

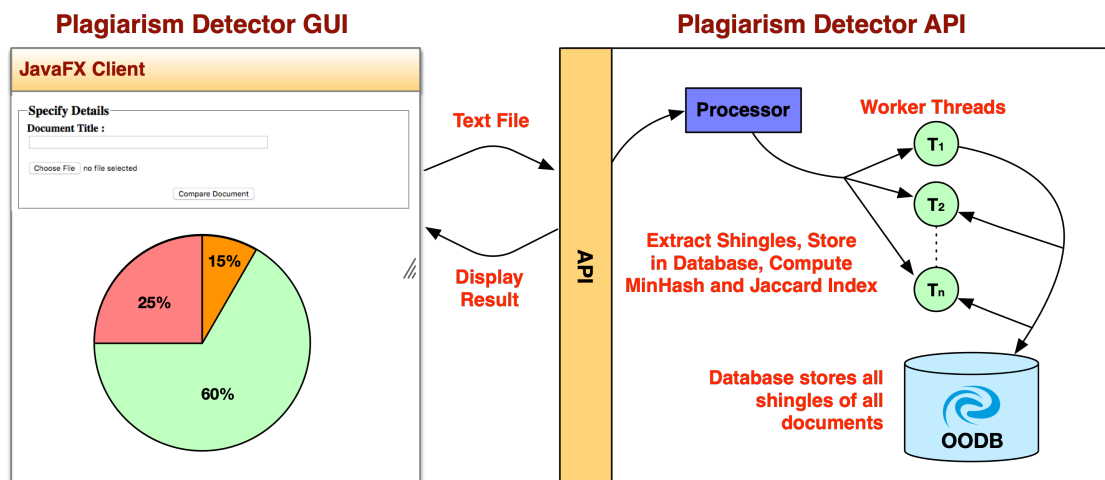
---

## Design Principles & Patterns

### *A Plagiarism Detection System*

#### 1. Overview

You are required to develop a **JavaFX** application and an **API** that enables two or more text documents to be compared for similarity. An overview of the system is given below:



A stub JavaFX application is available on Moodle to help you get started, but you are free to discard the stubs and build your own JavaFX application from scratch. You should consider using **Scene Builder**, a drag & drop rapid application development (RAD) tool from **Gluon**, to create the JavaFX client. Scene Builder is available for free from <https://gluonhq.com/products/scene-builder/>.

#### 2. Basic Requirements

Your implementation should include the following features:

1. A document should be specified or selected using a file chooser from the JavaFX application.
2. Each submitted document should be **parsed into its set of constituent shingles** and then compared against the existing document(s) stored in an object-oriented database (**MicroStream DB**) and then stored in the database.

3. The **similarity of the submitted document** to the set of documents stored in the database should be returned and presented to the user, either graphically or in a tabular form.
4. You are also required to provide a **UML diagram** of your design and to comprehensively **JavaDoc** your code.
5. You can assume that the **JavaFX** library dependencies are already available on the **module-path**. These have already been added to the **module-info.java** module descriptor. Do not include any of these dependencies with your submission or change how they are declared in the module descriptor.

In addition to these requirements, you should aim to accomplish the following in your application:

- Apply the **SOLID** principles throughout your application.
- Group together **cohesive** elements into methods, types and packages.
- **Loosely-couple** together all programme elements to the maximum extent possible.
- Create a **reusable set of abstractions** that are defined by interfaces and base classes.
- **Encapsulate** at method, type and package level.
- Apply creational, structural and behavioural **design patterns** throughout the application where appropriate.

You are free to asset-strip any online resources for text, images and functionality provided that you modify any code used and cite the source both in the README and inline as a code comment above the relevant section of the programme. You are not free to re-use whole components and will only be given marks for work that you have undertaken yourself. Please pay particular attention to how your application must be packaged and submitted and the scoring rubric provided. Marks will only be awarded for features described in the scoring rubric.

