How Compact Will My System Be? A Fully-Automated Way to Calculate LoC Reduced by Clone Refactoring

Tasuku Nakagawa, Yoshiki Higo, Junnosuke Matsumoto, and Shinji Kusumoto
Graduate School of Information Science and Technology
Osaka University
1-5 Yamadaoka, Suita, Osaka, Japan
{t-nakagw, higo, j-matumt, kusumoto}@ist.osaka-u.ac.jp

***Abstract*—A code clone (in short, clone) is a code fragment that is identical or similar to other code fragments in source code. The presence of clone is known as bad smell, which is phenomena of source code to be refactored. A motivation of refactoring (merging) clones is to reduce the size of source code. An existing study proposed a technique to estimate reduced lines of code by merging clones; however, there are two issues in the existing technique: (1) the existing technique does not consider the refactorability of clones in spite that it is difficult or even impossible to merge some clones due to the limitation of programming languages; (2) in the case that multiple clones are overlapping, the existing technique only considers one of them can be merged. Due to the above issues, estimated reducible LoC is occasionally different from the actual number. Consequently, in this research, we propose a new technique to calculate a reducible LoC. The proposed technique is free from the two issues, and it calculates a reducible LoC fully automatically. The proposed technique performs a loop processing of (a) detecting clones, (b) merging them, (c) compiling the edited source files, and (d) testing them. After finishing the loop, reducible LoC is calculated from the edited source files. This paper also includes comparison results of the proposed technique and the existing one. In the comparisons, we confirmed that a reducible LoC which was calculated with considering refactorability is 25% of a reducible LoC which was estimated without considering refactorability. We also confirmed that the proposed technique was able to merge clones that were not counted in the existing technique.**

***Keywords*-Software maintenance; code clone; refactoring;**

I. Introduction

Changes are occasionally (or even continuously) added to the source code after software systems have been released. A bunch of changes to the source code deteriorates its quality (e.g., collapsing its design, decreasing the readability), so that the maintenance cost gets more expensive [1], [2], [3]. The maintenance cost of the source code is often estimated from its size or complexity [3]. In the case of large-scale software systems, an enormous amount of money is required. If users of a software system can estimate its number, they should be able to decide whether they continue to use it or replace it with a new one [4].

A factor of deteriorating the quality of source code is the presence of clones. A clone means a code fragment that is identical or similar to other code fragments in the source code. Clones get involved in both software development and maintenance. The presence of clones makes the source code redundant so that inconsistencies in source code tend to happen unintentionally. Thus, from the perspective of maintainability of the source code, merging clones is important.

Merging clones is a well-known refactoring. Refactoring is defined as a set of operations to improve the internal structure of the source code without altering its external behavior [5]. *Extract Method* refactoring, which is one of the most often performed refactorings, is a set of operations to extract a code fragment in an existing method as a new method. If duplicated code fragments are extracted as a new method, the duplication is removed from the source code. Removing clones by refactoring makes it easier to keep consistencies in the source code because we do not have to put the same changes on duplicated code in multiple places. However, there is a possibility that refactoring itself introduces a new bug in the source code. Consequently, removing all clones without any special reason is not realistic: a reasonable indicator is required whether or not given clones should be refactored.

There is a study that estimates how many lines of code (LoC) is reduced by removing clones in the assumption that the reduced LoC is used as a primary indicator of the effects of clone refactoring [6]. This assumption means that, if a large number of LoC is reduced by clone refactoring, its effect is large and it is a sufficient motivation to remove the clones. The technique in the existing study utilizes the position information of clones (the path of the file including a given clone, start line and end line of the given clone). If multiple clones are overlapped with each other, the technique selects only one of them by using the greedy algorithm. The authors think there are two issues in the existing technique.

- The first issue is that the existing technique does not consider whether or not each clone can be removed. Some clones are difficult or even impossible to be merged as a single module such as class or method due to the limitation of programming languages. For example, if a clone includes a `return` statement, extracting it as a new method does not preserve the original behavior. In the original source code, the role of the `return` statement is getting out the existing method while in the refactored code, the role of the `return` statement is getting out the new extracted

method.

