How Should We Measure Functional Sameness from Program Source Code? – An Exploratory Study on Java Methods –

Yoshiki Higo
Osaka University
1-5 Yamadaoka, Suita, Osaka, Japan
higo@ist.osaka-u.ac.jp

Shinji Kusumoto
Osaka University
1-5 Yamadaoka, Suita, Osaka, Japan
kusumoto@ist.osaka-u.ac.jp

ABSTRACT

Program source code is one of the main targets of software engineering research. A wide variety of research has been conducted on source code, and many studies have leveraged structural, vocabulary, and method signature similarities to measure the functional sameness of source code. In this research, we conducted an empirical study to ascertain how we should use three similarities to measure functional sameness. We used two large datasets and measured the three similarities between all the method pairs in the datasets, each of which included approximately 15 million Java method pairs. The relationships between the three similarities were analyzed to determine how we should use each to detect functionally similar code. The results of our study revealed the following. (1) Method names are not always useful for detecting functionally similar code. Only if there are a small number of methods having a given name, the methods are likely to include functionally similar code. (2) Existing file-level, method-level, and block-level clone detection techniques often miss functionally similar code generated by copy-and-paste operations between different projects. (3) In the cases we use structural similarity for detecting functionally similar code, we obtained many false positives. However, we can avoid detecting most false positives by using a vocabulary similarity in addition to a structural one. (4) Using a vocabulary similarity to detect functionally similar code is not suitable for method pairs in the same file because such method pairs use many of the same program elements such as private methods or private fields.

Categories and Subject Descriptors

D.2.2 [**Design Tools and Techniques**]: Computer-aided software engineering; D.2.7 [**Distribution, Maintenance, and Enhancement**]: Restructuring, reverse engineering, and reengineering

General Terms

Experimentation, Measurement

FSE '14, November 16–22, 2014, Hong Kong, China

Keywords

Functionally similar code, Clone Detection, Structural similarity, Vocabulary similarity, Method name similarity

1. INTRODUCTION

In software engineering, program source code is one of the main research targets. Various studies have been conducted on source code, and these studies often utilize similarity of structure and/or vocabulary to measure the functional sameness of source code. For example, clone detection is a family of research studies that utilizes the similarities of code structure such as token sequences or abstract syntax trees [36, 38]. Detected clones are generally code instances implementing the same functions. They are sometimes merged as new modules [14, 15, 25]. On the other hand, keyword-based code searching is a representative study that utilizes the similarity of code vocabulary [18, 32]. Developers can obtain reusable code from keyword-based code searching systems by inputting keywords related to the function that they want.

Clone detection research assumes that, if the structures of two code units are identical or similar to each other, their functions are also identical or similar to each other. On the other hand, keyword-based code searching research assumes that, if the vocabulary in a code unit is similar to that of another unit, their functions are also similar.

However, of course, those assumptions do not always make sense. For example, herein, we consider two code units: one is an implementation of quicksort and the other is bubblesort. Both code units have the same function, which sorts numerical values stored in an array in ascending/descending order. Thus, both code units should include the same words such as "sort" or "array". In this case, the functional sameness appears in their vocabulary but not in their structures because the two code units implement different algorithms.

In addition, in some program languages such as C or Java, `for` statements are often used to perform an action for each element in a given array iteratively. Thus, many code units using `for` statements have a common operation that iteratively does something. However, such an iterative operation with a `for` statement is a stylized implementation. Such stylized implementations do not necessarily include the same words.

If a code unit is a declarative unit in programming language, the code has its own name. For example, in Java language, classes and methods have names. Corazza et al. reported that the vocabulary appearing in the signature of a method is the most informative one in Java language [6]. On the other hand, there are many Java methods whose names are not at all informative such as `main` or `actionPerformed`.

The purpose of this research is to reveal how we should leverage the three similarities, structural similarity, vocabulary similarity, and method name similarity, for measure functional sameness from program source code. In this research, we have investigated the relationships between functional sameness and the three similarities. In the investigation, we selected the Java method as the target code unit. To reduce bias in the investigation results, we conducted the same investigation on two different datasets. We investigated the relationships for approximately 14 million method pairs in the two datasets.

The following are the main findings of this research.

- The name of a method does not always reflect its function. In cases where a given name is used by a small number of methods, the degree of the functional sameness of the methods is likely to be high. However, if there are many methods that have the same given name, the degree of their functional sameness is low. Checking the number of methods having a given name is a way to learn whether the name well represents its function.
- There are hash-based clone detection techniques at the file level, method level, and block level. However, such techniques often miss functionally similar code generated by copy-and-paste operations between different projects.
- If we use structural similarity to detect functionally similar code in Java, we obtain many false positives such as consecutive switch-case or consecutive else-if statements. Such consecutive instructions are dependent on Java. If we use vocabulary similarity in addition to structural similarity, we can avoid the detection of most false positives.
- Method pairs in the same class share the same private fields an private methods. As a result, such method pairs tend to have a high vocabulary similarity. Consequently, using a vocabulary similarity is not suitable for method pairs in the same class.

The remainder of this paper is organized as follows: Section 2 describes the experimental design used in this study; Section 3 explains how we measure the three similarities; in Section 4, we describe how we prepared the datasets for the experiments; Section 5 shows the experimental results; and Section 6 discusses future research based on these results; Section 7 describes some threats to the validity of these experiments; Section 8 introduces existing works related to our experiment; lastly, in Section 9, we conclude this paper.

2. EXPERIMENTAL DESIGN

In this research, we investigate how the following three types of similarity should be used in detecting functionally similar code.

- Structural similarity
- Vocabulary similarity
- Method name similarity

Although all of the above similarities have been leveraged in existing research studies, they are not complete measures. Code pairs regarded as similar by these measures are occasionally recognized as false positives by humans. For example, in code clone detection, where structural similarity is generally leveraged, code fragments including repeated instructions tend to be detected as clones, but they are generally regarded as false positives [3, 12, 44]. Consequently, the authors made the following hypotheses.

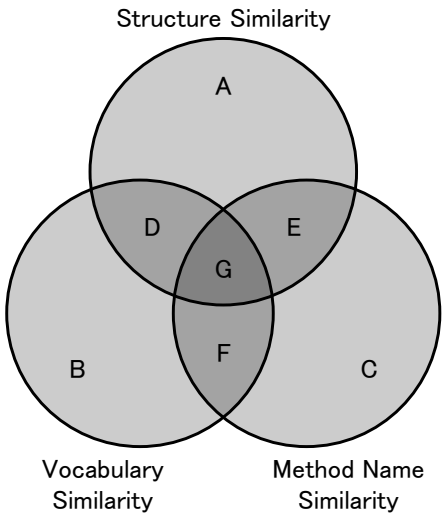


Figure 1: Three types of similarities used in this research

- Code pairs regarded as similar by two measures are more likely to be recognized as similar by humans than ones regarded by only one measure.
- Code pairs regarded as similar by all three measures are more likely to be recognized as similar by humans than ones regarded as similar by two measures.

Figure 1 shows the relationships between the three similarities. By using this figure, the probabilities that code pairs are functionally similar code are presented as follows.

$$A, B, C \leq D, E, F \leq G$$

In this research, we determine: (1) whether code pairs in each region (from A to G) are truly functionally similar code; and, (2) the characteristics of code pairs in each region. We conducted experiments on two large sets of open source projects written in the Java language. The details of the experimental targets are described in Section 4.