### 3. Deployment and Delivery

- **The project must be submitted by midnight on Friday 8th January 2021.** Before submitting the assignment, you should review and test it from a command prompt on **a different computer** to the one that you used to program the assignment.
- The project must be submitted as a Zip archive (**not a 7z, rar or WinRar file**) using the Moodle upload utility. You can find the area to upload the project under the “*Measuring Software Design Quality Using the Reflection API (50%) Assignment Upload*” heading of Moodle. Only the contents of the submitted Zip will be considered. **Do not add comments to the Moodle assignment upload form.**
- The name of the Zip archive should be `<id>.zip` where `<id>` is your GMIT student number.
- You must use the **package name** **ie.gmit.sw** for the assignment and the **module name** **gmit.software**.
- The Zip archive should have the following structure (do NOT submit the assignment as an Eclipse project):

Component	Category
plagiarism.jar	<p>A Java archive containing your API and runner class with a <b>main()</b> method. You can create the JAR file using the following command from inside the “<b>bin</b>” folder of the Eclipse project:</p> <pre>jar -cf plagiarism.jar *</pre> <p>The application should be executable from a command line using the <b>module-path</b> as follows:</p> <ul style="list-style-type: none"><li>• <b>Linux/Mac:</b> <code>java --module-path ./path/to/your/javafx-sdk-11.0.2/lib --module gmit.software/ie.gmit.sw.Runner</code></li></ul>

	<ul style="list-style-type: none"><li><b>Windows:</b> <code>java --module-path ./path/to/your/javafx-sdk-11.0.2/lib --module gmit.software/ie.gmit.sw.Runner</code></li></ul> <p>You only need to use the module-path syntax above to check your application after creating the JAR. Change the parameter <code>/path/to/your/</code> to the real system path on your computer.</p>
<b>src</b>	A directory that contains the packaged <b>source code</b> for your application.
<b>README.pdf</b>	A <b>PDF</b> file detailing the main features of your application in <b>no more than 300 words</b> . All <b>features and their design rationale</b> must be fully documented. You should consider also including the UML class diagram in this document to help explain your design clearly.
<b>design.png</b>	A <b>UML class diagram</b> of your API design. The UML diagram should only show the relationships between the key classes in your design. Do not show private methods or attributes in your class diagram. You can create high quality UML diagrams online at <b>www.draw.io</b> .
<b>docs</b>	<p>A directory containing the <b>JavaDocs</b> for your application. You can generate JavaDocs with the following command from inside the “src” folder of the Eclipse project:</p> <pre>javadoc -d [path to javadoc destination directory] ie.gmit.sw</pre> <p>Make sure that you read the <b>JavaDoc</b> tutorial provided on Moodle and comment your source code correctly using the <b>JavaDoc</b> standard.</p>

### 4. Marking Scheme

Marks for the project will be applied using the following criteria:

Element	Marks	Description
<b>Structure</b>	5	All-or-nothing. The packaging and deployment are both correct. The application launches correctly without any manual intervention from a command line as specified above.
<b>README</b>	10	All <b>features and their design rationale</b> are fully documented in 300 words or less. The README should clearly document where and why any design patterns have been used.
<b>UML</b>	10	A UML class diagram that correctly shows all the <b>important structures and relationships</b> between types in your design.
<b>JavaDocs</b>	10	All classes are fully commented using the <b>JavaDoc</b> standard and generated docs are available in the <i>docs</i> directory.
<b>Robustness</b>	25	The application fully implements the Basic Requirements 1-3.
<b>Cohesion</b>	10	There is very high <b>cohesion</b> between modules, packages, types and methods.
<b>Coupling</b>	10	The API design promotes <b>loose coupling</b> at every level.
<b>Extras</b>	20	Only <b>relevant extras</b> that have been fully documented in the README.

You should treat this assignment as a project specification. Any deviation from the requirements will result in some or all marks not being earned for a category. Each of the categories above will be scored using the following criteria:

Range	Expectation
<b>0–39%</b>	<b>Not delivering</b> on basic expectations. The submission does not meet the minimum requirements.
<b>40–60%</b>	Meets <b>basic</b> expectations. The minimum requirements are met in whole or in part.
<b>61–74%</b>	Tending to exceed expectations. A <b>high-quality</b> assignment with additional functionality added.
<b>75–89%</b>	Exceeding expectations and <b>very high-quality</b> . Any mark over 70% requires evidence of <b>independent learning</b> and cross-fertilization of ideas, i.e. your project must implement a set of features using advanced concepts not covered in depth during the module.
<b>90–100%</b>	<b>Exemplary</b> . Your assignment can be used as a teaching aid with little or no editing.