- The second issue is that in the case that clones are overlapping, the existing technique only considers merging one of the overlapped clones. However, even if two clones are overlapping, both of them may be able to be merged. Thus, an estimated reducible LoC may become a completely different number from the actual reducible LoC.

Both of the issues stem from the lack of checking refactorability of clones in the existing technique. However, it is not realistic to manually check whether or not each of the detected clones can be merged.

In this research, we propose a new technique to calculate reducible LoC. The proposed technique performs a loop of (1) detecting clones, removing a set of clones, (3) compiling and testing the edited source code as long as the LoC of the source code gets decreased by the refactorings. Due to its nature, the proposed technique is completely free from the above two issues. We implemented a software tool based on the proposed technique and applied it to several open source software systems. The purpose of the application is comparing the proposed technique to the existing one. As a result, we confirmed that the proposed technique calculated different reducible LoC values from the existing technique.

The remainder of this paper is organized as follows: we describe the preliminaries of this research in Section II; we explain the proposed technique in Section III; section IV is an introduction of the implemented tool; in Section V, we show and discuss the comparison results with the existing technique; in Section VI, we notice threats to validity of the proposed technique and the experiment; finally, we conclude this paper in Section VII.

II. Preliminaries

A. Clone Detection

A *clone* is a code fragment that is identical or similar to another code fragment [7]. A pair of code fragments that are identical or similar to each other is called *clone pair*. A set of code fragments in which any pair is a clone pair is called *clone set*. Reusing source code by copy-and-paste operations gets accelerated code implementation, but clones occurred to make it more difficult to maintain the source code [8]. For example, if a bug happens in a code fragment, developers need to check whether or not the similar bug exists in each of its clones. Consequently, it is important to understand where and how many clones exist in the source code.

So far, many clones detection techniques and tools have been proposed and developed [7], [9], [10]. However, there is no generic and strict definition of clones. Each clone detection technique has its unique definition and detects clones based on the definition. Thus, different clone detection techniques detect different clones from the same source code.

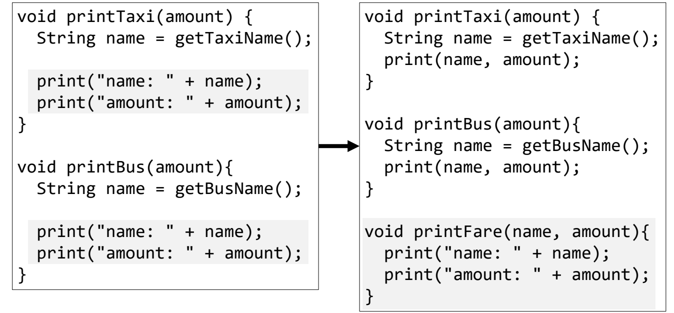


Fig. 1: Extract method refactoring

In this paper, we introduce a couple of popular detection techniques that are especially related to this research.

Token-based Clone Detection

This technique firstly translates the target source code to a sequence of tokens, then duplicated subsequences are detected as clones. Some normalizations are often applied before detecting subsequences. For example, user-defined identifiers such as variable names are replaced with special tokens. By applying this normalization, token-based detection technique can detect clones even if they include different variable names. The detection speed of token-based techniques are quite high, but detected clones are not necessarily suited for refactoring because token-based clones do not match with structural units of programming languages such as classes, methods or internal blocks in methods. CCFinder [7] and its successor CCFinderX [11] are well-known token-based detection tools.

