A Method of Detecting Code Cloning in Open Source Software

Zhexiong Wei
The School of Computer Science,
Carleton University, Ottawa, Ontario,
Canada

ABSTRACT
In this paper, we implement a method which is used to detect code copying in open source software. This method generates a small fingerprint which includes main features of class files contained by a Java Jar file. By comparing the intrinsic and extrinsic features of class files, the method can find the cloned code in anonymous jar files. This method provides a tool to detect cloned code correctly and effectively.

Keywords
Open Source Software, Code Cloning, Jar File, Class File, Intrinsic Features, Extrinsic Features.

1. INTRODUCTION
Open source software (OSS) has become critical for most organizations. Because there are many advantages of using open source software[1], such as high-quality software, lost software costs, abundant support and accountability, less dependence on vendors and so on. Open source software has impact not just for developers and IT-managers but also potentially for all the persons throughout the value chain of an organization including suppliers, customers, and partners. Today, more and more open source software are developed and adopted in commercial products development. Although organizations get benefits from open source software, they have to take a critical view of open source should raise some questions as well. Most software is a working-in-program and not stable and secure. It maybe cause the commercial product failed. Open source software applied several open source licenses to protect itself. Those licenses are different and incompatible. Before using the open source, the organization must survey the license of the software and obey the license to develop the commercial products. However, code of unknown origin is encountered occasionally. It is difficult but necessary to make it clear. This is related to the legality and success of the product.

1.1 Problem
In this paper, we aimed at detecting software clone, which is written in java programming language. We use the known software code to compare with the software clone from one commercial product. Then we can assure whether the organization steal others' achievement illegally. There are many clone detection methods researched, such as string matching. But sometimes those methods are out of work when the original source code is not available. We focus on accessible attributes of compiled files and calculate the similarity of their attributes.

1.2 Motivation
"The Open Source community attracts very bright, very motivated developers, who although frequently unpaid, are often much disciplined. In addition, these developers are not part of corporate cultures where the best route to large salaries is to move into management, hence some Open Source developers are amongst the most experienced in the industry. In addition all users of Open Source products have access to the source code and debugging tools, and hence often suggest both bug fixes and enhancements as actual changes to the source code. Consequently the quality of software produced by the Open Source community sometimes exceeds that produced by purely commercial organizations."[2] For many developers, peer review and acclaim is important. They will prefer to build software with clean design, reliability and maintainability which is admired by their peers. They develop software to contribute the open source community and get benefits from others' contribution. But there are some phenomena, to which we have to pay attention, destroying the balance of the open source community. Some organization or individual steal the intellectual property, just make it their own by modifying some code but dedicate nothing to the open source community. Those behaviors show a complete lack of respect for developer's work and make a heavy attack on the ecosystem of the community. We must take action to prevent this ethical problem and protect authorities and the community. This is the main motivation to detect code clone in Open source software

As so far, there are tons of open source tools and libraries. How to safely leverage open source to enhance your own source code, without incurring the legal risks that often accompany open source becomes very important for the commercial software developer and manager. Some issues [3] which you have met or should avoid are listed in the following:

- Some commercial software suppliers have been sued by open source advocates for downloading the open source while ignoring the license obligations. Some of these lawsuits have been settled out of court, but all of the ones that have gone to trial have been settled in favor of the open source plaintiffs.

- Maybe your customers have heard about some of these lawsuits, and know that sometimes the plaintiffs target them as customers instead of you as their software supplier, so they are demanding that you give them a thorough accounting of what is inside your software.

- Maybe you suspect one of your engineers downloaded some open source and didn't tell anyone about the license obligations.

- Maybe you are worried that your source code includes some open source software that could impact the value of your intellectual property.

Based on those issues, organizations must take measures to protect against these legal risks of open source. They should periodically conduct a complete audit of their source code, making sure you know exactly what open source is inside and what the license obligations of that open source is, but not just create an open source policy. But the price of the service supported by a professional audit firm is high, and sometime there is no entry to the original source code. We devote our energy to audit the open source software with low cost

1.3 Goals

There are many available applications detecting code cloning with simple string-matching. We pay more attention to the feature attributes of a class, such as parameters of methods, return value and the like. We take those feature attributes from the Java Archive (JAR) file to composite the unique footprint. Then calculate the similarity between the know JAR's fingerprint and an unknown JAR file. Eventually the application will output the result to display match percentage between the two JARs.