## 5. Efficient Detection of Plagiarism

The rapid detection of plagiarism in documents and data is an important area of study in computing. If we consider plagiarism detection as essentially a “matching problem”, then a vast number of potential applications present themselves. These include matching watched Netflix movies, online customer profiling, plagiarism identification, news aggregation services and bioinformatics. In the context of the big data age that we now live in, the challenge is to analyze large numbers of large documents in a computationally efficient manner, i.e. with a low space and time complexity. Note that some applications will be data-centric and require duplication detection at a byte or byte-block level. The more obvious applications are document-centric and focus on the identification of matching letters, words and sub-sentences.

### 5.1 Jaccard Similarity

A commonly employed technique for measuring the degree of similarity between two documents is to represent the documents as sets of letters, words or sub sentences. If we decompose a document into its *set of constituent words*, we can measure the similarity between them using the Jaccard Index. Developed by the Swiss botanist Paul Jaccard in 1901, the Jaccard Similarity of two sets, **A** and **B**, is expressed as:

$$J(A, B) = \frac{|A \cap B|}{|A \cup B|} = \frac{|A \cap B|}{|A| + |B| - |A \cap B|}$$

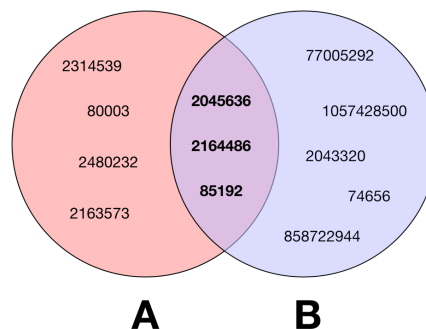
The intersection of two sets is the subset containing shared elements or the number of words common to both documents. The cardinality of a document,  $|A|$ , can be measured as the number of words that it contains. In practice, we hash the words, e.g. using the `hashCode()` method of `String`, and store that value instead of the actual sequence of characters. This has the twin benefits of eliminating the storage of variable length strings and provides a more efficient representation of a word. Consider the following two sets, **A** and **B**, defined as `String` array literals:

```
String[] A = {"John", "Paul", "Anne", "Pat", "Emer", "Una", "Enda"};
String[] B = {"Peter", "Anne", "Francis", "Enda", "Joe", "Alan", "Padraig", "Una"};
```

Clearly,  $A \cap B = \{\text{Anne, Enda, Una}\}$ . By calling the `hashCode()` method of `String`, both sets can be represented more efficiently as follows, as each hash code is a fixed length 32-bit integer that can be manipulated in a number of different ways.

```
int[] A = {2314539, 2480232, 2045636, 80003, 2163573, 85192, 2164486};
int[] B = {77005292, 2045636, 1057428500, 2164486, 74656, 2043320, 858722944, 85192};
```

The two sets can be visualized as a Venn diagram as follows:



The intersection of the two sets ( $A \cap B$ ) is the three integer representations of {Anne, Una, Enda}. The cardinality of the sets ( $|A|$  and  $|B|$ ) are 7 and 8 respectively. Thus, we can compute the Jaccard Similarity of the two sets as  $J(A, B) = 3 / (7 + 8 - 3) = 3 / 12 = 1/4$ .

## 5.2 Shingles and $k$ -mers

Despite its simplicity, using individual words as the basic unit for a document set element has the disadvantage of eliminating the semantics of a sentence when computing the Jaccard Index. Consider the following two sets **A** and **B**:

**A** = {object, oriented, programming, is, good, fun, for, me}  
**B** = {me, object, for, good, programming, is, fun, oriented}

Although both document sets have different meanings, their Jaccard Index is  $J(\mathbf{A}, \mathbf{B}) = 8 / (8 + 8 - 8) = 1$ , i.e. they are identical... In practice the following two alternatives to individual words are used:

- **Shingles:** the basic unit (element) is a fixed-size group of words. This has the advantage of retaining some of the semantics of the document. Because each individual word may have a variable length, the resultant shingles may also have variable length. We can represent the sets **A** and **B** with 3-word shingles as follows:

**A** = {object oriented programming, is good fun, for me}  
**B** = {me object for, good programming is, fun oriented}

The Jaccard Index is now  $J(\mathbf{A}, \mathbf{B}) = 0 / (3 + 3 - 0) = 0$ , i.e. the semantics of each document set has been preserved. This approach is often used when measuring the similarity of documents intended to be read by humans.

