Code Clone Detection using Method Signatures

Dishant Langayan
SITE
University of Ottawa
Ottawa, Ontario, Canada
dishant.langayan@uottawa.ca

ABSTRACT

Through this paper we present a new approach towards code clone detection between Java ARchieve files (JARs). Many large Java projects with multiple developments often end up duplicating JAR files and class between modules. Moreover, different teams end up using different versions of the reference libraries, which make the software unstable and difficult to maintain. Our approach uses the concept of Method Signatures to construct unique fingerprints of JARs. We compare these fingerprints with unknown JAR files to detect the clone in compiled code and in specific duplication of method signatures. The implementation of approach is efficient in both performance and accuracy.

1. INTRODUCTION

[Introduction paragraph.]

1.1 Problem

This paper focuses on the problem of detection of code clones between close source software programs and modules. Although there has been extensive research in this area and many systems have been developed to detect code clones in software, we focus on one particular type of technology and software, i.e. the programs that have been developed for the Java Virtual Machine (JVM) and have been packaged in Java ARchieve files called JARs. In other words, we detect code clones in software that have been developed using the Java Programming language.

Often the source code for the software is not available or is compiled to binary forms, which makes it even harder to detect code clones. Many clone detection software decompile binary files to obtain and analysis the source code []. These types of tools require lot of system resources such as memory, and thus are not optimized for performance but rather accuracy. Others are faster in performance but produce less accurate results.

In the following sections the paper addresses these problems of code clones for closed source software and optimizes our approach for both performance and accuracy.

1.2 Motivation

The main motivation of the paper was to implement an open source code clone detection program. In large software projects with multiple teams, programs packaged in JAR files often get duplicated between the teams, and at many times developers use different versions of the JAR file. This makes it difficult to manage and maintain the project and decreases the stability and extensibility of the software. Many solutions, to solve problems like these, have been proposed, but most are proprietary or costly and are not open source [2]. This was the primary motivation for the design of our approach.

1.3 Goals

Our main goal is to detect code clones, in specific method duplication, between two JAR files that contain compiled Java code using the concept of method signatures.

To achieve our primary goal we implement our approach using to Java Programming language and define the following sub goals:

- Generate a unique fingerprint for a JAR file that contains compiled Java classes and other file format.
- Compare a fingerprint with an unknown JAR file for code clones in the best optimal time that increases exponential with the fingerprint size.
- Output the similarity or certainty percentage between the fingerprint and unknown JAR file.

1.4 Objectives

One of the important objectives of our approach is to make the detection phase of our program highly efficient in terms of memory and performance for large JAR files (i.e. files > 5MB) while producing accurate results.

Another objective is to output 100% certainty when comparing a fingerprint to the same JAR file from which the fingerprint was generated. To achieve this objective each of our fingerprints has to be independent of any time constraints and have to be constructed in the same manner, regardless of the JAR file. This also leads us to another objective, i.e. to output a high certainty when comparing a JAR file to itself that has been re-factored significantly in terms of class names, package structures, methods moved to other classes and packages, and in terms of the size of JAR file itself.

Our final objective is to make the size of the fingerprint as small as possible while retaining the accuracy of the program. Although our fingerprint size will large depend on the number of methods in all classes in the JAR file, we can reduce the size by hashing strings to integer values or by including methods signature in the fingerprint that we feel represent the uniqueness of the fingerprint. In other words, to achieve our objective we could exclude methods like getter and setter functions from the fingerprint. Though this would reduce the accuracy by a small degree, it would still comply with our previous objectives.

1.5 Outline

The remaining part of the paper is outlined as follows: Section 2 presents some background topics, terminology, and concepts used in the approach and program. We also look into similar approaches to ours and related software. Section 3 presents our approach, where we describe the design of our program. Section 4 presents the results obtained from analysis of our implementation.

Section 5 concludes the paper with future work and improvements that can be made in our approach.

2. BACKGROUND

2.1 Method Signatures

We employ a very simple and elegant approach in detecting code clone using methods signatures. We define a method signature as a single string representation of a method that uniquely identifies it with in the class. A method signature string would usually include the methods name, its parameter types (in order), and it return type. In the Java Programming language, a method signature is usually composed of two components, i.e. the methods name and it parameter types [3]. Listing 1 show a typical Java method declaration and Listing 2 shows its signature constructed by the Java environment.

```
public double calculateAnswer(double wingSpan,
int numberOfEngines, double length, double
grossTons) {
        //do the calculation here
}
```

Listing 1: A typical Java method

```
calculateAnswer(double, int, double, double)
```

Listing 2: Signature of the method declaration in Listing 1

These types of method signatures are typically used by object-oriented programming languages to support method overloading, which means there can be two or more methods with the same name in a class as long as the number of parameters or their types is different. We use the concept of method signatures in our clone detection approach and include the method's return type as well in the signature.