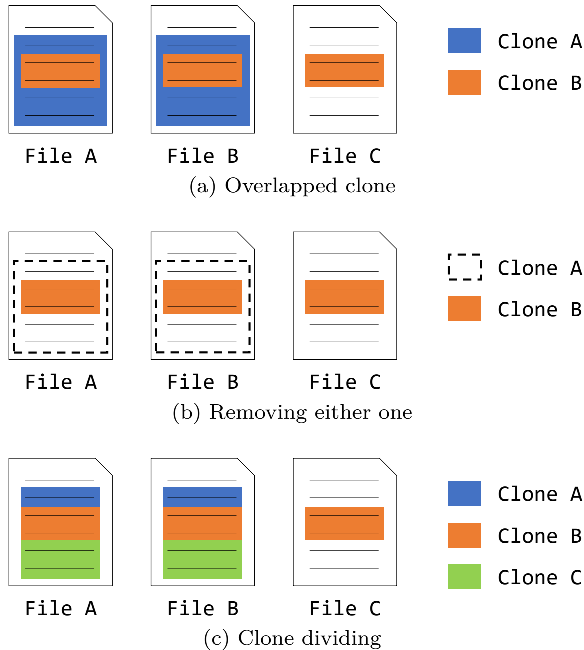
AST-based Clone Detection

In this technique, abstract syntax trees (in short, AST) are generated from the target source code. The isomorphic subtrees in the ASTs are identified as clones. Each detected clone corresponds to either structural unit of the programming language since it is a subtree in the ASTs. Consequently, clones detected by AST-based clone techniques have much better chemistry with refactoring than token-based clones. Deckard [9] is a well-known AST-based detection tool.

B. Refactoring Clone

Refactoring is known as a promising technique to improve the internal structure of source code without changing its external behavior [5]. Duplicated code (clones) is one of the typical bad smells (code to be refactored). There are a variety of ways to refactor (merge) clones. *Extract Method* refactoring is a simple yet well-known technique to remove clones. The original purpose of this refactoring pattern simplifies a long and/or complicated method by extracting a part of it as a new method. However, by applying *Extract Method* refactoring to clones, they can be removed. Figure 1 shows a simple example of *Extract Method* refactoring to two duplicated code fragments.

There are two major effects on applying *Extract Method* refactoring to a set of clones.



(a) Overlapped clone

(b) Removing either one

(c) Clone dividing

Fig. 2: Clone overlapping

- Code duplications are removed, which makes it easier to maintain consistency in the source code.
- LoC of the source code is changed; in many cases, LoC gets reduced. How many LoC is reduced depends on the size of a clone and the number of clones.

C. Clone Overlapping

In this paper, when two or more clones share one or more tokens, we say that "*the clones are overlapped with each other*". Figure 2a shows an example of overlapped clones. In this figure, clone A includes clone B. When clones are overlapped like this figure, we need to do special care to estimate reducible LoC. For example, the existing technique in literature [6] estimates reducible LoC in the case of removing clone A and in case of removing clone B separately, and then remove only either A or B, whose reducible LoC is larger than the other. Literature [6] also proposes a more aggressive approach, which is shown in Figure 2c. In this approach, the overlapped parts and the non-overlapped parts are separately considered for refactoring.

D. Reducible LoC

Herein, we introduce the definition of reducible LoC, which is proposed in the existing technique [6]. Herein, we assume that a clone set includes n code fragments, and each code fragment consists of C_{size} LoC. In a refactoring of merging the clone set, each clone is replaced with a method invocation, which is usually 1-line code. Consequently, reducible LoC C_{all} can be represented with the following formula.

$$C_{all} = n * C_{size} - n \quad (1)$$

In the case of Java, there is a 1-line code of method signature and open bracket "`{`" before a method body, and there is another 1-line code of close bracket "`}`". Thus, the LoC of a method becomes LoC of the method body plus two. An extracted code fragment (a clone) becomes a body of the new method. Consequently, LoC of the extracted method can be represented with the following formula.

$$M = C_{size} + 2 \quad (2)$$

By using the two formula 1 and 2, the LoC difference between the original code and refactored code can be represented with the following formula.

$$S = C_{all} - M = (n-1) * C_{size} - n + 2 \quad (3)$$

In the existing technique [6], the above formula is used to estimate reducible LoC by removing clones. As mentioned above, if multiple clones are overlapped with each other, the existing technique decides which clone is refactored by using the greedy algorithm. The literature [6] addresses that estimated reducible LoC by using the greedy algorithm is practical. However, the existing technique inherently includes the two issues that we mentioned in Subsection II-B.

III. Proposed Technique

We propose a new technique to calculate LoC that can be reduced by merging clones. The technique repeats (1) detecting clones, (2) editing source files, (3) compiling, and (4) testing. This technique enables users to obtain actual LoC that can be reduced by merging clones, not just estimating it. Moreover, this technique has an important feature: even if multiple clones are overlapped, the technique tries to merge each of them while the existing research [6] only considers one of the overlapped clones. Figure 3 shows an overview of the proposed technique.

