Reusing Reused Code

Tomoya Ishihara, Keisuke Hotta, Yoshiki Higo, Shinji Kusumoto
Graduate School of Information Science and Technology, Osaka University
1-5, Yamadaoka, Suita, Osaka, 565-0871, Japan
{t-ishihr, k-hotta, higo, kusumoto}@ist.osaka-u.ac.jp

***Abstract*—Although source code search systems are well known as being helpful to reuse source code, they have an issue that they often suggest larger code than what users actually need. This is because they suggest code based on the structure of programming languages such as files or classes. In this paper, we propose a new code search technique that considers past reuse. In the proposed technique, code are suggested at the unit of past reuse. The proposed technique detects reused code by using a fine-grained code clone detection technique. We conducted an experiment to compare the proposed technique with an existing technique. The result shows that the proposed technique helps more effectively to reuse code than the existing technique.**

***Index Terms*—code search; code clone; source code reuse;**

I. Introduction

In software development, it is beneficial to reuse existing source code. If developers reuse existing source code adequately, software development efficiency rises because they do not need to implement new functions. Besides, if they reuse source code that has already been well tested, software obtains high reliability easily. Consequently, it is useful to reuse code in software development. Currently, many researchers have proposed systems helping source code reuse.

One of the systems helping code reuse is code search systems [1]–[3]. Code search systems suggest code related to queries that users have input. They suggest code in the order of own measures. Code search systems do not require complex procedures for users. Users only need to decide queries when they use code search systems.

However, existing code search systems have an issue. They suggest source code including extra functionalities that users do not need. Hence, users must take costs to identify functions that they actually need. This issue is caused because they suggest source code based on structural unit of programming languages. In particular, this issue often occurs when users need a single or small functionality because most existing systems suggest source code on class or file unit. Users do not always require functionalities of the same unit or the same abstraction level. Sometimes users require a piece of a code fragment, which consists of several lines of code, and other times they need code forming the whole classes. Therefore, it is necessary for the systems to suggest source code with different sizes and different abstraction levels depending on users' requirements.

In this research, we propose a technique that detects reusable code that has been actually reused in the past and suggests it to users. That is to say, conventional search engines return entire PL units (methods, classes, files) while our approach returns code snippets that were duplicated. Also, our approach can suggest reused code even if they do not have API invocations unlike the conventional techniques [2]. The presence of reused code means that someone needed and reused the code in the past. Hence, authors think that code reused in the past will meet requirements of reuse in the future. We adopt a code clone detection technique for detecting reused code fragments [4]. One reason why code clones occur in source code is code reuse so that detecting code clones means detecting past code reuse.

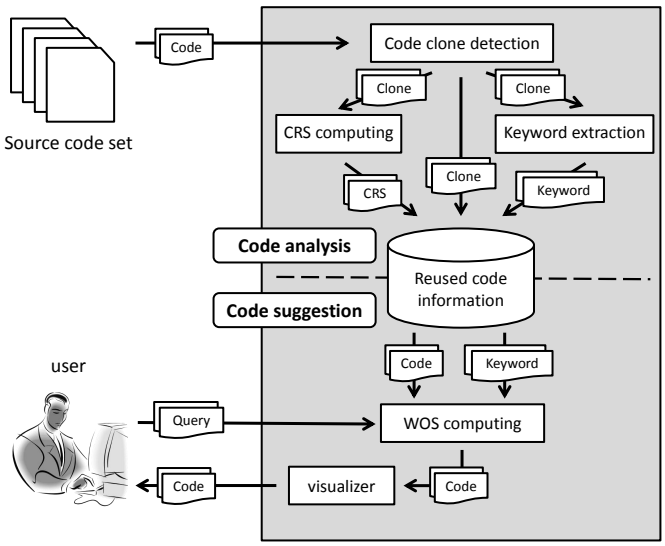

Figure 1. overview of the proposed technique

Contributions of this paper are as follows.

- We propose a new technique that suggests reusable code that has been reused. In the proposed technique, only reused code is suggested to users.
- We have conducted a small experiment and confirmed the usefulness of it.

II. Code Suggestion Architecture

A. Overview

In this research, we propose a technique that suggests source code based on its past reuse in order to provide reuse opportunities demanding on sizes and abstraction levels of users' requirements. Figure 1 shows an overview of the proposed technique. The proposed technique analyzes target source files and stores reused code into a database. If users input a query into a system, they obtain reused code that are related to the query.