Java is an object-oriented language, and multiple classes operate by cooperating with one another. Each class should have its own responsibility, so that two methods in the same class tend to be more closely related to each other than two methods in different classes. In addition, Java has the notion of *package*, which includes a set of classes that closely cooperate with each other. Thus, two methods in the same package should tend to be more closely related to each other than two methods in different packages. To summarize the above-mentioned assumption, the degree of similarity between two methods depends on their distance apart in source code. In this research, we took into account the distance between a given code pair when investigating it. More concretely, we used four distance categories:

Within-File (WF) two methods are in the same file,

Within-Directory (WD) two methods are in different files, but in the same directory,

Within-Project (WP) two methods belong to different directories, but to the same project, and

Across-Project (AP) two methods are defined in different projects.

3. SIMILARITY MEASURES

In this section, we explain how we measured structural, vocabulary, and method name similarities from a given pair of Java methods.

3.1 Structural Similarity

The measurement procedure for *structural similarity* (*SS*) consists of the following steps, which are based on the clone detection procedure of Nicad [37].

STEP 1 The source code of each target method is transformed into a token sequence. In this step, all white spaces, tabs, and new line characters are deleted.

STEP 2 All the tokens representing variable names, method names, and type names are replaced with special tokens. The three types of special tokens are all different from each other.

STEP 3 The longest common subsequence between the two normalized token sequences is identified.

STEP 4 A quantified value of *SS* is calculated using the following formula.

$$SS(T_A, T_B) = min(\frac{|LCS(T_A, T_B)|}{|T_A|}, \frac{|LCS(T_A, T_B)|}{|T_B|})$$

where "T_A" and "T_B" are normalized token sequences obtained from methods "A" and "B", and "$|T_A|$" represents the number of tokens included in "T_A". "$LCS(T_A, T_B)$" shows the longest common subsequence between "T_A" and "T_B".

An identified longest common subsequence is not necessarily a consecutive subsequence of the original sequence. In other words, the longest common subsequence algorithm considers additional shorter equal subsequences among two sequences as common sequence in addition to the longest equal consecutive subsequence.

The experiment in this paper, we used 0.7 as a threshold for determining whether given two methods are structurally similar to each other or not.

3.2 Vocabulary Similarity

We use *Jaccard similarity* [43] as *vocabulary similarity* (*VS*). The steps for measuring *Jaccard similarity* in this research are as follows.

STEP 1 Variable names and method names are extracted from the source code of each method by performing syntax analysis.

STEP 2 Nouns and verbs are obtained from the extracted names with their dictionary forms by performing camel/snake case splitting and stemming. Note that stop words are ignored.

STEP 3 A quantified value of *VS* is calculated using the following formula.

$$VS(V_A, V_B) = \frac{|V_A \cap V_B|}{|V_A \cup V_B|}$$

where "V_A" and "V_B" show sets of words in two methods "A" and "B", respectively. "$|V_A|$" is the number of words included in "V_A".

The experiment in this paper, we used 0.7 as a threshold for determining whether given two methods have *vocabulary similarity* or not.

3.3 Method Name Similarity

In this research, the unit of investigation is the Java method. Each Java method has its own signature, so we need to quantify the similarity between two given signatures. However, quantifying signature similarity appropriately as a single value is very difficult because a signature includes multiple elements that need to be considered. For example, we need to take into account the method name, number of parameters, type of each parameter, and name of each parameter. In this research, instead of quantifying the similarity of a whole signature, we use the simplest way to determine whether signatures of two given methods are similar; that is, if their method names are exactly the same, their signatures are regarded as similar. If not, they are regarded as not being similar.

Readers may think why *method name similarity* is not measured in an analogous way with *vocabulary similarity*. Generally, a method name consists of a few English words. Measuring *Jaccard similarity* from such a small number of words is meaningless. Consequently, in this research, we chose a binary similarity for *method name similarity*.

4. DATASETS

In this research, we conducted experiments on the following two datasets in order to reduce bias due to the datasets used[1].

APACHE The entire set of Java projects included in the *Apache Software Foundation*[2]. The SVN repositories are open to the public. In the experiment, we used a snapshot taken on 2013/Oct/31.

UCI A large set of Java software projects that includes approximately 13,000 projects and 20 million methods[3] . If we were to use the entire *UCI* dataset, we would need to measure similarity between 200 trillion method pairs. In this research, we used 500 projects in the dataset, which were extracted by using the following steps.

STEP 1 13,000 projects were sorted in the order of the number of methods they included.

STEP 2 The sorted list was divided equally into 10 sections.

STEP 3 50 projects were randomly extracted from each of the 10 sections.

The *APACHE* dataset consists of directories and files that were checked out from SVN repositories. SVN repositories often include *branches* and *tags* directories. The former directories include files that belong to branches, and the latter ones include files of tagged versions. In order to exclude source files under such directories, we used only source files under *trunk* directories, which are used for storing mainstream development.

[1]The two datasets are open to the public on our website, http://sdl.ist.osaka-u.ac.jp/~higo/fse2014/

[2]http://www.apache.org

[3]http://www.ics.uci.edu/~lopes/datasets/

Table 1: Overview of Datasets

	APACHE	*UCI*
No. of projects	84	500
No. of files	66,724	60,548
No. of methods	628,219	532,556
Total LOC	11,545,556	10,073,635

In addition, we expended considerable effort to eliminate test cases from the datasets. We obtained a list of source files whose paths included "test". Then, we checked every source file in the list manually to identify whether it was a test case.

Our elimination targets were not only test cases but also source code generated by tools. Here we used the same strategy as for generated code. That is, first we obtained source files whose paths include "generated". Then, we checked each of them manually. We also obtained a list of source files where code comments include "@generated", "antlr", "javacc", "sablecc", or the names of other compiler compilers. Then, each of them was interactively checked, and eliminated if it was regarded as generated code.

The same data cleansing was performed on the *UCI* dataset because it included projects that had been checked out from SVN repositories. Table 1 shows numerical data of the two datasets such as the number of source files, the number of methods, and LOC. However, some of the methods should not be targets even if they are neither test cases nor generated code. When we make programs using the Java language, we generally define many small methods such as getters and setters. Measuring the similarity between such small methods does not make sense. In addition, Merlo et al. reported that small methods tend to have similar metric values even if their contents are different [31]. Consequently, we removed small methods from our measurement targets. In this research, a given method was regarded as small and ignored if it satisfied either of the following conditions.

- It included 50 or fewer tokens.
- It included 10 or fewer words that appeared in user-defined identifiers.

As mentioned above, method pairs were classified into four categories based on the distance between the two methods in the pairs. Table 2 shows the number of method pairs in each category. The *within-file* category has the least number of method pairs, and the *across-project* category has the largest number of method pairs.

5. INVESTIGATION RESULTS

We investigated method pairs in each region shown in Fig. 1 by browsing their source code manually. In this investigation, we used 0.7 for the thresholds of *structural similarity* and *vocabulary similarity*. Table 3 shows the number of method pairs in each of the regions. If 100 or more method pairs were included in a given region, we investigated at least 100 pairs. If fewer than 100 pairs were included in a given region, we investigated all the pairs. In the reminder of this section, we describe the result for each of the regions.

In this paper, we describe only the results for the *APACHE* dataset due to space limitations. However, we would like to note that we obtained the same result from both the datasets. Some of the graphs for the *UCI* dataset can be seen on our website[4].

[4] http://sdl.ist.osaka-u.ac.jp/~higo/fse2014/

Table 2: Number of Method Pairs in Each Category

Category	*APACHE*	*UCI*
within-file	4,974	18,617
within-directory	13,162	24,722
within-project	559,592	147,181
across-project	14,162,007	14,723,451
total	14,739,735	14,913,971

5.1 Region "C"

Method pairs in Region "C" had the same name, but their *structural similarity* and *vocabulary similarity* were low. Manual investigation revealed that none of the selected pairs contains related methods. Hence, it does not seem worthwhile to detect them as functionally similar code. They quite often had highly abstract names such as "*get*" or "*execute*" or language-dependent names such as "*main*" or "*addActionListener*".