The input of the technique is a set of source files. The output is a set of source files in which clones have been merged as much as possible and their reducible LoC. Please note that this technique does not merge clones if merging them does not reduce LoC of the source files.

The technique is composed of two processes shown below. Both the processes are performed fully automatically.

- Preprocessing source file changes
- Clone merging

A. Preprocessing source file changes

First, a new class is generated. The new class is used as a utility class for placing merged clones. More concretely, each merged clone is declared as a static method in the class. Besides, target source files are edited to avoid compile error due to improper access modifier of methods and uninitialized local variables. Additionally, the source files are reformatted to avoid inconsistencies of reducible LoC due to coding style. In total, four changes are made in Preprocessing source file changes. We describe each of the changes in Section IV-A:

- generating a new class,
- changing access modifier of methods,
- initializing local variables, and
- reformatting.

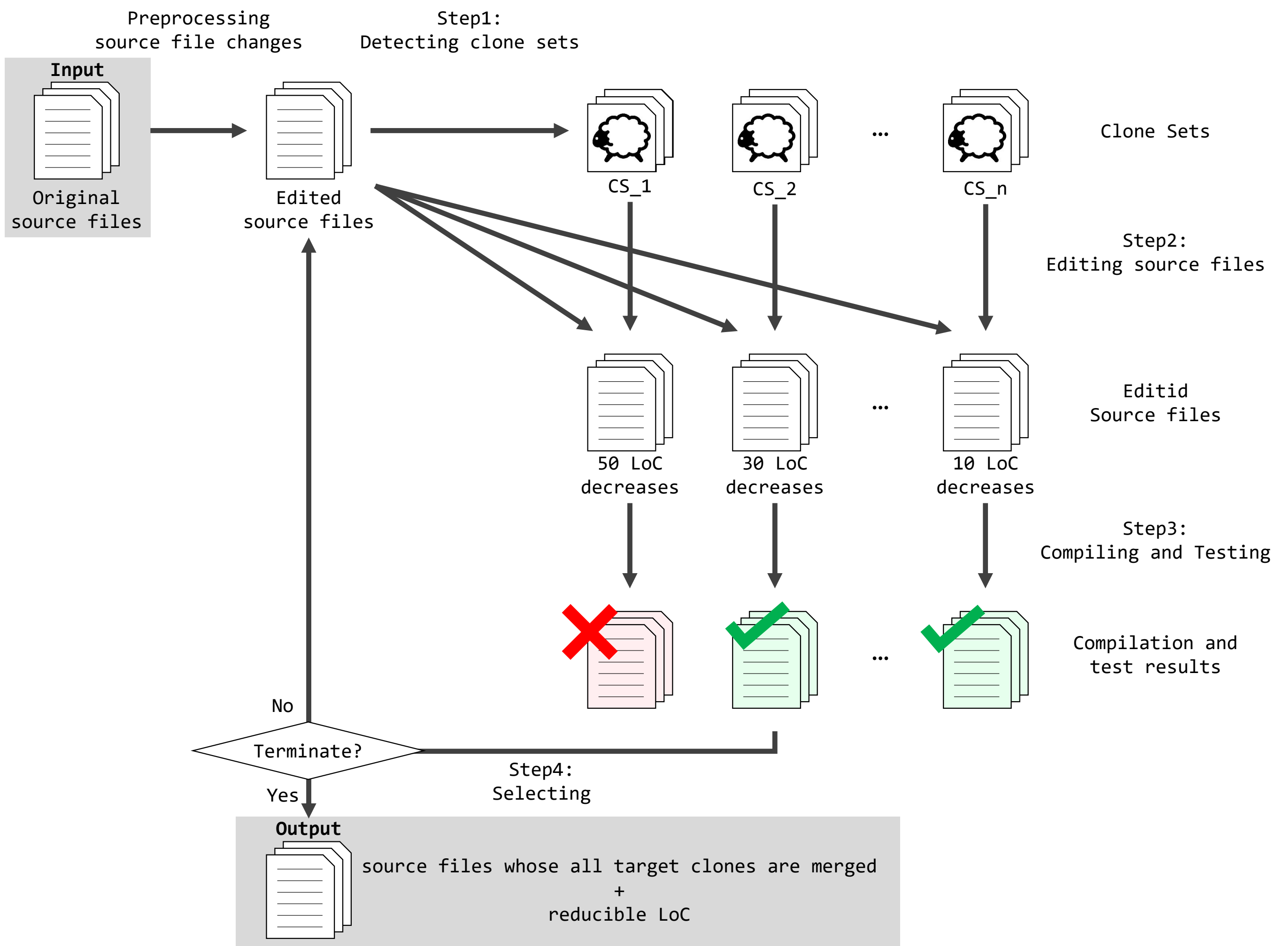


Fig. 3: Overview of proposed technique

B. *Clone Merging*

Clone merging is composed of four steps shown below. All of the steps are performed fully automatically.

Step1: detecting clone sets
Step2: editing source files
Step3: compiling and testing edited source files
Step4: selecting edited source files

In Step1, clone sets in the target source files are detected. In Step2, source files are edited to merge one of the clone sets that were detected in Step1. A method extracted to merge clones is placed in the Java class described in Subsection III-A. In Step3, the source files edited in Step2 are compiled and tested to verify the external behavior. Hereafter, we call edited source files which succeed in compiling and testing and whose LoC gets reduced *selectable source files*. Step2 and Step3 are performed on each of the clone sets detected in Step1. In Step4, the selectable source files whose reduced LoC is the largest is selected. Then, Step1∼Step4 are repeatedly performed on the selected source files. When either of the following termination conditions is satisfied, *clone merging* terminates and outputs the reducible LoC.

