LopezMOSS: An Implementation of Measure of Software Similarity Algorithms

\*Note: Sub-titles are not captured in Xplore and should not be used

Emil John Lopez  
De La Salle UniversityBataan, Philippines  
emil\_lopez@dlsu.edu.ph

*Abstract*—This electronic document is a “live” template and already defines the components of your paper [title, text, heads, etc.] in its style sheet. *\*CRITICAL: Do Not Use Symbols, Special Characters, Footnotes, or Math in Paper Title or Abstract*. (*Abstract*)

Keywords—component, formatting, style, styling, insert (key words)

# Introduction

Plagiarism is an age-old problem in intellectual property law. However, the speed of data transfer and the rise of the Web has made it easier to do the act and has made detection much harder. Furthermore, adjuging someone of commiting the act turns out to be very complicated and can lead the overseer and lawyers in a legal grey area, where it is unsure whether the act violates the law. It is important to examine its definition first to attempt to solve the problem. According to a paper, actions that fall under plagiarism include: (1) turning in someone else’s work, (2) copying someone’s idea without giving credit, (3) not putting quotation marks, and (4) changing words only without changing the structure of the sentence [1].

In educational institutions, software assignments are usually subject to this form of malpractice and so it threatens the integrity of the educational process. In particular, it is subject to the first and second definitions stated above. However, given the number of students each professor must handle, and the large amount of effort required to perform ad hoc comparisons between them, there is a need for a tool that reliably detects plagiarism and can look past obsfucation, reordering, refactoring and other methods of deception [2].

The first significant solution to this problem was introduced in 1994 by an associated professor in UC Berkeley. It used *winnowing,* a local document fingerprinting algorithm that grouped grammatical tokens in groups of some number of tokens, hashed them to minimize storage space, and counted the frequency of each group using their hash value. This is directly derived from other techniques such as Karp-Rabin String Matching [3].

Fingerprinting algorithms calculate numbers (which we call fingerprints) to help identify a document. Conflicts within these numbers usually indicate that some part of a document (or segment of code in this case) is similar to another document. Better fingerprinting algorithms have since been derived from this method, but it remains that the foundation of most of them is either Karp-Rabin String Matching or *n*-gram fingerprinting [4].

This paper will present an implementation of a variation of these key fingerprinting algorithms. In particular, it showcases a version of *n-*gram fingerprinting. The implementation will also have a *graphical user interface* (GUI) that will allow its user to simply pick directories which correspond to projects.

# RELATED WORKS

There have been many attempts at solving the code plagiarism problem since its conception in 1994. The following contains a list of all such services, including some implementation details in their design.

Most of these solutions have been found to fall under one of these three (3) categories: (1) text-vased, which uses plain text as its input, (2) token-based, which uses tokens, fundamental grammatrical units, and (3) model-based, which creates different models for the source code. Qualitatively, it might be accurate to say that the last two algorithms, which uses organized data structures, will be slower for larger dataset, which might prove problematic. However, for accuracy and efficiency, the third is most likely the best for large scale anti-plagiarism detection [5].

One such solution uses the concept of a *software birthmark,* a distinct characterestic of machine code. It involves the use of a type of birthmark called Dynamic Key Instruction Sequence, which can be directly extracted from compiled machine code. This means ordinary source code obsfucation does little to nothing to help evade detection because this will rely instead on a graph data structure built from the machine code of the program. At a lower level, it can even detect compiler optimizations and other tooling that might differentiate two similar pieces of code after compilation. [6].

Another solution, which takes the concept of n-gram fingerprinting and other token-based algorithms one step further is the Weight Abstract Syntax Tree Kernel method, which creates a syntax tree from the source code (like a compiler would) and the tree of two pieces of code to calculate the similarity between the. This approach works better than more popular implementations such as Sim and JPlag because it takes the actual grammatical structure of the tokens into account instead of simply just comparing them as independent entities. [7]

JPlag is a web service that attempted to find similarities amongst a group of programs written in Java, Scheme, C, or C++. It processes code in two phases: (1) the programs are parsed and are effective turned into tokenized strings, then (2) it uses a method called **Greedy String Tiling**, where it tries to match substrings of each token strings to each other as much as possible [8].