In the graph for *across-project* in Fig. 2, many method pairs having the same name are located near the bottom left corner. That is, their *structural similarity* and *vocabulary similarity* are low. On the other hand, some of the *same-name* method pairs are located near the top right corner. In order to ascertain the differences in characteristics between bottom-left method pairs and top-right method pairs, we analyzed the relation between the abstractness of their names, their *structural similarity*, and their *vocabulary similarity*. Figure 3 shows the result. For example, in Fig. 3(a), the left-most boxplot shows the distribution of *structural similarity* of *same-name* method pairs where there are five or fewer methods having the same name. This figure shows that the lower the number of methods that have the same name, the higher their *strucutral similarity* and *vocabulary similarity*.

Besides, Fig. 4 shows histogram representing frequency of structural and vocabulary similarities for the *same-name* and *different-name* method pairs for the category *across-project*. We can see that even most the *same-name* method pairs have low structural and vocabulary similarity. This result shows that the method name sameness of a given method pair does not necessarily indicate its high *structural similarity* or high *vocabulary similarity*.

5.2 Regions "A" and "E"

Method pairs in Region "A" have a high *structural similarity*, but they have low *vocabulary similarity* and different method names. Method pairs in Region "E" have high *structural similarity* and the same method names, but have a low *vocabulary similarity*.

In the category *across-project*, many method pairs included consecutive switch-case statements and consecutive if-else statements. In Java language, such implementations are often used in cases where we need to bifurcate a procedure into multiple branches. In other words, the reason their *structural similarity* was high was that they included language-dependent implementations. We were not able to find any other reason, such as that they had been created by copy-and-paste operations. Such code (repeated instructions) is occasionally regarded as false positives in clone detection [12]. There is even a clone detection technique that has a special function to avoid detecting repeated instructions as clones [34].

In the categories *within-project* and *within-directory*, many methods had similar procedure logic for different object types. For example, in project qpid, the following two files had methods whose names were "*construct*" (see Fig. 5):

Table 3: Number of Method Pairs in Each Region

Region	*APACHE*	*UCI*
A	45	161
B	149	355
C	29,591	37,047
D	229	598
E	82	80
F	98	176
G	472	1,918

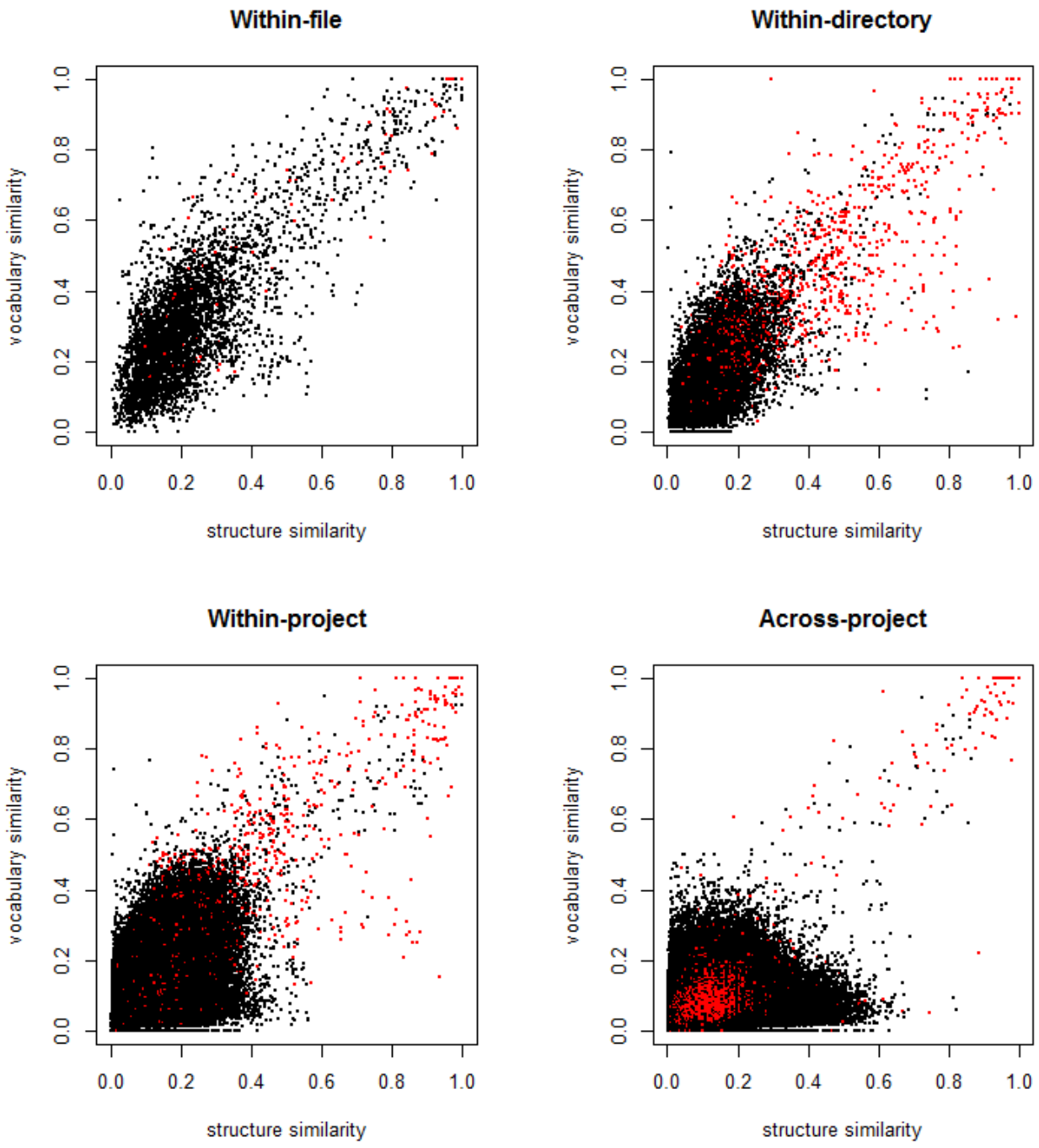
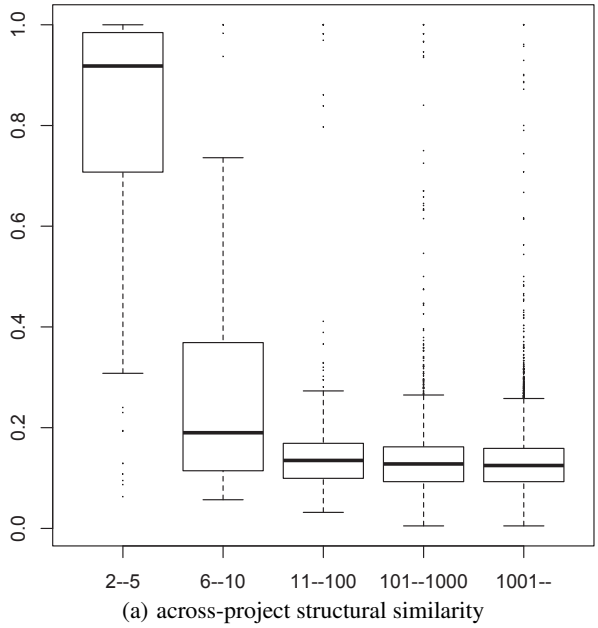


Figure 2: Overview of Three Types of Similarities for *APACHE* (Each dot represents a method pair. Each black dot is a method pair whose names are different and each red dot is a method pair whose names are the same)

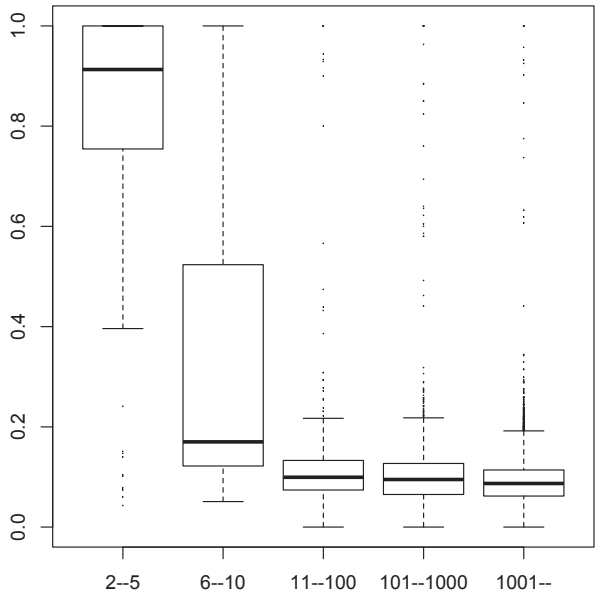
- messeging/codec/PropertiesConstructor.java, and
- transport/codec/AttachConstructor.java.