1.4 Objectives

Several clone detection techniques have been described and implemented, such as Text-based techniques, Token-based techniques, AST-based techniques, Metrics-based techniques. But most of those techniques need the original source code and occupied much memory. Our objectives are to make a small fingerprint using the feature attributes not all the content of source code or the compiled class. A complicated method maybe be written over hundreds lines code. If we use the feature attributes to replace the method, there are only several strings.

Our approach does not rely upon the original source code, because we don't need to read the content of methods. We can also get feature attributes from the JAR files. Although we just utilize much less information, it dose not mean the low accuracy. Following Walenstein [4], clone detection adequacy depends on application and purpose. Intellectual property thieves maybe modify the route and method content but rarely features of the class. Representation of a class is not content details but features which are the key points to identify two files.

1.5 Outline

In the following sections, we will discuss some basic knowledge, and our main work concentrated in the section 3 including the design strategy, algorithm and what decision we made. In the section 4 and section 5, we compare several groups of files, present results, draw the conclusion and look ahead the future work.

2. BACKGROUND

Many researches in the field of clone code detection have been done. Most of them are mutual and have a high degree of accuracy. However some disadvantages exist in that software, e.g. time-consuming, memory-waste, and requiring the source files. As the open source software developed quickly, more unknown tools and libraries make software developers and organizations feel troubled. Fortunately, some researchers have paid their attentions to the situation that there is not source code available but compiled program. Cate Huston[10] uses winnowing to fingerprint JAR files and some potentially interesting information(e.g. filenames, size of the jar file, the number of entities, and the jar name) of the JARs, ,to detect significant similarities of those JARs. From the conclusion of that paper we can learn that those potentially interesting information are less consistent. This demonstrates what probability to be changed in the software clone is and the key point of comparison is the features or contents of methods. Carson Browns and David Barrera [8] use the modification of n-gram method to generate the fingerprint of the compiled java program. They make much improvement including detection speed, small fingerprint and good accuracy. We tend to do some research in this aspect. We also take JAR files as input, and generate the fingerprint of the Class files which are Java's compiled files. The difference between our researches is that we do not consider all the byte code of methods in Class files but just the features of methods, e.g. the count and type of the input parameters, the type of the return value. Although software clones have been made plenty modification, such as renaming classes, variables and methods, adding some inessential code lines, changing the path of files and so on, the main function of the method will not be destroyed. According this point, the features which we have mentioned above will be changed less. And org.netbeans.modules.classfile API [7] supports the function to implement the fingerprint of Class files. Patrice Arruda et al [9] use graph to describe the dependency of classes and calculate the similarity with matrix. Their approach focuses on the relationship between classes. The idea of extracting extrinsic features of a class in our approach is enligthened by [9].

3. APPROACH

3.1 Design

Considering that Java Jar files mainly comprise class files, this method analyses the class files and use features of class files to describe the fingerprint of jar files. We divide the features of class files into two groups: intrinsic features and extrinsic features. Intrinsic features of a class file include description of its methods. The number and types of input parameters and the type of method return compose the basic feature of a method in a class. The extrinsic features are composed of the relationship of classes, for example superclass, interfaces, and inner classes of a class. These features can indicate the purpose of the class file so that the approach can distinguish the behavior of cloning. Employing these intrinsic and extrinsic features can discover the cloning code which is just modified by refactoring functions of modern IDE.