- **$k$ -Shingles /  $k$ -mers:** the basic unit (element) is a fixed-size block of characters of length  $k$ . A  $k$ -Shingle is more often referred to as a  $k$ -mer or  $l$ -mer in computer science (the suffix *-mer* means ‘part’ in Greek). The sets **A** and **B** can be represented as 5-mers by creating a tiling of characters of size  $k$  (spaces are shown below with an underscore for clarity).

**A** = {objec, t\_ori, ented, \_prog, rammi, ng\_is, \_good, \_fun\_, for\_m, e\_\_\_\_}  
**B** = {me\_ob, ject\_, for\_g, ood\_p, rogra, mming, \_is\_f, un\_or, iente, d\_\_\_\_\_}

The Jaccard Index is now  $J(\mathbf{A}, \mathbf{B}) = 0 / (10 + 10 - 0) = 0$ . Note that a single character insertion or deletion (a small change) will have large impact on the resultant set. This approach is heavily utilised in bioinformatics for comparing DNA and protein sequences.

## 5.3 Computational Complexity of Jaccard Index

The Jaccard Index between two sets, **A** and **B**, can be computed by comparing every element in **A** against every element in **B** to determine  $\mathbf{A} \cap \mathbf{B}$  (the cardinality of **A** and **B** can be computed in constant time). A naïve implementation of such an approach has a time complexity of  $O(n^2)$  and will not scale, even with shingles. Consider the problem of computing the Jaccard Index of Tolstoy’s *War and Peace* (562,442 words) with Caesar’s *De Bello Gallico* (137,919 words). If we assume that one million shingles can be compared every second, computing the Jaccard Index can quickly escalate into a practically intractable problem:

Shingle Word Size	War and Peace Shingles	De Bello Gallico Shingles	Number of Comparisons	Time at 10 <sup>6</sup> Shingles / sec
1	562,442	137,919	77,571,438,198	22 hrs
2	281,221	68,959	19,392,859,550	5.3 hrs
3	187,480	45,973	8,619,048,689	2.3 hrs
4	140,610	34,479	4,848,214,887	1.3 hrs
5	112,488	27,583	3,102,857,528	52 mins
6	112,488	27,583	2,154,762,172	36 mins

7	93,740	22,986	1,583,090,575	26 mins
8	80,348	19,702	1,212,053,722	20 mins
9	70,305	17,239	957,672,076	16 mins
10	62,493	15,324	775,714,382	13 mins

Before looking at an optimisation technique that eliminates almost all the comparisons necessary to compute the Jaccard Index, the next section reviews the *set* operations in Java supported by the implementations of *java.util.Set<E>*.

### 5.4 Set Operations in Java

The *java.util.Set<E>* interface supports basic set operations through the *contains()*, *size()* and the bulk methods inherited from the interface *java.util.Collection<E>*:

Method Signature	Set Operation	Big-O (TreeMap)
boolean <b>addAll</b> (Collection<? extends E> c)	Union ( $A \cup B$ )	$O(n \log n)$
boolean <b>removeAll</b> (Collection<?> c)	Difference ( $A \setminus B$ )	$O(n \log n)$
boolean <b>retainAll</b> (Collection<?> c)	Intersection ( $A \cap B$ )	$O(n \log n)$
boolean <b>containsAll</b> (Collection<?> c)	Subset ( $A \subseteq B$ )	$O(n \log n)$
boolean <b>contains</b> (Object o)	Element of ( $x \in A$ )	$O(\log n)$
int <b>size</b> ()	Cardinality ( $ A $ )	$O(1)$