Their *structural* and *vocabulary similarities* were 0.83 and 0.22, respectively. Such method pairs are latent refactoring opportunities. However, refactoring them is not an easy task because they generally include small code fragments that are different from each other. Complicated operations such as the *Form Template Method* [7] are required to refactor them. We also found method pairs that included the language-dependent repeated code mentioned in the previous paragraph.

In the category *within-file*, there were only 14 method pairs whose *structural similarity* was high but whose *vocabulary similarity* was low. The lowest value of *vocabulary similarity* was 0.39. If two methods are defined in the same class, they can use the same private methods and private fields. That is, method pairs in the same class tend to have a higher *vocabulary similarity*. Figure 6 supports this conclusion.

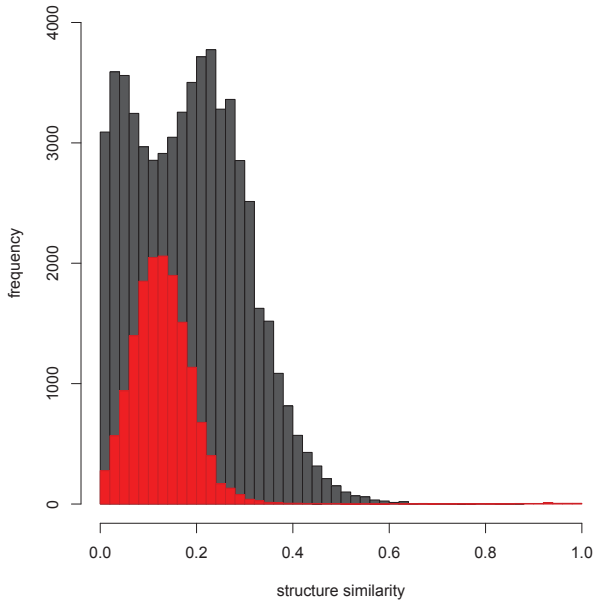


(a) across-project structural similarity

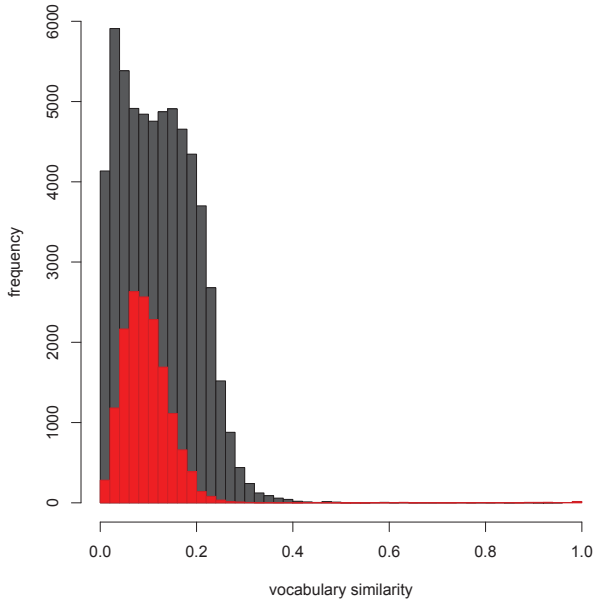


(b) across-project vocabulary similarity

Figure 3: Boxplot representing relationship *structural/vocabulary similarity* distribution and name abstractness for the category *across-project*. X-axis represents the degree of name abstractness. There are 5 levels of name abstractness.



(a) structural similarity



(b) vocabulary similarity

Figure 4: Histogram representing frequency of structural/vocabulary similarity for same-name/differnet-name method pairs for the category *across-project*. The black bars represent frequency of different-name method pairs and the red bars represent same-name ones.

5.3 Regions "B" and "F"

Method pairs in Region "B" have a high *vocabulary similarity*, but a low *structural similarity* and different method names. Pairs in Region "F" have a high *vocabulary*, but low *structural similarity* and the same method names.

In the category *across-project*, we found many cases of code reuse between different projects. After copying and pasting a code fragment, it was modified extensively (in many cases, new statements had been added to the pasted code). Such large modifications lowered the *structural similarity* between the original code and the copied code. However, user-defined names were not changed in such modifications. Consequently, their *vocabulary similarity* had been kept high. We could not find any different characteristics between pairs having the same names and different names.

In the categories *within-project* and *within-directory*, the structures of method pairs were partially similar to each other. They should be detected by token-based or string-based clone detection techniques such as CCFinder [20]. Some method pairs having the

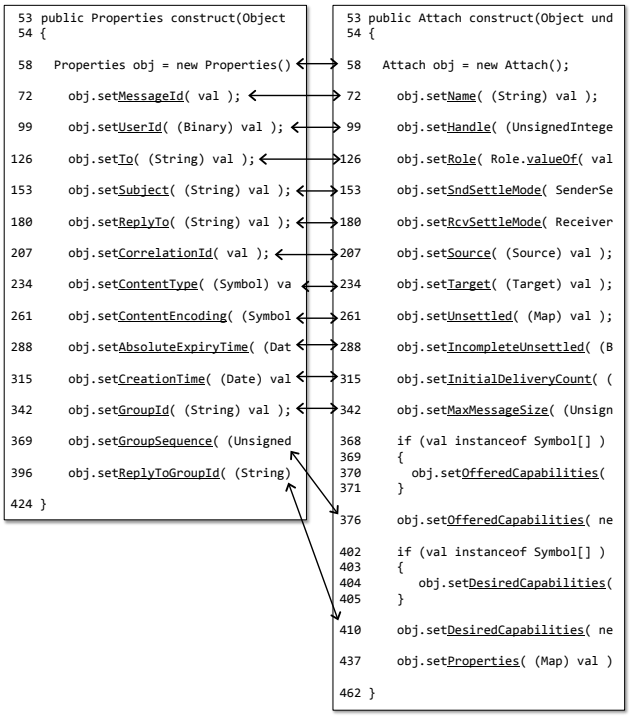


Figure 5: Method pair whose *structural similarity* is high but whose *vocabulary similarity* is low (Different variables are underlined. Bidirectional arrows show statement correspondences. Identical statements are omitted due to space limitations.)

same names are semantically the same procedure, even if their implementation ways are different. Some of them were overriding the same method in a common parent class. Some method pairs having different names implemented opposite procedures such as "*uncompress*" and "*compress*" or implemented related procedures such as logical AND and OR operators.

In the category *within-file*, methods can use the same resources such as private methods or private fields. This is because their *vocabulary similarity* tends to be higher. Figure 7 shows the distributions of *structural similarity* and *vocabulary similarity* in each category. This figure shows that *vocabulary similarity* in the category *within-file* stay higher than the other categories even if their *structural similarity* is not high.

