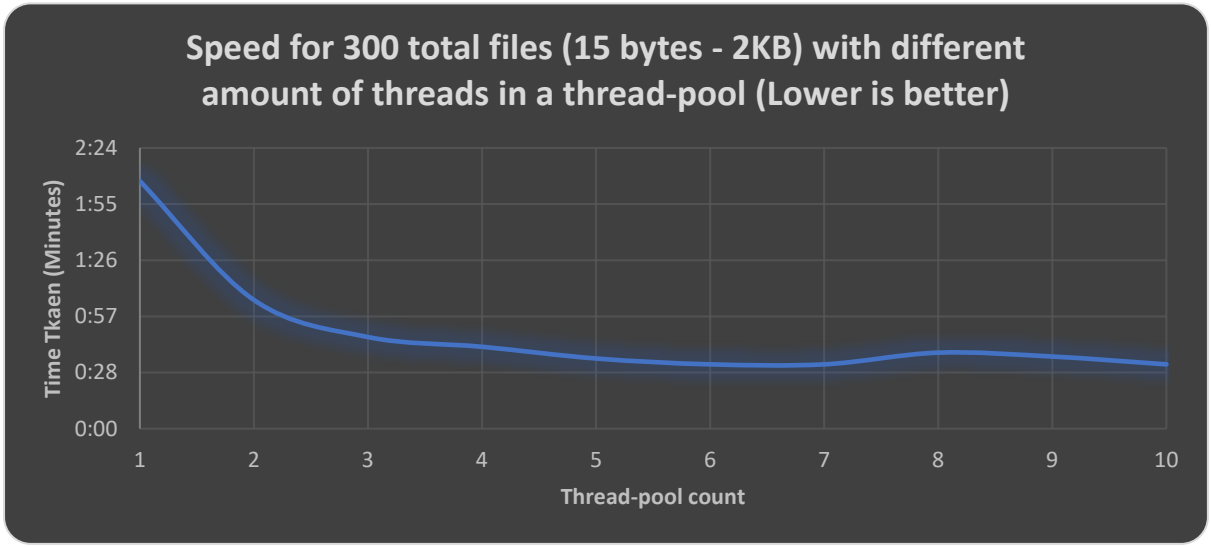


Figure 1 Test on program speed conducted on a 2018 MacBook Pro i7 2.7GHz Quad-Core with 16GB of RAM and 256GB SSD

Finally, the program must be able to accurately compare files with each other. After mentioning all the possible solutions with the other requirements for the program, it may affect the overall accuracy for the program and so it is crucial that the accuracy remains the same or improves if possible. The only way to ensure the accuracy of the program remains the same or improves would be to continuously test the program for each new modification to the program and possibly create automated testing to help with the process. As for the program's architecture, the main area which is responsible for the accuracy of comparisons is in the similarityCheck() method and so it is important that any changes made to it, doesn't reduce the accuracy of the comparisons.