We can easily compute the Jaccard Index of two instances of *java.util.Set<E>* by using the *size()* method to compute cardinality and the *retainAll()* method after first copying the all the elements of the first set into a temporary collection. The *retainAll()* method is implemented in the abstract class *AbstractCollection* as follows:

```
public boolean retainAll(Collection<?> c) {
    boolean modified = false;
    Iterator<E> it = iterator();
    while (it.hasNext()) { //  $O(n)$ 
        if (!c.contains(it.next())) { //  $O(\log n)$ 
            it.remove(); //  $O(\log n)$ 
            modified = true;
        }
    }
    return modified;
}
```

In the case of a tree map, this method has a running time of  $O(n \log n)$ , as it requires accessing each of the  $n$  elements in set **B**  $O(n)$  and then searching for that element in set **A**  $O(\log n)$ . In practice the running time will be significantly worse, as the usage would be as follows for two sets of strings:

```
Set<String> a = ...; //Create and initialise set A
Set<String> b = ...; //Create and initialise set B
Set<String> n = new TreeSet<String>(b); //A set to store the intersection  $O(n)$ .
n.retainAll(a); //  $O(n \log n)$ . Note that the while loop uses two  $O(\log n)$  operations.
```

### 5.5 MinHash

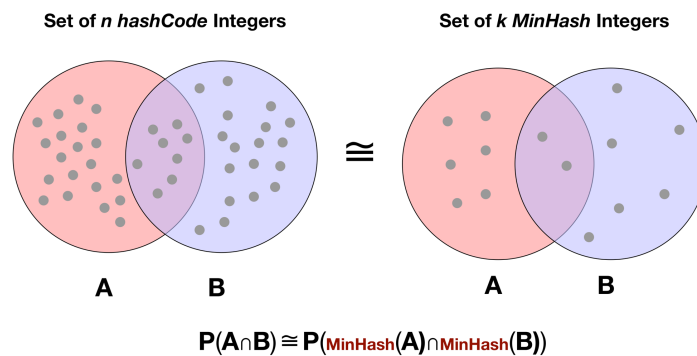
Although the Jaccard Index provides a mechanism for identifying duplication in documents, the number of comparisons required is prohibitively high for anything other than small data sets. For a comparison between large sets **A** and **B** to remain scalable, we require an algorithm that can compute or approximate  $J(A, B)$ , but at a fraction of the computational complexity. This is exactly what the **MinHash** algorithm does!

Developed in 1997 at AltaVista by Andrei Broder, **MinHash** uses  $k$  different integer hash functions and represents each document as the set of  $k$  elements containing the minimum value

for each hash function. To approximate  $J(A, B)$ , if  $n$  is the number of hash functions for which  $h_{min}(A) = h_{min}(B)$ , then  $J(A, B) \approx n / k$ . We can summarize the algorithm as follows:

- [1] Let  $M$  be a set of size  $k$ .
- [2] Generate a set  $H$  of  $k$  hash functions
- [3] For each function  $h()$  in  $H$
- [4]   Add  $\min(h(A))$  to  $M$
- [5] End For
- [6] Compute  $J(A, B) = A \cap B / k$

Assuming that an appropriate number of  $k$  hash functions are used and that each hash function is sufficiently different (XOR and bit rotations), the expected value of the MinHash similarity between two sets is equal to the Jaccard Index, i.e.:



Note that, as  $k \ll |A|$  or  $|B|$ , the computation complexity of  $J(A, B)$  is reduced to  $O(k)$ . A value for  $k$  can be computed from  $k = 1 / J(A, B)e^2$ , where  $e$  is the error rate. For example, if we expect  $J(A, B)$  to be 0.33 with an error rate of 10%, we will require  $k = 1 / 0.33 \cdot (0.1)^2 = 303$  shingles. The generation of a set  $H$  of  $k$  hash functions in [2] can be implemented with a rough approximation as follows:

```
//Create the set of hash integers as random numbers
Set<Integer> hashes = new TreeSet<Integer>();
Random r = new Random();
for (int i = 0; i < k, i++) { //Create k random integers
    hashes.add(r.nextInt());
}

//XOR the integer word values with the hashes
for (Integer hash : hashes) {
    int min = Integer.MAX_VALUE;
    for (String word : document) {
        int minHash = word.hashCode() ^ hash; //Bitwise XOR the string hashCode with the hash
        if (minHash < min) min = minHash;
    }
    shingles.add(min); //Only store the shingle with the minimum hash for each hash function
}
```