5.4 Regions "D" and "G"

In none of the categories did we find false positives, regardless of the methods' name sameness. We also found that if both *structural similarity* and *vocabulary similarity* were 1.0, their method names were always the same.

In the category *across-project*, we found many examples of code reuses between different projects. In the category *within-file*, the method pairs seemed to be good opportunities for performing the *Extract Method* refactoring pattern. If both the similarities of a given method pair are 1.0, their difference exists only in data types; for example, one is a quicksort implementation for an "*int*" array and the other is also a quicksort implementation for a "*byte*" array.

In the categories *within-project* and *within-directory*, such method pairs can be latent opportunities that similar procedures are pulled up to common parent classes. However, methods that are exactly

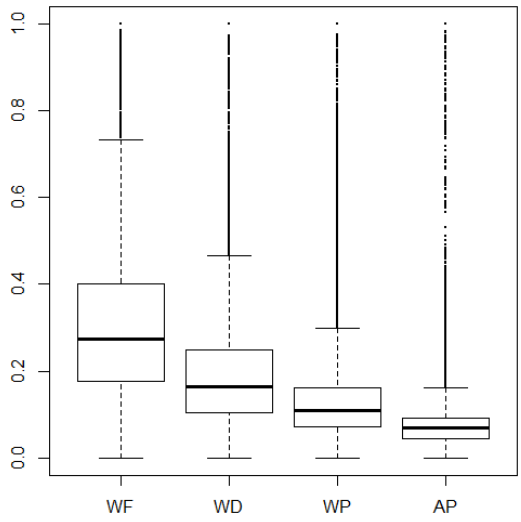


Figure 6: Vocabulary similarity in each category

the same are a minority. If a given method pair includes different statements, we need to use complicated modifications such as the *Form Template Method* to treat the differences. In addition, if two classes that include methods of a given pair do not have a common parent class, we first need to create it. If either of the two classes has an explicit parent class, which means it has an "*extends*" clause, the class hierarchy must be changed to create a new common parent class. However, creating a new class and changing an existing class hierarchy make up a large task and may be a design-level modification. Thus, we need to give careful consideration to it. In other words, if two classes already have a common parent class, we need less effort to refactor the method pair.

We investigated to what extent method pairs had common parent classes. In this investigation, "*java.lang.Object*" and the other classes in JDK were not treated as a common parent class. Table 4 shows the result. We can see that a considerable number of method pairs have common parent classes. The category *within-directory* has a higher rate of method pairs having a common parent class than the category *within-project*. Interestingly, however, if we consider only method pairs whose *structural similarity* and *vocabulary similarity* are 1.0, the category *within-project* has a higher rate. It is not a hard task to pull up method pairs if they are completely the same and have a common parent class.

6. TOWARD FUTURE RESEARCH

In this section, we discuss some directions for future research based on the investigation results.

6.1 Pulling up Similar Methods

In the categories *within-directory* and *within-project*, we found many pairs of functionally similar methods. However, most of them included different statements from each other. To promote refactoring of such method pairs, we need techniques to help refactoring.

Hotta et al. proposed a technique to identify the differences between a given pair of Java methods [15]. They leveraged program dependence graphs to detect non-duplicated statements, which should be kept in child classes in a case where we apply the *Form Template Method* refactoring pattern to a given pair of similar methods.

Krishnan and Tsantalis proposed a technique to identify a set of statements that should be extracted as a new method [25]. Clone

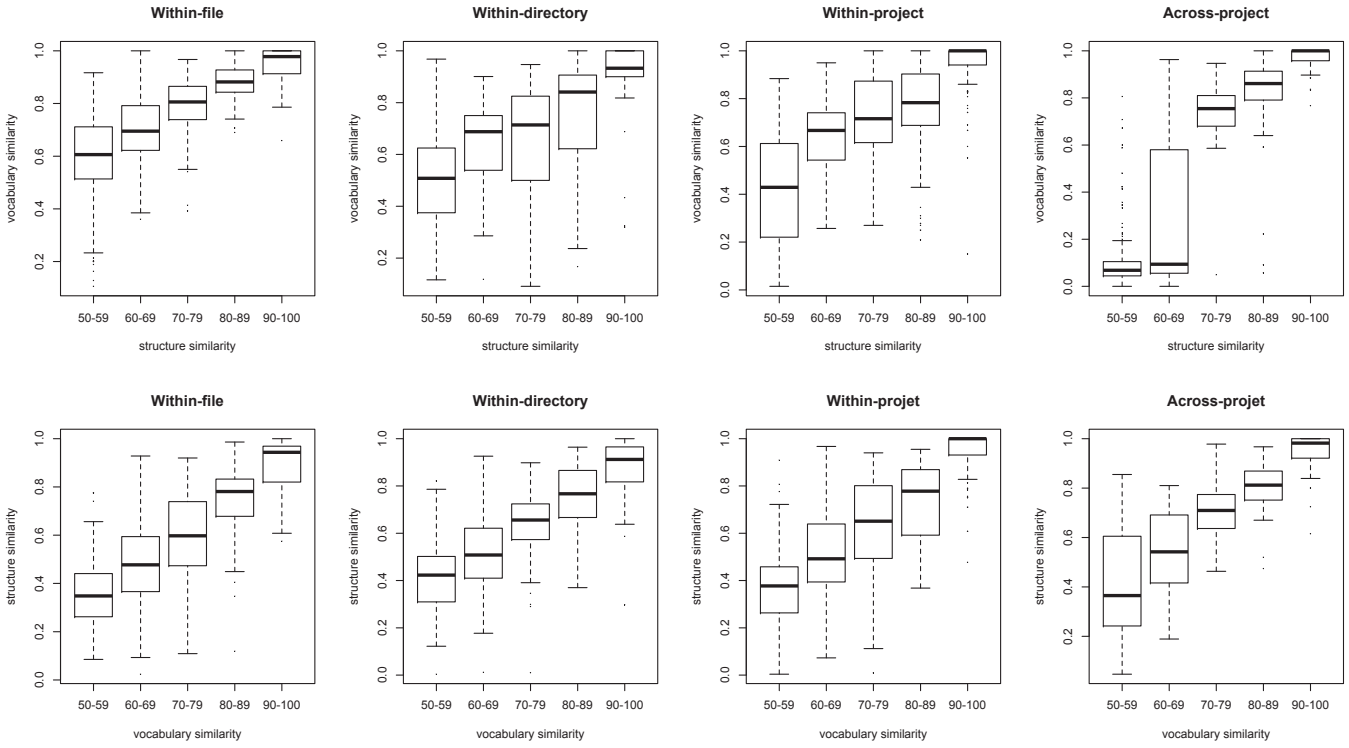


Figure 7: Structural-similarity-based and vocabulary-similarity-based boxplots

detection tools generally identify maximum duplications in source code as clones. However, maximum duplications often include various differences such as different variables, different literals, or different statements. Their technique identifies duplications that include a small number of differences. Clones detected by their technique are suited to refactoring, and developers can create easily reusable methods from detected clones.

Hotta's technique suggests method pairs where the *Form Template Method* pattern can be applied. However, humans may not think suggested method pairs should be refactored. If removing duplicate code is the primary goal of refactoring, it is worth supporting the deletion of duplicate code as far as possible, even by using complicated operations. On the other hand, there are many cases where developers do not want to change a class hierarchy or do not want to perform complicated refactoring operations.

Krishnan's technique suggests code fragments that can be refactored easily with some duplicate code remaining. His technique is intended for the *Extract Method* pattern. However, the same strategy should prove useful for other refactoring patterns such as *Pull Up Method*. There are probably many cases where developers want to perform simple refactorings that leave some duplicate code remaining, rather than complicated refactorings designed for removing all duplications. We need techniques for pulling up a chunk of duplicated code to a parent class with a small amount of effort.

