Assignment 1 (Report)

Design Explanation in Regards to Multithreading

Thread Responsibility and Purpose

The classes responsible for creating threads are UserInterface and Comparator.

UserInterface UserInterface creates a single thread, `comparator`. The purpose of `comparator` is to implement the core application logic.

Comparator Comparator creates a variety of different threads and additionally, a couple of executors (which will be discussed shortly). The created threads are: `searchThread`, `extractThread`, `compareThread` and `outputThread` and `interruptThread`.

The purpose of `searchThread` is to execute a runnable, `searchRunnable`. The thread executes this runnable to search the user-specified directory for any and all non-empty files to add to a `List`, named `files` for later content-extraction. Seeing as this is the only thread searching this directory, there is no need to use a `BlockingQueue` or any other concurrent data structure. Note, on macOS devices, this thread will also not add .DS_Store files for extraction.

`extractThread` also executes its own runnable, `extractRunnable`. This runnable is executed to extract the contents of all previously retrieved files to enqueue to another `BlockingQueue`, named `contentsQueue` for later content-comparison. Since content-extraction can be a fairly resource-intensive process (especially in regards to directories with many files), this thread makes use of an `ExecutorService`, named `extractionExecutorService`, to have the contents of each file extracted in its own thread, improving overall performance.

The next thread, `compareThread`, executes a `compareRunnable`. `compareThread` executes this runnable to compare the contents of each file using a method, `computeSimilarity()`. The results of each comparison is enqueued into yet another `BlockingQueue`, named `resultsQueue` and the progress bar of the GUI is updated upon each comparison. content-comparison can also be a resource-intensive process and thus, this thread makes use of `ExecutorService`, named `comparisonExecutorService`, to have each comparison (and its associated actions) executed in its own thread, further improving performance.

The purpose of `outputThread` is to execute the `outputRunnable`. The thread executes this runnable to have high-similarity file pairs added to the GUI. Additionally, this thread also saves all results to a file named `results.csv`, saved in the directory the user chose to have analysed.

Lastly, this class also creates `interruptThread`, which executes the `interruptRunnable`. The thread executes this runnable when either the “Stop” button or the “Exit” button is clicked. In this runnable, the thread interrupts each of the aforementioned threads to have them halt execution and

terminate and then immediately shuts down both `ExecutorService` instances, cleaning up before terminating comparison entirely.

Thread Communication

The main thread communicates with `comparator` through starting the thread when the user specifies that they would like to run a file comparison and also when they click the "Stop" button to specify that they would like to abort a comparison.

Each of the threads in `comparator` communicate using `Thread.join()`. Essentially, the core logic flows is as follows:

1. Enqueue non-empty files for content-extraction
2. Extract the contents of these files
3. Compare the contents of these files
4. Output the results to the GUI and a file

Therefore, `outputThread` joins `compareThread`, `compareThread` joins `extractThread` and `extractThread` joins `searchThread`. Once `searchThread` dies, `extractThread` begins execution; once `extractThread` dies, `compareThread` begins... so on and so forth. All while this is happening, the main thread is suspended so that comparison can be aborted the moment the user requests it by invoking `comparator.interrupt()`, which then executes its clean up procedure.

Resource Sharing

Each of the aforementioned threads share resources without incurring race conditions or deadlocks by making use of multiple `BlockingQueue` instances and synchronised `Thread.join()` invocations.

`BlockingQueue` Since there are multiple `ExecutorService` instances in use, this thread-safe data structure is required when extracting the contents of each file.

Recall that extraction for each file is to be done on its own thread and then enqueued for later comparison. Therefore, a `BlockingQueue`, named `contentsQueue` is required to store the contents of each file in a thread-safe manner.

Additionally, each comparison is also to be done on its own thread and then enqueued for later output to the GUI and file. Likewise, a `BlockingQueue`, named `resultsQueue`, is used to store the results of each comparison in a thread-safe manner.

`Thread.join()` Since multiple threads are going to require access to the same data structure (e.g. `extractThread` reading from `files`, which, needs be populated by the preceding `searchThread`), `Thread.join()` is used to prevent

premature accessing of shared data structures to ensure comparison is conducted in the expected manner (i.e. `compareThread` doesn't try and compare files from an empty `contentsQueue` because it hasn't yet been populated).

Thread Termination

The threads of real concern are `comparator` and all of its associated threads; `searchThread`, `extractThread`, `compareThread`, `outputThread` and `interruptThread`. Additionally, all of the threads created by `comparisonExecutorService` and `extractionExecutorService` are also considerable.

`searchThread`, `extractThread`, `compareThread` and `outputThread` all terminate when either "Stop" or window close buttons are clicked. Upon these actions, `comparator.interrupt()` is invoked, wherein, `interruptThread` executes `interruptRunnable`, causing the interruption and prompt termination of each of these threads. `extractionExecutorService` and `comparisonExecutorService` are also both shutdown immediately. Upon terminating all threads and shutting down all `ExecutorService` instances, `interruptThread` terminates, completing the termination process.

Scalability Considerations

Potential Problems Associated with Large Inputs

The most notable problems associated with large inputs are: 1. Drastically increased computation time 2. Drastically increased memory usage

Pertinent Non-Functional Requirements

With the aforementioned problems in mind, Let's consider two important Non-Functional Requirements: 1. The application must execute a full comparison procedure involving 300 files (from file-retrieval to output) within 5 seconds of receiving the user-specified directory. (To be evaluated on a 2021-model MacBook Air) 2. The application must use no more than 500 MB of RAM at any given moment (To be evaluated on a 2021-model MacBook Air)

Re-engineering Architecture

In regards to addressing both computation time and memory usage increases, the most apparent changes to be made would be to `Helpers.extractContents()` and `Helpers.computeSimilarity()`.

As to why these methods are the cause for concern, extracting the individual characters of a file is always a relatively time-consuming process. Moreover, the extraction process also involves a storage process. In our case, storage in the form of a contiguous data structure... to store each individual character... for

each and every file. Unsurpisingly, when attempting to run the `comparator` on directories with a large number of files, `OutOfMemoryError` would occur.

Additionally, the comparison process will also be incredibly time consuming as we are not simply comparing strings for similarity, but entire files.

So, the proposed solution would involve computing and comparing the hash values of each pair of files, using MD5 in Java for example. This will allow us to decrease computation time as hashing is generally a very fast means of comparing files. Moreover, it will also allow us to lower memory usage as the process of file hashing does not involve the reading of characters (or words, for that matter) individually.

Potential Problems Associated with Re-engineered Architecture

Unlike simply extracting individual characters from a file, hashing a file may be a relatively complex procedure, given the variety of different implementations–each with their own advantages and disadvantages. Moreover, long-term maintenance may also become an issue as with increased code complexity, readability and understandability may suffer. However, given that we desire highly-scalable code, this undertaking will be quite necessary.