3.2 Rules

Generally, jar files are kinds of Zip files. Thus before extracting features from class files, we need to decompress the jar file. Java Class Foundation Library provides API [5], java.util.jar, to finish this job. We use JarResource class which is from a Java World article [6] to obtain bytecode of all classes in a Jar file. The org.netbeans.modules.classfile API [7] can transform the bytecode of a class to an object in the memory. Therefore our algorithm can manipulate class files to get all features of methods

in a class file. To clarify our algorithm, we assume that X is an original Jar file which is used to generate a fingerprint and Y is an anonymous Jar file which is possible cloning Jar file. C_i denotes a Class file that belongs to X. D_j denotes a Class file that belongs to Y. M_k denotes a method that belongs to C_i. N_l denotes a method that belongs to D_j. The similarity of two methods is calculated by Rule 1:

Rule 1: If two methods have same number of parameters and same type of parameters and same type of return value, then the similarity of two methods $S_{methods}$ is 1; otherwise, the similarity $S_{methods}$ is 0.

If the similarity of two methods is obtained, the similarity of intrinsic features of two classes can be analyzed by Rule 2:

Rule 2: the similarity of classes $S_{in-classes} = \frac{\sum_{k=1}^{n} S_{methods}}{n}$, n is the number of M_k.

The similarity of extrinsic features of two classes can be calculated by Rule 3 to Rule 6:

Rule 3: the similarity $S_{ex-classes} = (S_{SuperClass} + S_{Interface} + S_{InnerClass}) / 3$.

Rule 4: the similarity $S_{SuperClass}$ is 1 if the type of Superclass is same; otherwise is 0.

Rule 5: the similarity $S_{Interface}$ is 1 if the number of interfaces is same and the types of interface are same; otherwise is 0.

Rule 6: the similarity $S_{InnerClass}$ is 1 if the number of inner classes is same and the types of inner classes are same; otherwise is 0.

The similarity of two classes is calculated by combining $S_{in-classes}$ and $S_{ex-classes}$ as Rule 7:

Rule 7: $S_{classes} = 0.6\, S_{in-classes} + 0.4\, S_{ex-classes}$.

The weight of $S_{in-classes}$ is set to sixty percentage points because our approach mainly focuses on methods in a class.

The similarity of two jar files is calculated by the Rule 8:

Rule 8: $S_{jars} = \frac{\sum_{i=1}^{n} \max S_{classes\_c_i}}{n}$, n is the number of classes in the original jar file, max $S_{classes\_c_i}$ indicates the maximum of similarity between C_i and $\{D_j | 1 \le j \le m\}$, m is the number of classes in the anonymous jar file.

3.3 Algorithm

Our approach compares the original jar file and the anonymous jar file by two components. The first component is Fingerprint Generator which manages to generate fingerprint from the original jar file. Figure 1 describes the main steps of generating fingerprint. The second component is Fingerprint Detector which is charge of computing the similarity of two jar files. Figure 2 describes the main steps of generating the similarity.