- No clone sets are detected.
- No source files edited in Step2 are selectable.

IV. Implementation

We implement our technique as a tool. The tool is written in Java and targets Java source files.

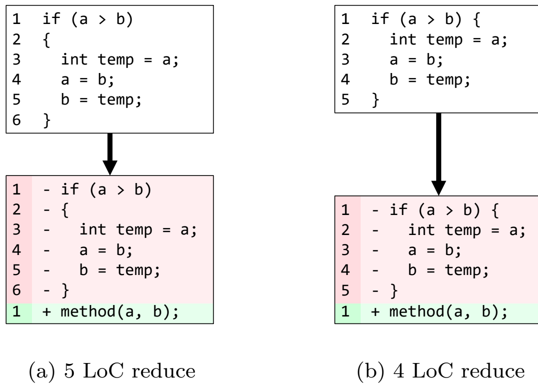
A. *Preprocessing source file changes*

Generating a new Java class

A class in which extracted methods are placed is newly generated. When the extracted method in the new class is called from the original place, it is invoked with its fully qualified name.

Changing access modifier of methods

When a method whose access modifier is not `public` is invoked from another class, a compile error occurs. In manual refactoring, developers can deal with this problem in various ways when refactoring clones. For example, developers can change the access modifier of methods properly, or they can declare the extracted method in the class in which clones were detected. However, in the proposed technique, it is difficult to infer which ways to take because extracting and declaring a method are performed fully automatically. Therefore, access modifiers of all methods are beforehand changed to `public`.



(a) 5 LoC reduce (b) 4 LoC reduce

Fig. 4: Different reducible LoC

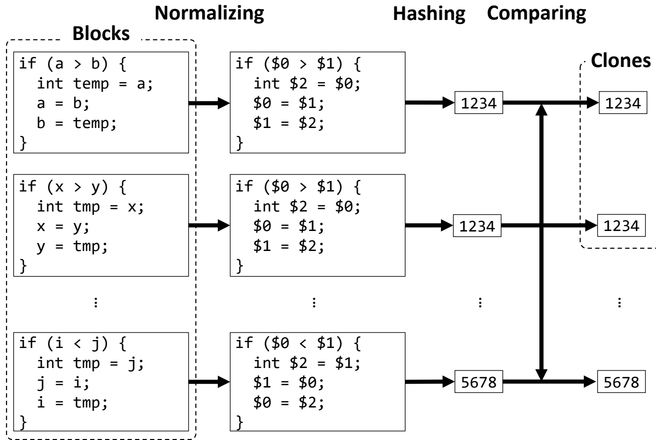


Fig. 5: Detecting clones

Initializing local variables

On the Java language specification, it is prohibited to reference uninitialized local variables. Thus, when an uninitialized local variable is passed to the extracted method as an argument, a compile error occurs. Developers may initialize local variables and pass them to the method when developers merge clones. In this study, all local variables which do not have `final` modifier and are uninitialized at the time of their declaration are initialized. In the case that the local variable is primitive type except for boolean type, it is initialized as "`0`". In the case of boolean type, it is initialized as "`false`". In the case of reference type, it is initialized with "`null`".

