What is a MOSS?

A Measure of Software Similarity is a measure which gives a quantitative value to the close of source code to one another.

In here is an implementation of a MOSS from professors at Stanford. It describes what a MOSS can precisely used for and what it must not be used for, at least without some additional checking.

<https://theory.stanford.edu/~aiken/moss/>

Experiences on using MOSS:

<https://www3.nd.edu/~kwb/nsf-ufe/1110.pdf>

Here is an implementation of MOSS that uses winnowing:

<http://theory.stanford.edu/~aiken/publications/papers/sigmod03.pdf>

Link to a central class in my implementation, the StreamTokenizer which separates Reader objects to common programming tokens:

<https://docs.oracle.com/javase/7/docs/api/java/io/StreamTokenizer.html>

A website which shows some key algorithms for MOSS software:

<https://web.wpi.edu/Pubs/E-project/Available/E-project-043015-122310/unrestricted/CheckSims.pdf>

There are some straightforward assessments for MOSS (which I can possibly think of):

* The first four is based on the very-popular **confusion matrix** often used in CS, AI and medicine research. The matrix is basically divided into four windows: true positives, true negatives, false negatives, and false positives. It is clear however that the fact that the concept of MOSS even exists conceptually means that determing the “positivity” of code plagiarism is an impossible task. However, the best way we might go about this is the intentionality we bake into our code. If we deliberately try to obsfucate a copy of source code and it fails to detect this, we might as well call this a false negative. If we just feed it innocent and non-plagiarized code and it says there was plagiarism, we might as well say this is a false positive. Now that it is somewhat established what each item in the matrix even means to us, we might be ready to use it as a criterion for accuracy.

To open all files in a directory, the **Files.walk** API is available in Java 8+, which means I might be able to use that to implement the Multi-Pair Comparisons.

<https://stackoverflow.com/questions/1844688/how-to-read-all-files-in-a-folder-from-java>

To load multiple files in say, a BufferedReader, I must first load each one in their own **FileInputStream**, and combine them via a **SequenceInputStream** object. This can be placed into a **BufferedReader** or an **InputStreamReader,** along with some charset specification (e.g. UTF-8), which can then be loaded into the aforementioned reader (for efficiency). Since this is now a Reader, this can just be passed into the **algorithm** package.

FOR NOW, I WILL FOCUS ON PROJECT PAIR COMPARISON.

I am currently thinking about the **projectpairmachine** package. Its main role, as I see fit, is to take a collection of Readers and combine them into a single **BufferedReader,** which will then be compare each readers’ content via the **algorithm** package. This clearly poses a bunch of problems I can think of now, each of which I must address before I start writing it:

1. **External Reader handling**: it is presumed here that the *Reader*s that will be loaded into the class can no longer be touched by the caller or any other section of the code that might have access to it, **unless the design of a SequenceInputStream** allows this to be done. It is hard to trust outside classes for this naturally. However, as I type this, I realize that as long as this class is package-private, this behavior can very well be limited by the package author.
2. **Inefficient Reloading of Files:** I would suspect that for this specific use case, where each file is reloaded N times (N being the number of files used), it is important that we maintain some level of buffering along the way. As such, we have motivation, at least if we want to load an arbitrary number of projects into a byte array, which we can recycle (since it is immutable)

**GUI Plans**

Best practices for JavaFX

<http://accelconf.web.cern.ch/accelconf/icalepcs2017/papers/thapl02.pdf>

<https://www.oracle.com/technical-resources/articles/java/javafx-productivity.html>

FXML Tutorial

<https://docs.oracle.com/javafx/2/get_started/fxml_tutorial.htm>

NOTE: A Problem with GLOB filters. It would seem that normal glob syntax doesn’t work. For complex paths, a double wildcard is required in front (e.g. \*\*/\*.cpp) which is problematic at best.