Table 4: Number of method pairs where common parent classes exist. The numbers in parentheses are method pairs whose structural and vocabulary similarities are 1.

(a) *APACHE*

	within-directory	*within-project*
(A) all in D and G	124	257
(B) common parent	43 (5)	47 (19)
rate of (B) against (A)	0.34 (0.04)	0.18 (0.07)

(b) *UCI*

	within-directory	*within-project*
(A) all in D and G	443	1,375
(B) common parent	285 (53)	785 (336)
rate of (B) against (A)	0.64 (0.12)	0.57 (0.24)

6.2 Detecting Semantic Clones

Detecting semantic clones (type-4 clones) is a challenging research topic. Existing graph-based detection techniques can detect a part of semantic clones, such as a pair of iterative procedures: one is implemented using a for loop, and the other is implemented using a while loop [13, 22, 24]. However, their detection capabilities are not adequate. Kim et al. proposed a technique that leverages states of memory while a target program is executing [21]. This technique can detect semantic clones that are not detected by graph-based detection techniques. However, we need to prepare many test cases to use this technique. In addition, such a dynamic analysis suffers from scalability issues.

Within-directory and *within-project* method pairs in Region "F" told us that using vocabulary and method name is a good way to detect semantic clones. Such method pairs were often semantic clones in the experiment. However, *within-file* method pairs in two regions were often false positives because they tended to have a higher *vocabulary similarity* (see Fig. 6).

```
…
 17 package org.apache.activemq.filter;
…
 31 public abstract class ComparisonExpression
             extends BinaryExpression implements BooleanExpression {
…
355   @SuppressWarnings({ "rawtypes", "unchecked" })
356   protected Boolean compare(Comparable lv, Comparable rv) {
357     Class<? extends Comparable> lc = lv.getClass();
358     Class<? extends Comparable> rc = rv.getClass();
359     // If the the objects are not of the same type,
360     // try to convert up to allow the comparison.
361     if (lc != rc) {
362       try {
363         if (lc == Boolean.class) {
364           if (convertStringExpressions && rc == String.class) {
365             lv = Boolean.valueOf((String)lv).booleanValue();
366           } else {
367             return Boolean.FALSE;
368           }
369         } else if (lc == Byte.class) {
370           if (rc == Short.class) {
371             lv = Short.valueOf(((Number)lv).shortValue());
372           } else if (rc == Integer.class) {
373             lv = Integer.valueOf(((Number)lv).intValue());
374           } else if (rc == Long.class) {
375             lv = Long.valueOf(((Number)lv).longValue());
376           } else if (rc == Float.class) {
377             lv = new Float(((Number)lv).floatValue());
378           } else if (rc == Double.class) {
378             lv = new Double(((Number)lv).doubleValue());
379           } else if (convertStringExpressions && rc == String.class) {
380             rv = Byte.valueOf((String)rv);
381           } else {
382             return Boolean.FALSE;
383         }
…
```

(a) method "compare" in project "activemq"

```
…
 21 package org.apache.qpid.filter;
…
 33 public abstract class ComparisonExpression
             extends BinaryExpression implements BooleanExpression
 34 {
…
403   protected Boolean compare(Comparable lv, Comparable rv)
404   {
405     Class lc = lv.getClass();
406     Class rc = rv.getClass();
407     // If the the objects are not of the same type,
408     // try to convert up to allow the comparison.
409     if (lc != rc)
410     {
411       if (lc == Byte.class)
412       {
413         if (rc == Short.class)
414         {
415           lv = ((Number) lv).shortValue();
416         }
417         else if (rc == Integer.class)
418         {
419           lv = ((Number) lv).intValue();
420         }
421         else if (rc == Long.class)
422         {
423           lv = ((Number) lv).longValue();
424         }
425         else if (rc == Float.class)
426         {
426           lv = ((Number) lv).floatValue();
427         }
428         else if (rc == Double.class)
429         {
430           lv = ((Number) lv).doubleValue();
431         }
432         else
433         {
434           return Boolean.FALSE;
435         }
436       }
…
```

(b) method "compare" in project "qpid"

Figure 8: Source Code of the Vocabulary-Similar Method Pair whose Structural Similarity is the Lowest

6.3 Identifying Code Reuse between Different Projects

We found many instances of code reuse between different projects in Regions "B" and "D". After copying and pasting code from a different project, reused code was modified. If the modifications were small, the pair of original and reused code fell into Region "D". If large modifications were performed, the *structural similarity* decreased and it fell into Region "B". There were 126 and 283 method pairs whose *vocabulary similarity* was larger than 0.7 in *APACHE* and *UCI*, and all of them seemed to be examples of code reuse by copy-and-paste operations. Figure 8 shows a method pair whose vocabulary similarity is greater than 0.7 and whose *structural similarity* is the lowest (0.47). Although *structural similarity* is low, it is obvious that the pair was made by copying and pasting because the head parts of the two methods are quite similar to each other.

There are several approaches to detect clones between different projects. They can be classified into two categories, fine-grained detection [17, 23, 27, 39, 41] and unit-level detection[19, 35, 40].

- Fine-grained detections can identify duplications even if they are only small code chunks in source files. However, their scalability is inferior to unit-level detections. For example, Livieri et al. took two days to complete clone detection from 700 million lines of code using 80 personal computers [27].

- Unit-level detections can identify duplications only if whole units such as file, class, or method are duplicated. However, they have high scalability. For example, Ishihara et al. took less than two hours to complete clone detection from 360 million lines of code by using a single workstation [19].

To date, there has been no empirical study that has compared *across-project* clone detection results between fine-grained and unit-level detection. This research shows there is a risk of missing clones even if we use both fine-grained and unit-level detection techniques. Consequently, we need to develop new methodologies for detecting *across-project* code reuse by using code characteristics other than its structure.

7. THREATS TO VALIDITY

In order to relieve bias due to the dataset being used, we used two different datasets in this experiment. The experimental results, which are described in Section 5, were almost the same for the two datasets. Consequently, we can say that if we use another dataset in the future, the result will be almost the same. However, we used only a set of parameters where the *minimum length of token sequences* was 50 and the *minimum number of words* was 10 and both of the *structural similarity* and the *vocabulary similarity* were 0.7. If we use another set of parameters, we may obtain different tendencies in the result.

The category *within-directory* means that given two methods are in different files but in the same directory. Directories are generally nested, but the top-most directories in the source folder should be the deciding factor. Consequently, if we had counted for only the top-most directories for the category *within-directory*, we might have obtained a different result.

We classified method pairs into eight categories based on binary determinations on the three similarities. Then, we sampled 100 method pairs from each of the categories. In such a way, the degree of similarities on the *structural similarity* and *vocabulary similarity* may not be considered appropriately. A random sampling of method pairs based on the similarity values should be an appropriate way.

We used the longest common subsequence algorithm to measure the *structural similarity* among methods because it is popular and its computational complexity is not so high. However, there are various ways to detect structurally similar code [36, 38]. If we had used another way to measure the *structural similarity*, we would have obtain a different distribution of *structural similarity* among methods. Bellon et al. compared some techniques that detect structurally similar code [3].

The *UCI* dataset includes the projects of *Apache Software Foundation*. Consequently, The 500 projects extracted from *UCI* dataset may include projects included in *Apache* dataset.

We split camel/snake cases, performed stemming, and removed stop words in extracting vocabulary from source code. However, some of the words were not extracted appropriately due to reasons such as they were short names. To extract vocabulary more appropriately, it would be better to use Lawrie et al.'s method [26].