Reformatting

In general, coding conventions are specified in each project. For example, some conventions specify using many line breaks and other conventions specify a small number of line breaks. If code fragments of convention violation exist, it is difficult to calculate a reducible LoC properly. Figure 4 shows examples in which different reducible LoC are calculated due to the coding style.

Figure 4a, *Extract Method* refactoring reduces 5 LoC, but Figure 4b, *Extract Method* refactoring reduces 4 LoC. Thus, it is necessary to unify coding style by using formatter. In an ideal world, we should prepare the formatters which reproduce coding conventions of each of the target projects, but of course, it is impossible. Thus, we use the default Eclipse formatter [12].

B. Clone Merging

Step1: Detecting clone sets

Figure 5 shows an example of detecting clones. Clones are detected in block level. We regard the following statements as blocks. Each following statement is defined as a derived class of class `Statement` in Eclipse JDT [13].

- Block
- DoStatement
- EnhancedForStatement
- ForStatement
- IfStatement
- SwitchStatement
- SynchronizedStatement
- TryStatement
- WhileStatement

Block-level clones are more coarse-grained than clones that are detected by token-based techniques. The number of block-level clones is less than the number of token-based clones. Block-level clones have a remarkable feature, which is that they are better candidates of *Extract Method* refactoring because the code fragments are syntactic chunks. The proposed technique uses Eclipse JDT to parse source files and identify blocks. If a block includes a `return` statement, it is not detected as a clone because it is difficult to extract it as a new method [14].

The proposed technique normalizes identified blocks according to the rules shown below because the larger number of clones can be detected by applying the rules.

- Identifiers are normalized as "$" + "number".
- The same identifier is normalized as the same normalized name.
- All literals are normalized as "$".
- Qualified names are normalized as identifiers.
- Class name is not normalized.
- Method name is not normalized.

The same identifier is normalized as the same normalized name to avoid false detection as much as possible. The reason why class and method names are not normalized is to avoid detecting clones whose differences are class or method names because classes and methods cannot be passed to method invocations as arguments in the Java specification.

After the normalization, a hash value is calculated from each block. The proposed method uses SHA256 hashing algorithm [15]. Since SHA256 outputs 256-bit hash value, hash collisions hardly occur.

Finally, the proposed technique compares the hash values of blocks to detect blocks of the same hash values as clones.

Step2: Editing source files

Using *Extract Method* refactoring on one of the clone sets detected in Step1, source files are automatically

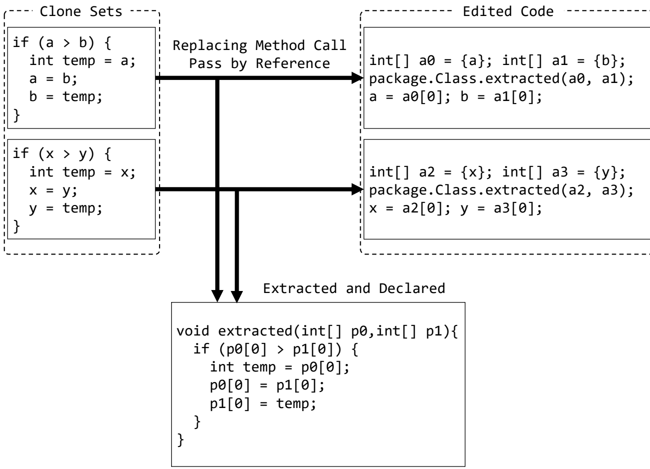


Fig. 6: Editing source files

edited. Figure 6 shows an example of automatically editing source files.

It is possible to reduce LoC because each of the clones is replaced to a method invocation by extracting clones as a method. This extracted method is declared as a `static` method in the class made in *Preprocessing source files changes* (see Section IV-A). The method is invoked with its fully qualified name.