The proposed technique consists of a **code analysis procedure** and a **code suggestion procedure**. In the code analysis procedure, the proposed technique detects code clones to collect a set of code fragments to be suggested to users. In the code suggestion procedure, the proposed technique suggests code fragments that are related to queries that users have input. Before the code suggestion procedure process users' queries, a database of reused code must have been built by performing the code analysis procedure. The code suggestion procedure is performed for every users' query meanwhile the code analysis procedure is performed only one time to build a database of reused code information.

B. Code Analysis Procedure

The code analysis procedure consists of the following steps.

STEP1: detecting code clones to collect a set of code fragments reused in the past.

STEP2: computing a score for every reused code fragment based on the caller-callee relationship.

STEP3: extracting keywords in the reused code fragments.

In the proposed technique, only the code fragments reused in the past are suggested to users. In order to identify past reuse, we adopt a code clone detection technique. One reason why code clones occur in source code is code reuse such as copy and paste operations. Hence, we can find code fragments that have been reused by detecting code clones from a set of a large amount of source files.

In addition, the proposed technique computes a score for each code fragment in order to suggest code that are suitable for users' requirements. In this research, we use *Component Rank* technique [1] to compute scores for code fragments. *Component Rank* technique computes a score for each function based on the caller-callee relationship. A score for a code fragment is the same as the score for the function including the code fragment. It computes a score independently on queries that users input because it uses static information of function calls. Also, the proposed technique considers the number of past reuse in computing scores. Code fragments that have been reused many times have high scores.

C. Code Suggestion Procedure

The code suggestion procedure includes the following steps.

STEP1: finding code fragments related to queries that users have input from the database.

STEP2: ranking code fragments based on their scores and strength of relevance to queries

STEP3: suggesting code fragments in the ranking order.

In this research, we use two scores, *Component Rank Score (CRS)* and *Word Occurrence Score (WOS)*. *CRS* represents scores computed by using *Component Rank* technique. *WOS* represents strength of relationship between code fragments and the queries that users have input. *WOS* becomes high if queries match the keywords that sufficiently indicate a functionality of a code fragment. The proposed technique dynamically computes *WOS* each time users submit queries while it computes *CRS* beforehand at the code analysis procedure. Finally, the proposed technique sorts code fragments suggested to users in the order.

III. Details of Code Suggestion

A. Code Clone Detection

In this research, we need to detect small code clones in order to suggest code fragments on a variety of sizes and abstraction levels. If we use a fine-grained detection technique, we can detect both small code clones such as several lines of code and large ones such as the whole classes. However, if we use coarse-grained detection technique such as file clone detection technique, we can detect only large code clones. We also need to select a scalable code clone detection technique because the proposed technique needs to detect code clones from a large amount of source files. Consequently, we adopt an index- and statement-based code clone detection [4]. This technique detects code clones through the following steps.

STEP1: extracting all the statements from every method as a sequence.

STEP2: computing a hash value for every p consecutive statements. Value p is specified by users in advance.

STEP3: detecting subsequences whose hash sequences are the same as code clones.

The proposed technique realizes scalable code clone detections by using hash-based code matching.

B. Component Rank

The proposed technique computes a *CRS* score for every code fragment by applying a variant of *Component Rank(CR)* [1]. *CR* technique computes a score for every function based on the caller-callee relationship, by applying a variant of *PageRank*, which is used to measure the relative importance of web sites. It determines a score based on two concepts:

- functions called in many other functions are significant.
- functions called in significant functions are significant.

Also, *CR* technique groups similar functions. *CR* technique redefines the score of the function by summing up scores of functions that are similar to it.

In this research, the proposed technique computes a score for every method by using *CR* algorithm in order to compute a score for every code clone. A score of a given code clone is defined as the score of the method including the code clone.

C. Keyword Extraction

The proposed technique extracts keywords from code fragments. Hence, the proposed technique does not extract any keywords from code fragments that have not been reused and does not suggest them to users. We extract keywords from identifiers such as variable names, method names, class names and Javadoc comments attached to methods and classes. Also, the proposed technique uses the natural language processing such as stemming and eliminating stop words. Finally, the proposed technique stores keywords into a database.

D. Measuring Relationship between Queries and Keywords

We select *TF-IDF* technique as a measure of *WOS*. *TF-IDF* technique computes a score for every term in all documents based on two metrics, *TF* and *IDF*. A *TF* represents a term frequency of a certain document. A *IDF* represents an inverse document frequency including a certain term. *TF-IDF* technique is commonly used in search systems into which users input queries.