8. RELATED WORK

Tiark et al. conducted an experiment on type-3 clones[5] [44]. Their concern was what kinds of code characteristics contributed to type-3 clone detection. They revealed that, if a given clone pair had a similar word set in their identifiers, humans were not likely to reject it. Their result is similar to our result described in Section 5.4. However, they investigated only code that had been detected as clones. They investigated neither code where the vocabulary similarity was high nor code having the same signature.

Abebe et al. proposed using not only the structure of code but also its vocabulary for predicting fault-prone modules [1]. They confirmed that using vocabulary had improved the accuracy of prediction. In their experiment, predictions using a CK metrics suite were compared with ones using CK metrics and bad smell information of a vocabulary. The majority of cases using vocabulary with CK metrics improved prediction.

Bigger et al. investigated the vocabulary relationship between comments, identifier, and literal on 125 projects [4]. They found that 75% of words in the vocabulary appeared in identifiers. On the other hand, only a few words appeared only in comment or literal. Their investigation is a comparison between comment, identifier, and literal. They did not compare vocabulary between projects.

Marcus et al. proposed a class cohesion metric based on code comments and identifiers in code [30]. In their evaluation, they made two bug-prediction models: one was made from the proposed metric and existing structure-based cohesion metrics such as LCOM1[5]; the other was made only from existing metrics. Then, they compared the bug prediction accuracy of both the models and they confirmed that the proposed metric was useful for bug prediction.

Haiduc and Marcus investigated how many words in source code were domain terms [11]. Their investigation targets were six graph theory libraries, and they found that that 62% of words were domain terms. The result indicates that methods within a project or domain have the same words in their code. We did not investigate vocabulary similarity of same-domain software. However, our investigation result showed that methods within a project have a higher vocabulary similarity than ones across projects. Our investigation result showed the same trend as their investigation result.

Source code clustering is a promising technique for maintaining legacy code or software evolution. For example, clustering can be used for detecting source code that should be re-modularized [33, 45] or identifying abstract data types [9]. Maletic and Marcus showed that identifying similar modules in a software system was helpful in understanding it [28, 29]. They utilized *Latent Semantic Indexing* techniques to make similar module clusters. Such support reduces the developer's cost for finishing a given task when developing or maintaining systems, and a developer can finish the task better than without support.

Different software programs use different words even if they include the same processing [10]. This is known as the vocabulary problem, which states that "no single word can be chosen to describe a programming concept in the best way" [8]. Bajracharya et al. developed a system to automatically learn how APIs can be used [2]. They assumed that source code using the same APIs implemented similar processing contents even if the source code used different user identifiers such as variable names. They then developed a technique called *Structural Semantic Indexing*. In addition, there are methods that automatically identify a set of words related to one another even if they are not related as English words [42, 46].

Corazza et al. proposed a software clustering technique using vocabulary information [6]. They classified vocabulary into six categories: class name, field name, method name, parameter name, comments, and statements. They gave different weights to different categories, and clusters are made by weighted vocabulary similarities. They showed that vocabulary-based clustering was more accurate than structure-based clustering in the context of remodularization in their experiment.

Hotta et al. compared fine-grained and unit-level clone detections [16]. They developed a unit-level detection tool that detects similar blocks such as an `if` statement or `for` statements in Java source code. They evaluated the tool by using the four Java software projects included in Bellon's benchmark [3]. They revealed the unit-level detection had enough accuracy, but did not have high recall compared to fine-grained detectors. They conducted experiments for each of the projects and they did not target *across-project* clones.

9. CONCLUSION

In this paper, we investigated the relationships between *structural similarity*, *vocabulary similarity*, and *method name similarity* of Java methods with consideration of their positional relationship, which has four categories: *within-file*, *within-directory*, *within-project*, and *across-project*. Our experimental targets were two different sets of open source projects. For each of the datasets, we measured the three similarities on approximately 14 million method pairs.

As a result, we found the following. (1) Method names do not always reflect functional code similarity. If there are a small number of methods that have a given name, the methods are likely to include functionally similar code. (2) Existing hash-based clone detection techniques at the file-level, method-level and block-level miss many instances of copy-and-pasted code between different projects. (3) In cases where we use *structural similarity* for detecting similar code, we obtain many false positives. However, most of the false positives are avoidable by using *vocabulary similarity* in addition to *structural similarity*. (4) Using *vocabulary similarity* for detecting similar code is not suitable for method pairs in the same file because such method pairs use many of the same program elements such as private methods or private fields. Their high *vocabulary similarity* is due to using the same program elements, not due to using the same words.

We also showed some directions for future research based on the experimental results. They include the following: (A) techniques for pulling up similar methods to the common parent classes, (B) detecting semantic clones, and (C) identifying code reuse between different projects.

10. ACKNOWLEDGMENTS

This study was supported by a Grant-in-Aid for Scientific Research (S) (25220003), a Grant-in-Aid for Exploratory Research (24650011) from the Japan Society for the Promotion of Science, and a Grant-in-Aid for Young Scientists (A) (24680002) from the Ministry of Education, Culture, Sports, Science and Technology.

[5]type-3 clones are duplicate code that include gapped lines.

11. REFERENCES

[1] S. L. Abebe, V. Arnaoudova, G. Antoniol, and Y. Gueheneuc. Can Lexicon Bad Smells Improve Fault Prediction. In *Proceedings of the 19th Working Conference on Reverse Engineering*, pages 235–244, 2012.

[2] S. K. Bajracharya, J. Ossher, and C. V. Lopes. Leveraging Usage Similarity for Effective Retrieval of Examples in Code Repositories. In *Proceedings of the 18th International Symposium on Foundations of Software Engineering*, pages 157–166, 2010.

[3] S. Bellon, R. Koschke, G. Antoniol, J. Krinke, and E. Merlo. Comparison and Evaluation of Clone Detection Tools. *IEEE Transactions on Software Engineering*, 33(9):577–591, 2007.

[4] L. R. Biggers, B. P. Eddy, N. A. Kraft, and L. H. Etzkorn. Toward a Metrics Suite for Source Code Lexicons. In *Proceedings of the 27th International Conference on Software Maintenance*, pages 492–495, 2011.

[5] S. R. Chidamber and C. F. Kemerer. A Metrics Suite for Object Oriented Design. *IEEE Transactions on Software Engineering*, 20(6):476–493, 1994.

[6] A. Corazza, S. D. Martino, V. Maggio, and G. Scanniello. Investigating the Use of Lexical Information for Software System Clustering. In *Proceedings of the 15th European Conference on Software Maintenance and Reengineering*, pages 35–44, 2011.

[7] M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[8] G. W. Furnas, T. K. Landauer, L. M. Gomez, and S. T. Dumais. The Vocabulary Problem in Human-system Communication. *Communications of the ACM*, 30(11):964–971, 1987.

[9] J. Girard and R. Koschke. A Comparison of Abstract Data Types and Objects Recovery Techniques. *Science of Computer Programming*, 36(2–3):149–181, 2000.

[10] M. Grechanik, C. Fu, Q. Xie, C. McMillan, D. Poshyvanyk, and C. Cumby. A Search Engine for Finding Highly Relevant Applications. In *Proceedings of the 32nd International Conference on Software Engineering*, pages 475–484, 2010.

[11] S. Haiduc and A. Marcus. On the Use of Domain Terms in Source Code. In *Proceedings of the 16th International Conference on Program Comprehension*, pages 113–122, 2008.

[12] Y. Higo, T. Kamiya, S. Kusumoto, and K. Inoue. Method and Implementation for Investigating Code Clones in a Software System. *Information and Software Technology*, 49(9-10):985–998, 2007.