Another such tool is *Parikshak* [9], which is similar to Jplag in that it uses the **Running-Kark-Rabin Greedy String Tiling** algorithm. It breaks down the program into tokens, then proceeds to group each token of size n, aptly called *n-gram representation.* It then converts these groups into corresponding numbers (each digit of which represents the type of the token). To generate a similarity score from such comparisons, it uses the **Jaccard Similarity Coefficient**, the quotient of the total number of fragments that appear in both programs and the number of fragments different between the two programs.

# METHODOLOGY

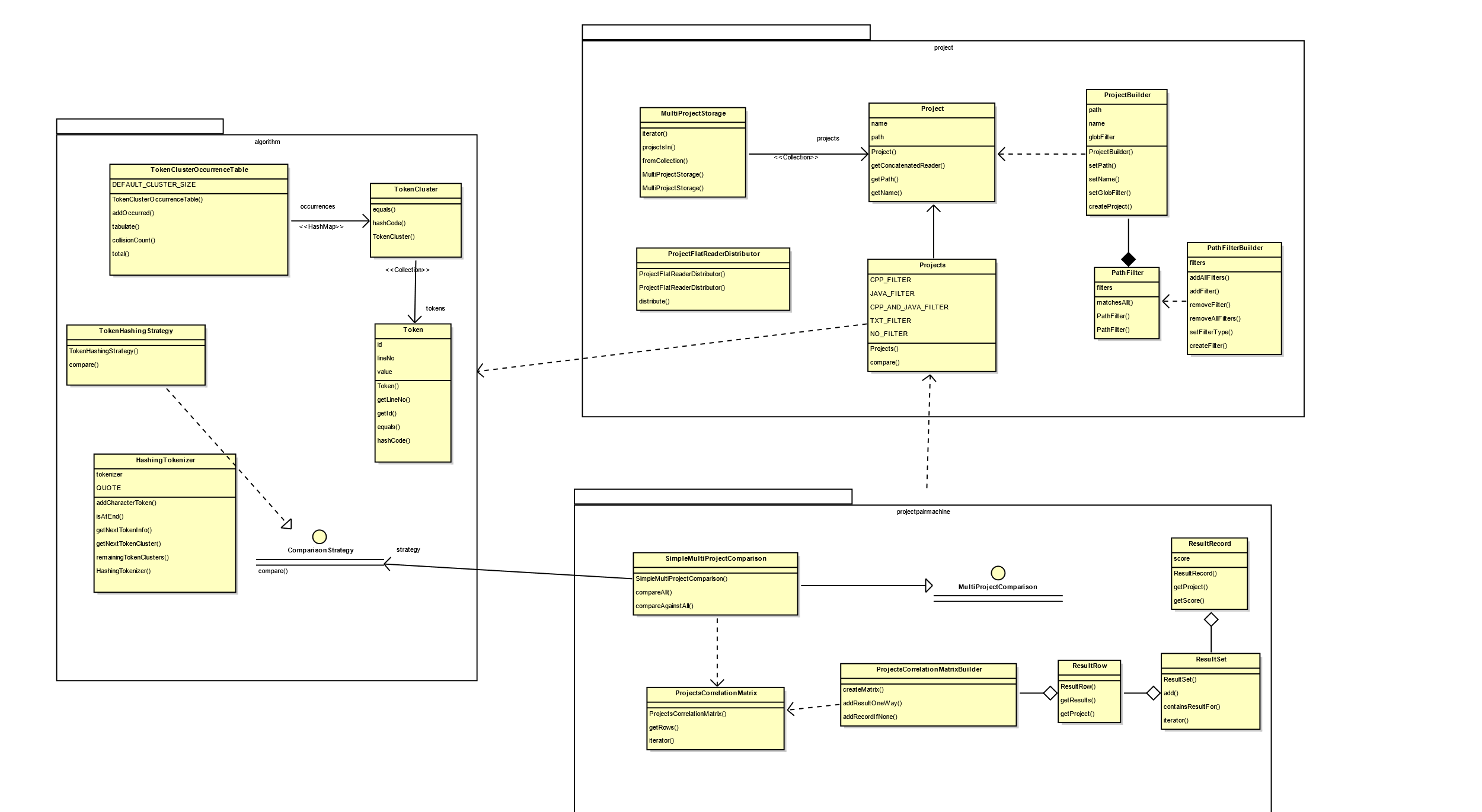


Fig. 1. Class Diagram of Entire System

## Overall Structure

The entire system is divided into four different parts: the algorithm, the project organizer, the project pairing system, and the GUI. The overall relationship between the parts are shown in the class diagram above. The next few sections shall be dedicated to providing exposition of how each subsystem works individually and together.