Often a single signature can become very long as the method might have many parameters. Comparing these long strings can reduce the performance of the program and consume more memory space. An alternative and efficient way to compare these strings would be to use a Hash function to compute hash values for all the strings and then compare the hash values. A hash function is a mathematical function that converts a string input to a signed integer value. Although hash values cannot be consider unique, i.e. more than one string input can may to the same hash value, in a simple example using the hasCode() function in java.lang.String API on a large set of strings, we obtained and evenly distribution on hash values. In other words, each string was mapped to a unique has value.

2.2 Related Software

There has extension research software developed to detect code clones [2], but most analyze source code files and are targeted for many programming languages.

[Add specific project and research works.]

3. APPROACH

3.1 Design

To detect code clones, the program is divided into two main parts a fingerprint generator that will generate a unique fingerprint for a JAR file, and a fingerprint detector that takes an existing fingerprint and an unknown JAR file. The approach considers certain file formats in a JAR and constructs the fingerprint based on these files. For the purpose of detecting code clones we consider only compiled Java classes, i.e. files ending with a ".class" extension. For each of these Java class files we only consider the signature of methods in that class.

In our approach, the uniqueness of a JAR file is represented by the method signatures rather class names, inheritances, or other dependencies, because class names and the structure of classes can be re-factored very easily. Even though methods can be moved to other packages and classes, the signature of a methods and its function will very likely remain the same. Moreover, we only consider unique method signatures throughout the JAR file under observation.

When determining the certainty percentage between a fingerprint and an unknown JAR file, comparison of method signatures contribute 90% to the overall certainty percentage. The rest 10% depends on other factors and files in the JAR, i.e. we also compare JAR names, size of the JAR, number of contents in the JAR file, etc. Table 1 shows the list of all the metrics used in the fingerprint for comparison and the weightings given to them.

Table 1. List of metrics and their weightings

Metrics	Weighting	Overall %
Method signatures	1 (for each match)	90
Name of JAR	1	10
Size of JAR file in bytes	1	
No of entries in JAR	1	
No of .html files	1	
No of .xml files	1	
No of .txt files	1	
No of .java files	1	

The following subsections describe how the program generates a fingerprint, how it compares it with other JARs, and some of the decision made in the design for performance and accuracy purposes.

3.2 Fingerprint Generator

The fingerprint generator accepts only the path of the JAR file. Since our approach considers methods signatures in compiled Java class files, we extract only files with "`.class`" extension to a temporary directory using the `java.util.jar` Java API. This API contains many useful methods and classes for analyzing a JAR file. Using the `JarFile` class in the API with can get a list of all the entries in the JAR and iterate or retrieve individual entries. The program iterates through all the entries and whenever an entry is a class file we retrieve an input stream from the JAR using the above mentioned API and output the stream to the temporary directory. For all other entries only the count is maintained. Upon termination of the generator the temporary directory is deleted.

Once all class files have been extracted the program makes use of the org.netbeans.modules.classfile API in the Netbeans package [5], to get a list of methods from the class file. This Netbeans API loads the Java class to a ClassFile object, which then allows us to iterate through the all the methods and it parameter in that class. This Netbeans API does not allow classes to be instantiated or executes its method, like the Java Reflection API, and is particularly useful for our approach as we not concerned about instantiating classes or executing their methods and statements.

The method signature, constructed by using the Netbeans API, consists of the method's return type, method name, and its parameter types. These signatures are then stored in a HashSet which stores only unique values, so duplicate signatures are not stored. After all class files in the JAR have been analyzed, each method signature is hashed to output a signed integer value. Since we are looking for exact matches comparing integer values is faster that comparing two strings. Though another motivation was to reduce the fingerprint size as some signature strings can be large, hashing the signature and comparing integers neither improved the performance nor reduce it. An array of these hashed signatures was then constructed and sorted.

To construct the final fingerprint, all the values obtained from the analysis were encoded to a single string, separated by the newline delimiter "\r\n". Using this approach our encoding could be saved to a plain text file, and easily decoded by our fingerprint detector for comparison. Each fingerprint also contained our generator's unique identifier, which the detector uses to verify that a fingerprint was constructed by our generator. The first eight values in the encoding are the generator id, name of the JAR, size of the JAR, number of entries in the JAR, number of HTML, XML, Text, and Java source files. The rest of the encoding consists of the list of hashed method signatures.