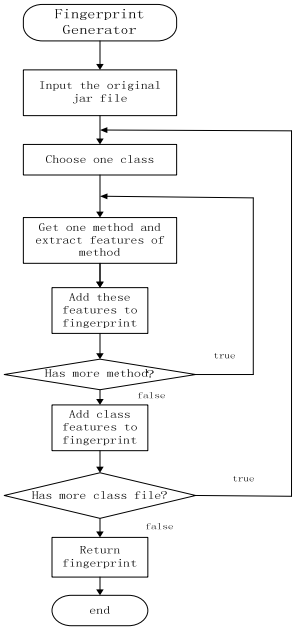

Figure 1: Fingerprint Generator

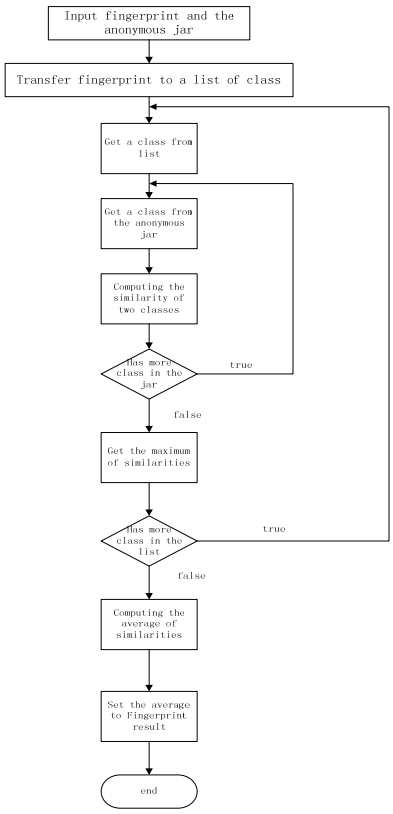

Figure 2: Fingerprint Detector

3.4 Decision Made

Our approach only focuses on class files in a jar file because class files are the core of a jar. The similarity between two methods is simply calculated by Rule 1 and expressed by 1 or 0. There is no value of similarity between 0 and 1. As a prototype of first implementation, we intuitively keep the algorithms simple and correct. Although this calculation losses some accuracy, the intention can be described clearly and results are satisfying reasonably.

We also only consider the basic type of Java such as int, short, long, byte, float, double, string, char and boolean, and the original objects of Java such as Integer, IO Stream and the like. Because the source code trends to be changed by name refactoring, not to be changed with the type and number of parameters of a method. Except these primary circumstances, other self-defined types are uniformly defined as Object type. Through these simple categories of types of parameter, our approach can cover most cases when comparing two methods.

In the course of implementing the algorithm, some classes in the different jar files are very similar but the similarity calculated by our algorithm is not high. However, some classes are different but the similarity is high. Through analyzing the source code artificially, we find that these classes have several methods which have no parameter and no return value. This circumstance will influence the result of comparison. Assuming that there are three classes A, B and C, Class A has 6 methods (X1, X2, X3, X4, X5, X6) among which there are two methods (X5 and X6) have no parameter and no return value. Class B has 4 methods (Y1,Y2 ,Y3 ,Y4). Class C has 4 methods (Z1, Z2, Z3, Z4) among which there are two methods (Z3 and Z4) have no parameter and no return value. Assuming X1, X2, X3 are equal to Y1, Y2, Y3, respectively, so the similarity of Class A and B is 0.5. Assuming X1, X2 are equal to Z1, Z2, respectively, the similarity of Class A and C is 0.67 because Class C has two methods Z3 and Z4 compared to X5 and X6 even if they are entirely different. Actually, Class A is more similar to Class B than Class B. Thus methods of no parameter and return value have negative effect in our algorithm. In most classes, methods of no parameter and return value trends to be less important. Considering this situation, we decide to remove methods which have no parameter and return value. The benefits contributed by these methods are much smaller than the harms.

4. RESULTS

To examine the correctness and effectiveness of this algorithm, eight selected jar files are used as the testing set. The environment of testing and the performance are described at first. Next the testing results and analysis are demonstrated.

4.1 Performance

This implementation runs on a laptop, whose technical parameters are shown as below:

Processor: Pentium(R) Dual-Core CPU T4500 @ 2.30GHz

Memory: 4.00 GB

System Type: 64-bit Operating System

The amount of jar files ranges from 1Kb to 2.6Mb. There are four jar files which come from Eclipse IDE plugins, two from our own implementations, one from Spring Framework, one from an anonymous company. All comparison finished in 126.919s. The worst case is the largest pairs of jar files. Although the algorithm spends a little more time, the result sounds good.

4.2 Testing Results

In the set of testing jar, we first choose the simplest jar file, test.jar, to compare with this jar itself. The result is very good and the time is ideal. Next when comparing two entirely different jar files, the result which is provided by this algorithm is correct. The jar files, org.eclipse.help.ui_3.2.0.v20060602.jar and org.eclipse.help.ui_3.5.0.v20100517, are obtained from Eclipse plugins. They have similar functions but different versions. The org.eclipse.help.ui_3.2.0.v20060602.jar calculates the similarity of this jar file itself and then calculates the similarity of org.eclipse.help.ui_3.5.0.v20100517. The result shows that the former's similarity is higher than the latter's. This data is reasonable because the difference between two versions is larger than one version itself. Testing the different versions jar files of JUnit, we discover that their similarity is low because JUnit version 4 has many modifications of structures and functions compared with JUnit version 3.8. JUnit version 4 introduces the annotation function to facilitate programming test units. The similarity of JUnit version 3.8 and JUnit version 4 is different with its reverse because of different figerprints produced by different jar files. Although the similarity has a little different, the time consumed by calculation is almost equal. Finally, we compare two larger jar files, one from Spring Framework and another from an anonymous company which has a possibility to

clone the jar files of Spring Framework. The result reveals that the similarity of these two jar files is 82.976% and the calculation costs 126.919s. This indicates the second jar file has a high probability of cloning source codes. Table 1 lists the all results of our testing set.