## Project Organizer

The project organizer is responsible for all tasks that involve the projects to be used by the system. This naturally includes loading (of project and streams), filtering, concatenating (loading of files into memory), and aggregating them. All the relevant code for this section can be found in the **project** package of LopezMOSS.

### Loading and Filtering

The user of the **project** package can choose to load any arbitrary directory and convert it into a **Project** object. This is done by the project builder class, which takes a path and a path filter (which is how the project management system selects which files are relevant). Once both requirements for making a project was given, the builder can be made to build a **Project** object. This object will internally load all the files in the path into *file streams*.

To do this, all the files in the project had to be traversed and so, a *depth-first* traversal was done on the project path using the **Files** API’s walk method. The stream of files from this traversal was loaded into a **Stream**<**File**> object by the facility During this, the path filter object earlier was used to determine which files were relevant to the comparison by passing its filter’s *matchesAll* method to the *stream* filter method. The **PathFilter** does this by storing Strings representing a GLOB (Unix) or REGEX filter. An example of Java filtering for both would be: \*.java for GLOB and w+.java for REGEX. To check if a passed file path satisfies the filter, Java’s **PathMatcher** facility was used.

The input streams from the files here are passed to the next phase: concatenation

### Concatenation

One of the main requirements of the program is that it should be able to process a large amount of source code in a relatively small amount of time. To achieve this, all the code text had to be buffered ahead of time to allow for reusability. However, another key requirement is that they all have to be concatenated before they are processed by the algorithm.

At first, it seemed like these requirements were at odds with one another because combining them would require reading each file into a string or some buffer and then combining them together by copying the contents. The Java **String** class and byte arrays are all immutable, so multiple copies of the same content had to be loaded in to achieve this. This meant that if we relied simply on loading them unto strings, the program would suffer in terms of both time and memory consumption. It was inferred by the author that this was impractical as it severely depletes the system of all its resources.

The solution involves the manipulation of the streams loaded in the first stage. The interesting aspect about Java file streams is that they do not load the files into memory beforehand. Since it was already clear from the start that the files will only be used **together**, there was no benefit to loading them into memory separately. Thus, the file input streams of all files were loaded the **ProjectFlatReaderDistributor** using a provided project **Path.** After the input streams had successfully connected to the files, they were combined using the **SequenceInputStream,** the stream-equivalent for concatenation by the **MultiStreamReaderGenerator** class.

This combined input stream was then converted to a byte output stream and the underlying byte array was taken and stored by the class. This means that the *byte array containing the entirity of the project* was now loaded into RAM without any extra copies in memory and all of it was directly transferred from secondary memory.

Now, when an outside class requests a concatenated copy of a project folder, it could be loaded directly from the byte array by loading the generated byte array into a **ByteArrayInputStream,** where it would just be referenced internally with no harm (since the byte array was made immutable by a final declaration inside the class) and since the algorithm required a **Reader** (See Part C of Methodology for more info).

### Aggregation

In order to include multiple projects, there needed to be a storage that stores a collection of Project folders. Furthermore, it needs to be constructible using a path that contains multiple projects or a collection of paths because these are the two ways projects can be loaded into the system according to the requirements.

To prevent ambiguity between these two forms of storage construction, two static factory methods are introduced: projectsIn and fromPathCollection. The first one traverses every folder in the projects path and loads them into Project objects and the second one takes the paths from the collection and turns them into project objects. The folder traversal was done using the iterator for the **Path** object

## Algorithm

The algorithm used for this program is directly derived from Checksim’s algorithm which uses n-gram representation of tokens to compare two pieces of source code. The algorithm itself can be divided into three phases: (1) hash-tokenizing, (2) clustering, and (3) occurrence counting or tabulation. Before discussing the algorithm itself, the author will take the time to explain the interface for this part of the system. The pseudocode below