In this research, a keyword corresponds to a term and a code fragment corresponds to a document. The proposed technique computes *WOS* by using the following formula.

$$W_{wos}(i,j) = \frac{t_{i,j}}{|T_i|} \times \log \frac{|D|}{d_i} \quad (1)$$

- $t_{i,j}$ represents the number of frequency of term i in code fragment j
- T_i represents a set of the terms i in all the code fragments
- D represents a set of all the code fragments
- d_i represents the number of documents including term i
- $|S|$ represents the number of elements included a set S

If queries match keywords extracted only from a small number of code fragments, *WOS* becomes high.

E. Merging Scores

The proposed technique derives relative importance among code fragments based on *CRS* and *WOS*. Code fragments are suggested to users in the order of the relative importance.

F. Implementation

It is necessary that the proposed technique quickly detects code clones from a large amount of source files and also quickly computes a score of code clones. Hence, we implemented some heuristics in the proposed technique. For example, existing systems have computed a score of each class by using *Component Rank* algorithm. On the other hand, the proposed technique needs to compute a score for every method. The proposed technique needs much more time to compute scores of methods because there are considerably more methods than classes in the source files. Hence, the proposed technique computes a score of every method approximately by distinguishing between method calls within a project and method calls across projects. This approximate calculation is widely used [5].

IV. Experiment

A. Experimental Setup

In order to evaluate the proposed technique, we have implemented a prototype based on the proposed technique. In this section, we called this prototype P. We applied it to target source files and created a database. In this experiment, we used the source files that had been used in the evaluation of *SPARS* as our target [1]. The target source files consist of about 190,000 Java source files, which is about 400 projects.

In addition, we have built the following two variants of the prototypes for comparison.

- A variant in which the proposed technique was implemented except considering the number of past reuse. For example, even code that has been reused more than 10 times were handled equally to the one that has been reused only twice. We called this variant V_1.
- A variant which suggested source code based on the structure of programming language without code clone detection. This variant suggested code at method unit. We called this variant V_2.

The two variants of the prototypes also were applied to the target source files and created databases.

B. Methodology

In this research, 6 participants used either of 3 tools for each given task. Firstly, they took a lecture by the authors for understanding given tasks and for learning how to use the tools. After the lecture, each of them independently did the tasks. For each task, they considered keywords themselves. If they did not find reusable code fitting to the given tasks with a keyword, they searched code with another keyword freely. In the experiment, we did not restrict the number of searching code with keywords because we search reusable code by freely changing keywords in the real situation in code reuse. The time limit of each task set to 20 minutes. Participants timed their code reuse, which was from the beginning of searching code to the completion of the task.

Three participants were master's course students and the other 3 participants were Ph.D students of computer science at Osaka University. All the participants had experiences of Java programming more than one year. The average was 2.83 years. They had no industry experiences. They were divided into 3 groups. A pair of a master's course and a Ph.D students belonged to each group. Each participant was given 9 tasks. Each group used the tool showed in Table I to complete each task.

C. Tasks

We used tasks that had been used to evaluate *SPARS* [1]. We selected 9 tasks that could be easily executed out of the 10 tasks. Also, participants determined difficulty of the given tasks into 3 levels in the experiment. Each difficulty level has 3 tasks, respectively. Table II shows a list of the tasks used in this experiment.

D. Result

Table III shows the result of the experiment. The second, third and fourth rows in the table mean the total times that two participants took to complete each of the given tasks. After the participants had finished all the tasks, we confirmed that all the tasks were executed correctly. If a certain participant

TABLE I
THE TOOL THAT EACH GROUP USED TO COMPLETE EACH TASK

	task1,2,3	task4,5,6	task7,8,9
group1	P	V_1	V_2
group2	V_2	P	V_1
group3	V_1	V_2	P

TABLE III
THE TOTAL TIMES THAT PARTICIPANTS TOOK TO FINISH EACH THE GIVEN TASK

	task1	task2	task3	task4	task5	task6	task7	task8	task9
P	1,091	1,393	2,400	2,400	1,525	848	1,360	1,529	853
V_1	1,309	1,109	2,400	1,994	367	1,286	2,400	1,890	2,011
V_2	365	925	2,400	1,508	689	954	1,930	2,400	240
difficulty	easy	easy	difficult	difficult	easy	medium	medium	difficult	medium
best tool	V_2	V_2	-	V_2	V_1	P	P	P	V_2

```
157  protected String streamAsset(HttpServerRequeset request,
                    HttpServerResponse response)
158  {
159     //   Get Asset ID
160     int A_Asset_ID = WebUtil.getParameterAsInt(request, "Asset_ID");
                    :
                    :
260     ServletOutputStream out = response.getOutPutStream();
261     ZipOutputStream zip = new ZipOutputStream(out);     //Servlet out
262     zip.setMethod(ZipOutputStream.DEFLATED);
263     zip.setLevel(Deflater.BEST_COMPRESSION);
264     zip.setComment(readme);
265
266     //  Readme File
267     ZipEntry entry = new ZipEntry("readme.txt");
268     entry.setExtra(assetInfo);
269     zip.putNextEntry(entry);
270     zip.write(readme.getBytes(), 0, readme.getLength());
271     zip.closeEntry();
                    :
                    :
328     return null;
329  }
```

