GALWAY-MAYO INSTITUTE OF TECHNOLOGY

Department of Computer Science & Applied Physics

B.Sc. Software Development – Advanced Object-Oriented Design Principles & Patterns (2017)

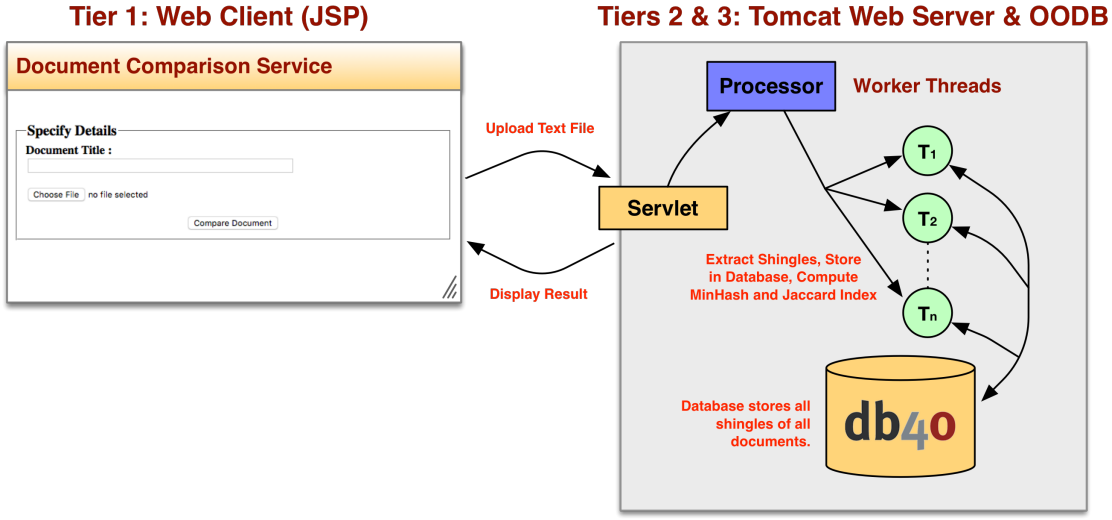
ASSIGNMENT DESCRIPTION & SCHEDULE

A JEE Application for Measuring Document Similarity

Note: This assignment will constitute 50% of the total marks for this module.

1. Overview

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



2. Minimum Requirements

Your implementation should include the following features:

1. **A document or URL should be specified or selected** from a web browser and then dispatched to a servlet instance running under Apache Tomcat.
2. Each submitted document should be **parsed into its set of constituent shingles** and then compared against the existing document(s) in an object-oriented database (**db4O**) 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 session user.

You are also required to provide a ***UML diagram*** of your design and to ***JavaDoc*** your code. Note that the whole point of this assignment is for you to demonstrate an understanding of the principles of object-oriented design by using abstraction, encapsulation, composition, inheritance and polymorphism WELL throughout the application. You should carefully consider how design patterns can be applied throughout your design. For example, patterns such as façade, chain of responsibility, command and proxy have obvious applications as mechanisms for handling incoming HTTP requests. Please pay particular attention to how your

application must be packaged and submitted. Marks will be deducted if you deviate from the requirements. Finally, as 4th year software students, you should appreciate that, if your code does not compile you cannot pass the assignment…

3. Deployment and Delivery

- ***The project must be submitted by midnight on Sunday 8th January 2018*** using both Moodle and GitHub.

GitHub:

- Submit the HTTPS clone URL, e.g. https://github.com/myaccount/my-repo.git of the public repository for your project. All your source code should be available at the GitHub URL. *You should try to use GitHub while developing your software and not just push changes at the end.*

Moodle

- The project must be submitted as a Zip archive ***(not a rar or WinRar file)*** using the Moodle upload utility. You can find the area to upload the project under the "A JEE Application for Measuring Document Similarity (50%) Assignment Upload" heading of Moodle.
- The name of the Zip archive should be <*id*>.zip where <*id*> is your GMIT student number.
- **Do not package the Db4O libraries** with your application.
- The Zip archive should have the following structure (do NOT submit the assignment as an Eclipse project):

Marks	Category
jaccard.war	A Web Application Archive containing the resources shown under Tomcat Web Application. All environmental variables should be declared in the file **WEB-INF/web.xml**. You can create the WAR file with the following command from inside the "jaccard" folder**: jar –cf jaccard.war *** You can package your API classes individually in the *WEB-INF/classes* directory or as a JAR archive in the *WEB-INF/lib* directory.
src	A directory that contains the packaged source code for your application.
README.txt	Contains a description of the application and an itemized list of any extra functionality added.
design.png	A UML diagram of your API design. Your UML diagram should only show the relationships between the key classes in your design. Do not show methods or variables in your class diagram.
docs	A directory containing the JavaDocs for your application. You can generate JavaDocs using Ant or with the following command from inside the "src" folder of the Eclipse project: **javadoc -d [*path to javadoc destination directory*] ie.gmit.sw** Make sure that you read the JavaDoc tutorial provided on Moodle and comment your source code correctly using the JavaDoc standard.