### The Interface

During the first two weeks of the project, the author could not decide which algorithm specifically they should implement. However, a few requirements were already known to them: (1) the comparison must yield a score between either 0.0 to 1.0 or 0.0 to 100.0 based on the similarity of the two input programs, and (2) it had to compare two projects or two files. By creating a generic interface which reflect these two requirements, the author kept note of the exact form the algorithm class should take. The interface they came up with was as follows:

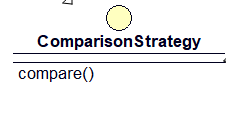


Fig. 2. UML Diagram for ComparisonStrategy

As can be seen, this class is more precisely a functional interface. The use of double as a return type makes sure that we can return a score between 0.0 to 1.0. The author recognizes that floating point representation is very imprecise and there are better numerical representations such as a large int or [class for big numbers]. However, for simplicity, double was decided in the end, especially since extreme precision was unnecessary for the purposes of the system. The method also takes in two Readers, which means all classes depending on this class must convert their input to the algorithm to reader form. This is naturally the reason for some decisions made in the project package (as discussed in Part B).

This interface has two implicit contracts (which can be found in the Javadoc): (1) the internal state of all implementing classes must be the same before and after the compare function is called to allow reusability of the same instance and (2) the output value must be between 0.0 to 1.0 regardless of input.

### Hashed tokenizer

Tokenizing is the process of extracting tokens, the fundamental grammatical units of some language. This section provides some exposition on the inner mechanisms of LopezMOSS’s internal tokenizer, which it uses to perform comparisons. It can be divided into two steps: (1) tokenizing and (2) hashing.

Tokenizing is one of the key requirements for this system because character, word, and line-based solutions have some limitations in terms of abstraction and comparison power.

Characters, when manipulated properly, might be able to provide us with the accuracy we need but it is not abstract to be processed by the developer. There have been instances of character-based comparisons in the past, all of which work efficiently and quickly but it was decided during the planning stage that it would take too much time to do this approach.

Words are a little better to deal with, but they are severely limited because delimitation does not contain enough information and such approaches can easily be obsfucated. For instance, *for(int* and *for (int*, although extremely similar to the reader of the code, will be interpreted as different by word-based approaches if whitespace is used as the delimiter.

Line-based comparisons are easily the most abstracted and fastest to build but they are extremely vulnerable to obsfucation because even single-character changes are enough to hide plagiarism.

The first approach is not considered because it would take too long and it is too tedious and the last two approaches are incredibly vulnerable because they treat programs as plain text and do not take into account that source code actually contains a lot more information than simple words or lines. In reality, code is divvied up into small units called tokens. They turn out to be the smallest we could break up a program without loss of information.

Developing a good tokenizer takes time and even if the author did decide to implement it from scratch, there are enough nuanced cases that there would not be enough time to develop it flawlessly. They decided to turn to Java’s StreamTokenizer to do Reader manipulation and tokenizing for them. This class spits out strings, line breaks, white spaces, or floating-point values that correspond to the next token from a reader. This was the central piece of the HashingTokenizer class. However, output type was varied enough that it would require a lot of processing to differentiate them. Thus, the tokenizer was made in charge of identifying the token type and then spitting out a generic Token object that could be of any type.

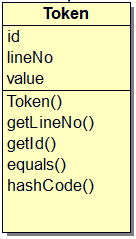


Fig. 3. Token Class UML

Given the purpose of this class, maintaining the value of the tokens was not necessary. Instead, the class only needs to maintain the identity of a token so it could be compared with other tokens. A unique ID is assigned to a token to help with this. It is assigned by taking the hash of the internal value of the token. For example, if, when hashed, yields [insert if hash here], so all instances of if will have the same value and thus will be equal. This unique ID system is done for two reasons: (1) the type of the tokens varied and converting them all to strings before storage just leads to memory bloating and (2) storing actual string tokens is costly in terms of memory and time (memory because strings generally take up more bytes that an integer hash value, and time, because comparing strings takes O(N) time depending on the string length.

The processes above are done for all tokens from the reader until the system encounters an end-of-file flag.

### Clustering

Tokens alone are prone to false positives in this particular use case. This is true for multiple reasons.

The more problematic reason is all languages and libraries have some amount of boilerplate code. For example, all C/C++-like languages have a main function declaration.

The more localized reason is that some tokens tend to appear in most source code. This can include anything from conditionals to preprocessors.