(a) output of the tools. P and V_1 suggested a highlighted part of the method. V_2 suggested the whole method.

```
FileInputStream in = new FileInputStream(input);
ZipOutputStream zip = new ZipOutputStream(output);  //Servlet
                                                    //out
zip.setMethod(ZipOutputStream.DEFLATED);
zip.setLevel(Deflater.BEST_COMPRESSION);

final byte[] buf = new byte[1024];
zip.putNextEntry(new ZipEntry(input.getName()));
int len;
while((len = in.read(buf)) > 0){
      zip.write(buf, 0, len);
}
zip.closeEntry();
```

(b) code in which a certain participant implemented the given task

Figure 2. an example of code reuse in the experiment

could not complete the given task within the limited time, we regarded that the participant took 20 minutes to complete the task. The fifth row means the difficulty of each task. The sixth row means the best tool, with which participants took the shortest time to complete the task. For the task 3, the best tool was not selected because no participant completed the task within the limited time.

TABLE II
A LIST OF THE TASKS

task	requirement
task1	implement a quicksort algorithm
task2	implement a binary search algorithm
task3	build a Java applet that shows an analog clock
task4	build a Java applet that shows a textarea
task5	implement a functionality of random number generation
task6	implement functionalities of a stack such as push and pop
task7	implement a functionality that compresses a file or a directory with ZIP format
task8	implement a functionality that dumps a class file
task9	implement functionalities that read an input file and write the copy of the file through a stream

We compared the means of completion time for each technique by using Tukey's HSD test with significance level $\alpha = 0.05$. The result of the test showed that there were no differences among the means of completion time for the 3 techniques. Then, we investigated which technique made the completion time shortest for each task. As looking at the table III, we found that V_2 was the most effective tool for users on 4 tasks. However, compared to P, V_2 is the most effective for relatively easy tasks. On the other hand, P is the most effective for relatively difficult tasks.

This was caused by the difference of sizes of functionalities. Difficult tasks were implemented with a combination of multiple methods meanwhile easy tasks were implemented with a single method. Participants using P could make an effective combination of functionalities because P suggested only necessary parts of methods. On the other hand, because V_2 suggested the whole methods, participants using V_2 needed to find where they should reuse. Participants had difficulties to complete the task 4 because of a low experience of Java applet. Hence, this task was determined as difficult nevertheless it could be implemented with a single method. Figure 2 shows an example of code suggested by P, V_1 and V_2. P and V_1 suggested a highlighted part of the method meanwhile V_2 suggested the whole method that was about 200 lines of code. In order to improve efficiency to reuse code, it is necessary that systems help to reuse functionalities that users takes much time to implement. Consequently, P is useful to support reusing code that it is difficult for users to implement.

Also, comparing between P and V_1, the participants finished over half of all the tasks in a shorter time by using P than by using V_1. For example, P suggested the code in Figure 2 on the top if they input a query into it, "zip deflate". On the other hand, V_1 suggested the code at the 43rd from the top. This means that functionalities that users actually needed were suggested on the top by considering the number of past reuse.

V. Threats to Validity

Participants. We had 6 participants use the 3 tools and complete the given tasks. All the participants had experiences of Java programming more than one year. However, if there is a large difference of skills of Java programming among the participants, the difference of skills has possibilities to influence result in this experiment. In this experiment, 6 participants were divided into 3 groups. Each group consisted of 2 participants, a master's course student and a Ph.D student. Hence, the difference of skills among the groups should have been small.

Time pressure. The participants must have completed each task within 20 minutes. Hence, there was a possibility that the limited time might have pressured the participants and so some of them missed the code that should be reused and consequently spent more time to complete the task.

Target data set. The proposed technique decides the order of code suggestion considering the number of code clones. Which code becomes code clones is dependent on target data sets. Hence, it is possible that a result with another data set is different from the one in this experiment.