3.3 Fingerprint Detector

The fingerprint detector program takes in a previously generated fingerprint and a JAR file, and computes the certainty percentage between the two. The detector's first task it to make sure that the fingerprint was constructed from our generator. It then decodes the encoding in the fingerprint to an array. To compare this encoding with the specified JAR file, the detector uses the fingerprint generator, described in the previous section, to generate the fingerprint for the JAR file. An array is also constructed from the newly created fingerprint encoding. The detector then compares the two arrays.

The first eight entries in the arrays contribute only 10% towards the total certainty and are entries like name of JAR file, size of JAR, etc. The detector does not look for exact matches for these initial encodings. For example, when comparing JAR file names the program check whether the first file name "contains" the second file name. Therefore, commons-attributes-api-2.2.jar and commons-attributes-api-1.0.jar would be considered as a match. Similarly, when comparing number of XML files, if the difference is less than or equal to 1, then it would be considered as a match.

When comparing method signature, exact matches are considered only. The hashed signatures from the first array are compared with all the hashed values in the second array until a match is found. All matches contribute to the remaining 90% of the total certainty percentage. To calculate this percentage, the number of matches found is divided by the method lists size of the first array. The final certainty is then the addition of these two percentage values.

3.4 Decisions Made

Several decisions were made in the design that affected the performance and accuracy of the program. One of the biggest decisions was the use of Netbeans ClassFile API over Java Reflection API. The initial implementation involved the use of the Reflection API, which proved to be very slow for large JAR files, and impacted the performance greatly. Although the Reflection API gives much more information about a class, the ClassFile API is low in memory usage as it does not load a class file and it dependencies. Moreover, we only required the signature of a method and not its implementation, therefore, ClassFile was the appropriate API for our program.

To speed up the comparison of method signatures for very large JAR files it was decided to hash the signature string, as comparison of hashes it faster would consume less system resources. Although hashes are not unique and more than one string key can map to the same hash value, we believe that the false positive rate will be negligible.

4. RESULTS

To measure the performance and accuracy of the approach, various JAR files were used from the Eclipse 3.6 Classic IDE project and compared to each other. Different versions of the same JAR files were also included in the analysis as a reference, since they general should result in a higher similarity percentage, unless there is a major source code change between the two versions.

But all these results don't give us knowledge on whether our approach is returning valid code clones. To measure the validity of our approach we obtained the source for the commons-attributes-api-2.2.jar and performed several type of re-factoring and compared them to each other, and expected a certainty percentage range to be 80-95%.

The subsections below present the performance and accuracy results of our approach during construction of the implementation.

4.1 Performance

For performance measurement, JAR file in the Eclipse project, of sizes ranging from 30KB – 10MB, were used. These JARs were compared with themselves and different versions of them, which were obtained from their respective project web sites. Besides JAR files from the Eclipse project, we also compared our JAR file, i.e. our implementation of the approach to itself. The analysis was done on a Sony Vaio laptop with Intel® Core™2 Duo processor (T6600 @ 2.20GHz), 4BG of memory running Windows 7 64-bit operating system. Same tests were also done on a low memory configuration UNIX system, and computation times were 1-5 seconds slower large JAR files (i.e. > 5MB) and negligible for small files. Therefore memory and other hardware didn't impact much on the performance.

Table 2 show the results obtained from the performance analysis. In general, it was observed that for JAR sizes less than 100KB the certainty percentage was calculates within 2 seconds. The time shown in milliseconds includes the time to extract the second JAR file to the temporary directory. The performance of our approach cannot be measured and compared with just the file sizes as JAR files can contain other libraries and JARs (not included in the calculation of the certainty percentages), but variably depends on the number of methods to be compared with and the extraction time of the second JAR file.

This was also observed when junit-4.8.jar, with 600 methods in the fingerprint, was compared with junit-3.8.1.jar, with 318 methods in the fingerprint. The certainty was calculated approximately in 2.5 seconds. But when the same test was reversed, computation time doubled as the number of methods to compare with also doubled.