In practice, we need to take care of changes of variables on *Extract Method* refactoring. For example, when only one of the variables is changed in the code fragment, *Extract Method* refactoring can be performed by returning the variable and assigning it in the caller place. However, in the Java language specification, it is impossible to return two or more parameters simultaneously. *Extract Method* refactoring on such clones is not realistic. Unfortunately, examining whether or not each variable is changed in the target code fragment requires deep source code analysis. Thus, In this study, we implemented our tool to pass arguments by reference to the extracted method to keep the external behavior. Our tool uses an array type to pass arguments by reference. Before invoking the extracted method, arrays whose types are the same of variables in the target code fragment are newly defined and initialized with each of the values. These arrays are passed to the method. In the method, the array element at index zero is referenced. After invoking the method, each array element at index zero is assigned back to each variable. These processes are devices of implementation for automated refactoring, not for manual refactoring.

Step3: Compiling and testing edited source files

In Step3, source files edited in Step2 are compiled and tested. At first, compilation is performed. After compilation gets success, test runs. If both compilation and test get success, edited source files are recorded as selectable source files. If either compilation or test fails, edited source files are not recorded.

Step4: Selecting edited source files

After Step2 and Step3 are performed on each of the clone sets detected in Step1, Step4 is performed. In Step4, the selectable source files whose reduced LoC is the largest is selected. Then, Step1∼Step4 are repeatedly performed on the selected source files.

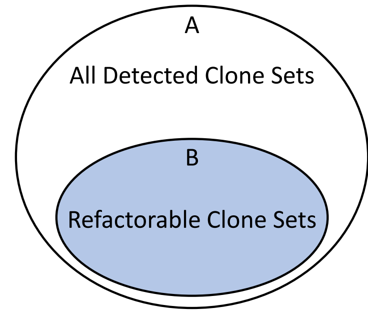


Fig. 7: Target clone sets

V. Experiment

A. Experimental Design

We apply our technique to open source software (in short, OSS). Additionally, we compare our technique to the existing technique [6].

B. Experimental Item1

We investigate how different reducible LoC is obtained in considering refactorability. First, clone sets are detected from the target OSS by using our technique. These clone sets correspond to `A` in Figure 7. Then, the existing technique is applied to them, and a reducible LoC is estimated. Finally, we use our technique to calculate a reducible LoC from target OSS and compare the two techniques.

C. Experimental Item2

We investigate how different reducible LoC is obtained in considering clone overlapping. First, clone sets are detected from the target OSS by using our technique. Second, the source files in the target OSS are edited to merge each clone set, and compilation and test are performed. As a result, we obtain refactorable clone sets. These clone sets correspond to `B` in Figure 7. Then, the existing technique is applied to them, and a reducible LoC is estimated. Finally, we use our technique to calculate a reducible LoC from target OSS and compare the two techniques.

The difference between Item1 and Item2 is the target clone sets, in Item1, the targets are all detected clone sets (`A` in Figure 7) while the targets are only refactorable clone sets (`B` in Figure 7) in Item2.

D. Experimental Targets

We experimented on OSS written in Java. Our targets are composed of `jEdit`, `JFreeChart`, `JRuby`, `Ant`, and `JMeter`, which existing research of clone refactoring [16] targets and `Closure Compiler` (in short, `Compiler`) and `Joda-time`, which Defects4J [17] targets. Defects4J is a collection of reproducible bugs and a supporting infrastructure with the goal of advancing software engineering research. The reason why we add `Compiler` and `Joda-time` to the targets is that we think the projects possess sufficient tests. The proposed technique runs tests to verify the external behavior. Thus, it is to be desired that the target projects possess sufficient tests to verify

the external behavior. The projects targeted in Defects4J possess tests which can detect the bugs, and we think they possess sufficient tests.

Table I shows the name, version, and total LoC of the target projects. We do not target test code and tutorial code of the targets. We measure the total LoC after we use the formatter (see Section IV).

E. Results and Discussion of our technique

Table II shows the results of our technique performing on the target projects. We use a personal workstation in this experiment.

- CPU: 2.40GHz, 12Core
- Memory: 32GB
- OS: Ubuntu18.04

As a result, the refactorable and line-reducible clone sets (Herein, line-reducible clone sets mean that refactoring them can reduce LoC of the target projects) account for 5∼10 percent of the all detected clone sets. The major causes why compilation or test fails are shown below.