Table 1: Testing Results

Fingerprint Generated From	Another Java Jar File	Certainty	Time (s)
Test.jar	Test.jar	100%	0.015
Test.jar	Com.zxwei.comp5900.jar	0%	0.017
org.eclipse.help.ui_3.2.0.v20060602.jar	org.eclipse.help.ui_3.2.0.v20060602.jar	82.976%	126.919
org.eclipse.help.ui_3.2.0.v20060602.jar	org.eclipse.help.ui_3.5.0.v20100517.jar	98.846%	3.959
Junit.jar	Junit.jar	94.313%	3.797
Junit.jar	Junit-4.1.jar	98.0%	1.01
Junit-4.1.jar	Junit.jar	25.396%	0.875
Spring.jar	Ssb-core-01.50.00.jar	31.675%	0.891

5. CONCLUSION

Our approach of detecting code cloning in open source software focus on class files in jar files. Utilizing the intrinsic and extrinsic features of classes can effectively seek out the functional copies of codes. This algorithm can be immune to simple refactoring. Although the algorithm costs a little more time, the result is correct and accurate.

5.1 Review goal and contributions

Our algorithm and implementation fulfills the objective of detecting code cloning and provide a tool to help companies or institutions who want to carry out open source software avoid distinguish the copied codes.

5.2 Future work

Our approach has three main aspects of improvement in the future. In the calculation of similarity of two methods, we can set different weight to the number of parameters, types of parameters and type of return value instead of Boolean value (0 or 1). This improvement can further increase the accuracy of our algorithms. In the aspect of extrinsic features, attributes of a class and reference of other objects in a class are both important properties of a class. Adding these features into the calculation of extrinsic features can promote the accuracy of calculating similarity of two classes. In the aspect of intrinsic features, considering structures of if-else and while clauses can strengthen detailed description of a method, thereby escalating the accuracy of the algorithm. After improved on these three aspects, this algorithm will have a better result.

6. REFERENCES

[1] Jason Williams, Peter Clegg. Emmett Dulaney Expanding Choice: Moving to Linux and Open Source with Novell Open Enterprise Server Published Mar 7, 2005 by Novell Press

[2] "Analysis of the Impact of Open Source Software" (PDF) [QINETIC2001], Peeling & Satchell, QinetiQ, 2001.

[3] Manage The License Obligations And Risks That Come With Developing Commercial Software While Embedding Open Source Inside, http://www.sourceauditor.com/ Copyright, Source Auditor, 2008, All rights reserved.

[4] A. Walenstein. Problems creating task-relevant clone detection reference data. In Proceedings of the Working Conference on Reverse Engineering (WCRE'03), pages 285–294. IEEE Computer Society Press, 2003.

[5] Java Platform API Specification http://download.oracle.com/javase/6/docs/api/

[6] Arthur Choi, Java Tip 49: How to extract Java resources from JAR and zip archives, JavaWorld.com, January 3rd, 1998, http://www.javaworld.com/javaworld/javatips/jw-javatip49.html

[7] org.netbeans.modules.classfile, http://bits.netbeans.org/dev/javadoc/org-netbeans-modules-classfile/org/netbeans/modules/classfile/ClassFile.html

[8] Carson Brown, David Barrera, Dwight Deugo: FiGD: An Open Source Intellectual Property Violation Detector. SEKE 2009: 536-541

[9] Patrice Arruda, Pierre Chamoun, Dwight Deugo: A Framework for Detecting Code Piracy Using Class Structure. SEKE 2010. http://www.scs.carleton.ca/ocics/seminars/index.php?Abstract=OCICS_SEMINAR_0052&Num=1

[10] Cate Huston, Fingerprinting Jar Files Using Winnowing, http://catehuston.com/files/fingerprinting%20jar%20files%20using%20winnowing.pdf