Table 2. Performance testing with various file sizes

	JAR Files	Certainty %	Time (ms)
1	commons-attributes-api-2.2.jar (36KB) to itself	100	733
2	commons-attributes-api-2.2.jar (36KB) to commons-attributes-api-2.0.jar	97.14	765
3	com.dishant.fpgd.v1.0.jar (66KB) to itself	100	406
4	junit-4.8.jar (232KB) to itself	100	5242
5	junit-4.8.jar (232KB) to junit-4.7.jar (227KB)	95.11	5226
6	junit-4.8.jar (232KB) to junit-3.8.1.jar (119KB)	25.59	2434
7	Previous test in reverse	44.47	5476
8	org.eclipse.jface.text_3.6.1.jar (955KB) to itself	100	19375
9	org.eclipse.jdt.ui_3.6.1.jar (9758KB) to org.eclipse.jface.text_3.6.1.jar (955KB)	5.37	19853
10	Previous test in reverse	21.71	192770

The performance of our approach was greatly impacted in extracting class files from the JAR to a temporary directory. For JAR files great than 1MB in size, extraction time ranged from 30 seconds to several minutes. Although, the time to compare the two fingerprints was always within a minute. Figure 1 show the extraction time for various large JAR files when compared to the time to compare the fingerprint with the commons-attributes-api-2.2.jar (36KB, 118 methods). Figure 2 show the same time to compare the fingerprint for the same JARs in details. From this figure it can be seen that when the commons-attributes-api-2.2.jar is compared with various other JARs the comparison time of fingerprint is fairly constant and fast, .i.e. within the range of a few milliseconds.

As discuss in section 3.4, one of the decisions made was to hash the method signatures in-order to improve performance of the comparisons. The difference in time was negligible for smaller JAR files, when the same test in Table 2 was performed without hashing, but for larger JAR files difference was seen in 1-15 seconds. Figure 3 present the comparisons of the results obtained when large JAR files were compared without hashing signatures.

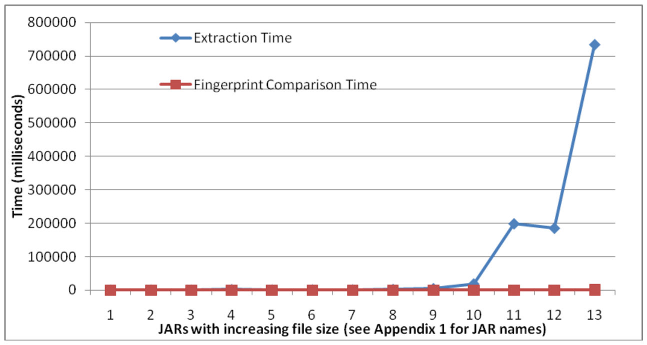

Figure 1: Extraction time and fingerprint comparison time for various JAR files increasing in file size

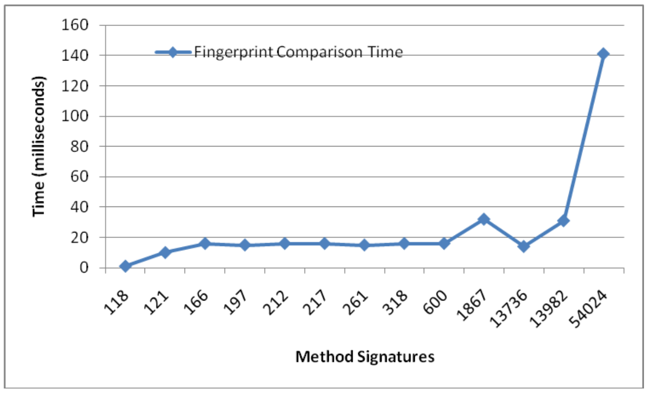

Figure 2: Comparison of method signatures with various JARs

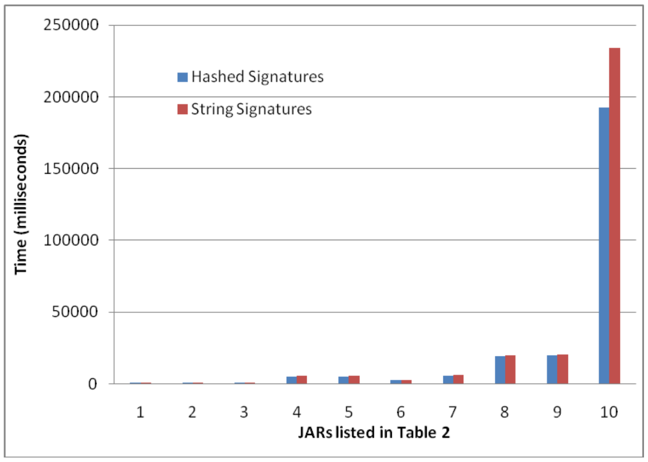

Figure 3: Performance comparison between hashed signature and un-hashed signatures.

4.2 Accuracy

One of our main objective was to return a 100% certainty when a JAR if compared to itself. This was confirmed in the results obtained in Table 2 above. Using our approach we will always get a 100% certainty as we are comparing fingerprints and the fingerprint generated for an unmodified JAR file will always be the same regardless of the time and number of times it is generated.