[13] Y. Higo and S. Kusumoto. Code Clone Detection on Specialized PDGs with Heuristics. In *Proceedings of the 15th European Conference on Software Maintenance and Reengineering*, pages 75–84, 2011.

[14] Y. Higo, S. Kusumoto, and K. Inoue. A Metric-based Approach to Identifying Refactoring Opportunities for Merging Code Clones in a Java Software System. *Journal of Software: Maintenance and Evolution*, 20(6):435–461, 2008.

[15] K. Hotta, Y. Higo, and S. Kusumoto. Identifying, Tailoring, and Suggesting Form Template Method Refactoring Opportunities with Program Dependence Graph. In *Proceedings of the 16th European Conference on Software Maintenance and Reengineering*, pages 53–62, 2012.

[16] K. Hotta, Y. Higo, and S. Kusumoto. How Accurate Is Coarse-grained Clone Detection?: Comparison with Fine-grained Detectors. In *Proceedings of the 8th International Workshop on Software Clones*, pages 1–18, 2014.

[17] B. Hummel, E. Juergens, L. Heinemann, and M. Conradt. Index-based Code Clone Detection: Incremental, Distributed, Scalable. In *Proceedings of the International Conference on Software Maintenance*, pages 1–9, 2010.

[18] K. Inoue, R. Yokomori, T. Yamamoto, M. Matsushita, and S. Kusumoto. Ranking Significance of Software Components Based on Use Relations. *IEEE Transactions on Software Engineering*, 31(3):213–225, 2005.

[19] T. Ishihara, K. Hotta, Y. Higo, H. Igaki, and S. Kusumoto. Inter-Project Functional Clone Detection Toward Building Libraries - An Empirical Study on 13,000 Projects. In *Proceedings of the 19th Working Conference on Reverse Engineering*, pages 387–391, 2012.

[20] T. Kamiya, S. Kusumoto, and K. Inoue. CCFinder: A Multilinguistic Token-based Code Clone Detection System for Large Scale Source Code. *IEEE Transactions on Software Engineering*, 28(7):654–670, 2002.

[21] H. Kim, Y. Jung, S. Kim, and K. Yi. MeCC: Memory Comparison-based Clone Detector. In *Proceedings of the 33rd International Conference on Software Engineering*, pages 301–310, 2011.

[22] R. Komondoor and S. Horwitz. Using Slicing to Identify Duplication in Source Code. In *Proceedings of the 8th International Symposium on Static Analysis*, pages 40–56, 2001.

[23] R. Koschke. Large-scale Inter-system Clone Detection Using Suffix Trees and Hashing. *Journal of Software: Evolution and Process*, pages n/a–n/a, 2013.

[24] J. Krinke. Identifying Similar Code with Program Dependence Graphs. In *Proceedings of the 8th Working Conference on Reverse Engineering*, pages 301–309, 2001.

[25] G. P. Krishnan and N. Tsantalis. Unification and Refactoring of Clones. In *Proceedings of the International Conference on Software Maintenace, Reengineering and Reverse Engineering*, pages 104–113, 2014.

[26] D. Lawrie, D. Binkley, and C. Morrell. Normalizing Source Code Vocabulary. In *Proceedings of the 17th Working Conference on Reverse Engineering*, pages 3–12, 2010.

[27] S. Livieri, Y. Higo, M. Matushita, and K. Inoue. Very-Large Scale Code Clone Analysis and Visualization of Open Source Programs Using Distributed CCFinder: D-CCFinder. In *Proceedings of the 29th international conference on Software Engineering*, pages 106–115, 2007.

[28] J. I. Maletic and A. Marcus. Supporting Program Comprehension Using Semantic and Structural information. In *Proceedings of the 23rd International Conference on Software Engineering*, pages 103–112, 2001.

[29] A. Marcus and J. I. Maletic. Identification of High-Level Concept Clones in Source Code. In *Proceedings of the 16th international conference on Automated software engineering*, pages 107–114, 2001.

[30] A. Marcus, D. Poshyvanyk, and R. Ferenc. Using the Conceptual Cohesion of Classes for Fault Prediction in Object-Oriented Systems. *IEEE Transactions on Software Engineering*, 34(2):287–300, 2008.

[31] J. Mayrand, C. Leblanc, and E. Merlo. Experiment on the Automatic Detection of Function Clones in a Software System Using Metrics. In *Proceedings of the 1996 International Conference on Software Maintenance*, pages

244–253, 1996.

[32] C. McMillan, M. Grechanik, D. Poshyvanyk, Q. Xie, and C. Fu. Portfolio: Finding Relevant Functions and Their Usage. In *Proceedings of the 33rd International Conference on Software Engineering*, pages 111–120, 2011.

[33] H. A. Müller, M. A. Orgun, S. R. Tilley, and J. S. Uhl. A Reverse-engineering Approach to Subsystem Structure Identification. *Journal of Software Maintenance: Research and Practice*, 5(4):181–204, 1993.

[34] H. Murakami, K. Hotta, Y. Higo, H. Igaki, and S. Kusumoto. Folding Repeated Instructions for Improving Token-Based Code Clone Detection. In *Proceedings of the 12th International Working Conference on Source Code Analysis and Manipulation*, pages 64–73, 2012.

[35] J. Ossher, H. Sajnani, and C. Lopes. File Cloning in Open Source Java Projects: The Good, The Bad, and The Ugly. In *Proceedings of the 27th International Conference on Software Maintenance*, pages 283–292, 2011.

[36] D. Rattan, R. Bhatia, and M. Singh. Software Clone Detection: A Systematic Review. *Information and Software Technology*, 55(7):1165–1199, 2013.

[37] C. K. Roy and J. R. Cordy. NICAD: Accurate Detection of Near-Miss Intentional Clones Using Flexible Pretty-Printing and Code Normalization. In *Proceedings of the 2008 The 16th IEEE International Conference on Program Comprehension*, pages 172–181, 2008.

[38] C. K. Roy, J. R. Cordy, and R. Koschke. Comparison and Evaluation of Code Clone Detection Techniques and Tools: A Qualitative Approach. *Science of Computer Programming*, 74(7):470–495, 2009.

[39] H. Sajnani and C. Lopes. A Parallel and Efficient Approach to Large Scale Clone Detection. In *Proceedings of the 7th International Workshop on Software Clones*, pages 46–52, May 2013.

[40] Y. Sasaki, T. Yamamoto, Y. Hayase, and K. Inoue. Finding File Clones in FreeBSD Ports Collection. In *Proceedings of the 7th Working Conference on Mingin Software Repositories*, pages 102–105, 2010.

[41] W. Shang, B. Adams, and A. E. Hassan. An Experience Report on Scaling Tools for Mining Software Repositories using MapReduce. In *Proceedings of the international conference on Automated software engineering*, pages 275–284, 2010.

[42] L. Tan, Y. Zhou, and Y. Padioleau. aComment: Mining Annotations from Comments and Code to Detect Interrupt Related Concurrency Bugs. In *Proceedings of the 33rd International Conference on Software Engineering*, pages 11–20, 2011.

[43] P. Tan, M. Steinbach, and V. Kumar. *Introduction to Data Mining, (First Edition)*. Addison-Wesley Longman Publishing Co., Inc., 2005.

[44] R. Tiarks, R. Koschke, and R. Falke. An Extended Assessment of Type-3 Clones As Detected by State-of-the-art Tools. *Software Quality Control*, 19(2):295–331, 2011.

[45] T. A. Wiggerts. Using Clustering Algorithms in Legacy Systems Remodularization. In *Proceedings of the 4th Working Conference on Reverse Engineering*, pages 33–43, 1997.

[46] J. Yang and L. Tan. Inferring Semantically Related Words from Software Context. In *Proceedings of the Working Conference on Mining Software Repositories*, pages 161–170, 2012.