There are many solutions to this problem, such as examining structure instead of content, or analyzing machine code behavior for similar patterns. This however requires a deeper dive into compiler theory, which the author has limited knowledge on.

The simplest way to alleviate this was using an n-gram representation for tokens. This will work better than individual token comparison in principle because it lessens the boilerplate problem to an extent. This is because there are subtle differences in how boilerplate code is implemented. It will also catch some types of obfuscation. For instance, a variable declaration can be moved around but it will always be clustered to prevent breaking the program.

*//Original:*

*int a = 0;*

*int b = 5;*

*int sum = b + a;*

//*Copy*:

*int b = 5;*

*int a = 0;*

*int sum = a + b;*

Fig. 4. Example of Obfuscation

Notice that even though the code had been reordered, the grouping of the identifiers and values remain the same. This is because our potential culprit wants to preserve the behavior of the program.

Furthermore, switching around the order of the operation will also be detected because the n-gram clusters will be designed to be equal even when the tokens are reordered.

N-gram clustering was achieved by using the Tokenizer earlier to gather **Token**s. Every N iterations, the collected tokens are batched together into a TokenCluster object. This will be the object representation of the n-gram clusters. What makes the **TokenCluster** class distinct from one another (at least in the eyes of a hashmap) is the **hashCode** representation of each one. The hashes of each token are recomputed and combined to make a single hash value for the cluster. Re-ordering the tokens will have no effect on this value so obfuscation by reordering will not work.

These clusters are released via a stream by the **Tokenizer** class. The next module, the occurrence counter, will be responsible for tracking and tabulating each one and eventually finding the number of unique and similar clusters between two projects.

### Cluster Occurrence Counting

With the **hashCode** of the token cluster class in place, occurrence counting now becomes a straightforward task. When a reader was put into the **TokenClusterOccurrenceClass,** token clusters were extracted from it, then they are placed into a **HashMap** that maps each token cluster to integer pairs that indicate how many times it had appeared for both readers [projects]. From this table, the score could now be computed with the **Jaccard Similarity Coefficient** using the total number of unique token clusters and similar token clusters. In particular, this coefficient is defined as the quotient of the number of unique tokens and similar tokens.

## Testing the Program

# Acknowledgment

# References

|  |  |
| --- | --- |
| [1] | H. Maurer, F. Kappe and B. Zaka, "Plagiarism-A Survey," *Journal of Universal Computer Science,* vol. 12, no. 8, 2006. |
| [2] | K. Bowyer and L. Hall, "Experience Using "MOSS" to Detect Cheating On Programming Assignments," *Deparment of Computer Science and Engineering.* |
| [3] | S. Scheilmer, D. Wilkerson and A. Aiken, "Winnowing: Local Algorithms for Document Fingerprinting," *SIGMOD,* 2003. |
| [4] | M. Heon and D. Murvihill, "Program Similarity Detection with Checksims," *Worchester Polytechnic Institute,* 2015. |
| [5] | M. Ducarik, E. Krsak and P. Hrkut, "Current trends in source code analysis, plagiarism detection and issues of analysis big datasets," in *International scientific conference on sustainable, modern, and safe transport*, Zilina, 2017. |
| [6] | Z. Tian, Q. Zheng, T. Liu, M. Fan, E. Zhuang and Z. Yang, "Software Plagiarism Detection with Birthmarks Based on Dynamic Key Instruction Sequences," *IEEE Transactions on Software Engineering,* pp. 1217-1235, 2015. |
| [7] | D. Fu, Y. Xu, H. Yu and B. Yang, "WASTK: A Weighted Abstract Syntax Tree Kernel Method for Source Plagiarism Detection," *Scientific Programming,* vol. 2017, 2017. |
| [8] | L. Prechelt, G. Maphohl and M. Philippsen, "Finding Plagiarisms among a Set of Programs with Jplag," *Journal of Universal Computer Science,* vol. 8, no. 11, pp. 1016-1038, 2003. |
| [9] | S. Sharma, C. S. Sharma and T. Veena, "Plagiarism Detection Tool "Parikshak"," in *International Conference on Communication, Information & Computing Technology*, Mumbai, 2015. |
| [10] | M. Agrawal and D. K. Sharma, "A State of Art on Source Code Plagiarism Detection," in *International Conference on Next Generation Computing Technologies*, Dehradun, 2016. |