A high certainty percentage was also seen when various versions of the JARs were compared. The certainty obtained for minor version build was above 90%, and for some major version changes we saw certainty as low as 44%. This was seen when comparing JUnit v3.8.1 with JUnit v4.8, as there were many new features added and removed from version 3.8.1 to version 4.8.

To measure the accuracy of our approach, the source code for commons-attributes-api-2.2.jar was re-factored; some with minor changes others with major structural changes, and was compared to the original unmodified version. We obtain certainty percentage ranging between 84-99%, confirming that approach is resilient code refactoring with high accuracy results. Table 3 lists the re-factorings applied and result obtained.

Table 3. Accuracy results when performing various refactoring on original source code

	Refactoring Description	Certainty %	Time (ms)
1	No refactoring applied	100	733
2	Comments removed and variable names changed	99.23	723
3	Class names renamed	98.14	803
4	Classes moved to other packages	97.43	743
5	Packages removed	81.25	699
6	Additional methods and classes added	97.88	786
7	Some methods removed	84.78	755

5. CONCLUSION

Though this paper we have presented a fast and accurate clone detection approach using method signatures, that is resilient to minor and structural code refactoring. Although our approach only works for compiled Java classes, it is an advantage as we don't have to decompile the class file to obtain the source code, and thus improve the performance of the detection. Our approach lacks in detail analysis of code clones, such as detecting whether statements within methods were copied from another JAR. Our program cannot detect code clone for Intellectual Property (IP) purposes, but rather it is a tool for detection of code clones between JARs that have re-used method implementations.

We were able to achieve all our goals and satisfy mostly all our objectives. Though we were not successfully in reducing the fingerprint size for very large files, it didn't impact the performance of the program, and the comparison mainly depended on the method signatures to compare, as our worst-case computing time is $O(n \cdot m)$, where n and m are the number of methods in the two JARs being compared.

5.1 Future Work

Although we believe our implementation of the approach is fast in detecting code clone for closed source JAR files, there are several code optimization that can be made.

As many real-world JAR files include referenced libraries and APIs, we would like to extend the program to include sub JARs files are well, i.e. JAR files zipped within another JAR file. This would address the problems we had mentioned in section 1.1, i.e. to detect code duplication within sub modules of a project or duplication of libraries within a large project.

6. REFERENCES

[1] Apache Commons. http://commons.apache.org/attributes/. Accessed November 11, 2010.

[2] Clone Detection Literature - University of Alabama at Birmingham. http://students.cis.uab.edu/tairasr/clones/literature/. Accessed November 11, 2010.

[3] Defining Methods, Java Programming Language. http://download.oracle.com/javase/tutorial/java/javaOO/methods.html. Accessed November 11, 2010.

[4] JAR File Specification. http://java.sun.com/j2se/1.5.0/docs/guide/jar/jar.html. Accessed November 11, 2010.

[5] NetBeans ClassFile Reader Java API Documentation. http://bits.netbeans.org/dev/javadoc/org-netbeans-modules-classfile/org/netbeans/modules/classfile/ClassFile.html. Accessed November 11, 2010.

[6] [More references required.]

7. Appendix I

Table 3: Comparison of Various JARs to the commons-attributes-api-2.2.jar

JAR Name	JAR Size	Methods	Extraction Time	Fingerprint Comparison Time	Certainty
commons-attributes-api-2.2.jar	36	118	733	1	
apache-commons-codec-1.3.jar	46	121	796	10	5.05
javax.servlet.jsp_2.0.0.v200806031607.jar	63	166	764.5	16	8
org-netbeans-modules-classfile.jar	157	197	1934	15	8.1
javax.servlet_2.5.0.v200910301333.jar	117	212	991	16	7.24

org.eclipse.ant.core_3.3.jar	90	217	1295	16	7.33
org.eclipse.core.runtime_3.6.0.v20100505.jar	70	261	1295	15	7.34
junit-3.8.1.jar	119	318	2333	16	5.81
junit-4.8.jar	232	600	4984	16	8.861
org.eclipse.jface.text_3.6.1.r361_v20100825-0800.jar	955	1867	19305	32	8.099
tools.jar	12341	13736	198921	14	9.62
org.eclipse.jdt.ui_3.6.1.r361_v20100825-0800.jar	9758	13982	185718	31	7.43
rt.jar	43607	54024	734834	141	14.2
commons-attributes-api-2.2.jar	36	118	733	1	
apache-commons-codec-1.3.jar	46	121	796	10	5.05