4. Marking Scheme

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

Marks	Category
(50%)	Robustness. The application executes correctly.
(10%)	Cohesion. High cohesion between methods, classes and packages (SRP)
(10%)	Coupling. Loose coupling between classes and packages (interfaces / abstractions)
(10%)	JavaDocs and UML Diagram
(10%)	Packaging & Distribution (GitHub and Moodle)
(10%)	Documented (and relevant) extras.

You should treat this assignment as a project specification. Any deviation from the requirements will result in a loss of marks. Each of the categories above will be scored using the following criteria:

- 0–30% Not delivering on basic expectations
- 40-50% Meeting basic expectations
- 60–70% Tending to exceed expectations
- 80-90% Exceeding expectations
- 90-100% Exemplary

5. Efficient Detection of Duplication

The rapid detection of duplication in documents and data is an important area of study in computing. If we consider duplication 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 twon 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:

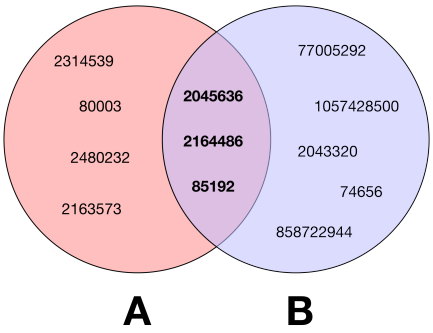
```
String[] A = {"John", "Paul", "Anne", "Pat", "Emer", "Una", "Enda"};
```

```
String[] B = {"Peter", "Anne", "Francis", "Enda", "Joe", "Alan", "Padraig", "Una"};
```

Clearly, **A** ∩ **B** = {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** ∩ **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 = ¼.

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**(**A, 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**(**A, 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(A, B)** = 0 / (10 + 10 – 0) = 0. Note that a single character insertion of 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 **A** ∩ **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 $\boldsymbol{O}(n^2)$ 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 10^6 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 **addAll**(Collection<? extends E> c)	Union (A ∪ B)	$\boldsymbol{O}(n \log n)$
boolean **removeAll**(Collection<?> c)	Difference (A \ B)	$\boldsymbol{O}(n \log n)$
boolean **retainAll**(Collection<?> c)	Intersection (A ∩ B)	$\boldsymbol{O}(n \log n)$
boolean **containsAll**(Collection<?> c)	Subset (A ⊂ B)	$\boldsymbol{O}(n \log n)$
boolean **contains**(Object o)	Element of ($x \in$ A)	$\boldsymbol{O}(\log n)$
int **size**()	Cardinality (|A|)	$\boldsymbol{O}(1)$

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.
```

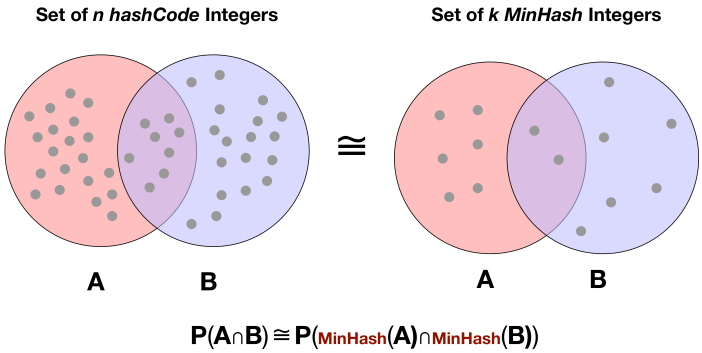
5.5 MinHash

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 $h_{min}(\mathbf{A}) = h_{min}(\mathbf{B})$, then $\mathbf{J(A, B)} \cong n / k$. 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 $min(h(\mathbf{A}))$ to **M**
[5] End For
[6] Compute $\mathbf{J(A, B)} = \mathbf{A} \cap \mathbf{B} / k$

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 $k \ll |\mathbf{A}|$ or $|\mathbf{B}|$, the computation complexity of $\mathbf{J}(\mathbf{A}, \mathbf{B})$ is reduced to $\boldsymbol{O}(k)$. A value for k can be computed from $k = 1 / \mathbf{J}(\mathbf{A}, \mathbf{B})e^2$, where e is the error rate. For example, if we expect $\mathbf{J}(\mathbf{A}, \mathbf{B})$ to be 0.33 with an error rate of 10%, we will require $\boldsymbol{k} = 1 / 0.33*(0.1)^2 = 303$ shingles. The generation of a set $\mathbf{H}$ of $\boldsymbol{k}$ 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
}
```