Implementation. In this research, in order to create a database quickly, some approximate calculations were implemented in the proposed technique. For example, the proposed technique computed a score of every method approximately by distinguishing between method calls within a project and method calls across projects. Hence, if the proposed technique computed a strict score of every method by using such as a high performance machine, there was a possibility that the result in this experiment varies.

VI. Related Work

Inoue at el. proposed *Component Rank(CR)*, which computes scores of functions based on the caller-callee relationship, and *Keyword Rank(KR)*, which gives different weights for each keyword based on its position in source code [1]. They also built a source code search system, *SPARS* [1], in which the above ranking algorithms were adopted. *CR* gives a high score to a function called in many other functions and a function called in functions whose scores are high. Also, it merges scores of similar functions and redefines their scores. Hence, scores of functions to which many functions are similar become high. *KR* gives scores to keywords based on their token types. If the types of a given keyword is significant, its score becomes high. For example, keywords included in method names or class names are significant.

McMillan et al. proposed a technique that computes scores of functions based on *Navigation Model(NM)* and *Association Model(AM)*. They built a source code search system, *Portfolio*, based on their proposed technique [3]. *NM* represents how programmers trace functions. It gives scores to functions based on the caller-callee relationship, by applying a variant of *PageRank*. *AM* represents the relationship among keywords. It computes strength of the keyword relationship by applying *Spreading Activation*. After users submit a search query, *Portfolio* shows a function call graph included suggested functions. Users can learn the usage of the functions by tracing functions with the graph and viewing their usage.

The two existing techniques are similar to the proposed technique in terms of computing a score of every function based on the caller-callee relationship. In addition, *SPARS* merges similar functions like the proposed technique. However, the two techniques suggest source code based on the structure of programming languages. On the other hand, the proposed technique suggests source code based on past reuse. Hence, particularly when users are going to reuse small functionalities, the proposed technique better than the two techniques in terms of suggesting only the functionality that users actually need.

Holmes et al. proposed an approach to help pragmatic reuse [6]. It enables developers to investigate code that they want to reuse and to integrate the code semi-automatically by using captured pragmatic reuse. Meanwhile the approach supports reuse from only a system given by users, the proposed technique suggests functionalities from a variety of systems.

VII. Conclusion

This paper proposed a new technique that suggested reusable code that had been reused in the past in order to meet the different requirements of users. The proposed technique calculates the number of past reuse for each code fragment and ranks the suggested code fragments by considering it. We have conducted an experiment that measured and compared the time when 6 participants completed the given task. They completed the given tasks by using 3 tools including one in which the proposed technique is implemented.

The result shows that the proposed technique helps more effectively to reuse code than a conventional technique. The result also shows that consideration of the number of reuse in the past contributes to the effective code reuse.

As future works, we are going to build a web-based code search system in which the proposed technique is fully implemented. At this time, the proposed technique considers not only the relevance to a task but also the ease of reuse. After building it, we are going to invite more participants and conduct large scale experiments. Also, we consider that we apply the proposed technique to the code completion. If users write code half way and execute code completion, the code that has been reused in the past is auto-completed immediately.

Acknowledgments

This study has been supported by Grants-in-Aid for Scientific Research (S) (25220003), Grant-in-Aid for Exploratory Research (24650011), and Grand-in-Aid for Young Scientists (A) (24680002) from the Japan Society for the Promotion of Science.

References

[1] 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*, vol. 31, no. 3, pp. 213–225, 2005.

[2] M. Grechanik, C. Fu, Q. Xie, C. McMillan, D. Poshyvanyk, and C. Cumby, "A search engine for finding highly relevant applications," in *Proc. of the 32nd International Conference on Software Engineering*, 2010, pp. 475–484.

[3] C. McMillan, M. Grechanik, D. Poshyvanyk, Q. Xie, and C. Fu, "Portfolio: Finding relevant functions and their usages," in *Proc. of the 33rd International Conference on Software Engineering*, 2011, pp. 111–120.

[4] B. Hummel, E. Jürgens, L. Heinemann, and M. Conradt, "Index-based code clone detection: incremental, distributed, scalable," in *Proc. of the 26th International Conference on Software Maintenance*, 2010, pp. 1–9.

[5] A. Z. Broder, R. Lempel, F. Maghoul, and J. Pedersen, "Efficient pagerank approximation via graph aggregation," *Information Retrieval*, vol. 9, no. 2, pp. 123–138, 2006.

[6] R. Holmes and R. J. Walker, "Systematizing pragmatic software reuse," *ACM Transactions on Software Engineering and Methodology*, vol. 21, no. 4, pp. 1–44, 2012.