- The types of variables between different code fragments in a clone set are different.
- There are clones including references to a Superclass.
- Exceptions thrown in the clone fragment are caught at the outside of the fragment.
- Loop control statements(e.g., `break`) are used in the clone fragment, but loop statements(e.g., `for`) are used in the outside of the fragment.

Some clone sets will be refactorable if we devise to implement our tool. For example, if all code fragments of a clone set are in a class and the code fragments include references to a Superclass, extracting the code fragments as a method in the same class can generate compilable source code.

F. Results and Discussion of Topic1

Table III shows the results of Experimental Item1. The existing technique regards the larger number of clone sets as refactorable than the number of clone sets that the proposed technique was actually able to refactor and estimates more reducible LoC. This is because the existing technique does not consider the refactorability of the clone sets and estimates reducible LoC on non-refactorable clone sets. Thus, if directly the existing technique is applied to clone sets detected by a clone detection tool, it estimates more reducible LoC than actual and the value is improper.

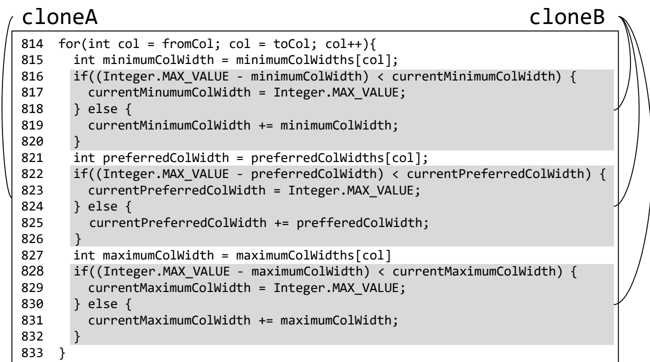


Fig. 8: Real overlapped clones

G. Results and Discussion of Topic2

Table IV shows the results of Experimental Item2. The number of clone sets that the proposed technique merges and the reducible LoC that the technique calculated is equal to or greater than the existing technique. This is because when overlapped clone sets are detected, the existing technique merges the most line-reducible clones and ignores the others while the proposed technique merges the most line-reducible clone sets, detects clone sets again, and merges the other overlapped clone sets if they are line-reducible. Figure 8 shows examples of such clone sets. Both clone A and B are detected in `jEdit`. Clone A includes B. Clone A is 20-LoC, and the clone set consists of 2-clones. Thus, 16-LoC can be reduced when they are merged. On the other hand, B is 5-LoC, and the clone set consists of 6-clones. Thus, 17-LoC can be reduced when they are merged. In this case, the existing technique targets the clone set of clone B and ignores A. However, at first, the proposed technique merges the clone set of clone B.

TABLE I: Target OSS

Name	Version	Total KLoC
jEdit	5.4.0	163
JFreeChart	1.0.19	236
JRuby	1.7.27	334
Ant	1.10.1	231
JMeter	3.2	79
Compiler	20190618	250
Joda-Time	2.10.3	74

TABLE II: Results of our technique

Name	#Detected CS	#Merged CS	Reducible LoC	Execution Time
jEdit	423	57	534	40m
JFreeChart	848	145	1104	2h48m
JRuby	858	236	1281	6h4m
Ant	635	7	327	20m
JMeter	114	5	21	3m
Compiler	286	27	183	3h13m
Joda-Time	89	9	61	7m

TABLE III: Results of Topic1

Name	Existing technique		Proposed technique	
	#Merged CS	Reducible LoC	#Merged CS	Reducible LoC
jEdit	237	2115	57	534
JFreeChart	462	5168	145	1104
JRuby	519	4201	236	1281
Ant	371	3633	7	327
JMeter	49	291	5	21
Compiler	179	1599	27	183
Joda-Time	23	171	9	61

TABLE IV: Results of Topic2

Name	Existing technique		Proposed technique	
	#Merged CS	Reducible LoC	#Merged CS	Reducible LoC
jEdit	55	517	57	535
JFreeChart	142	1098	145	1104
JRuby	231	1270	236	1281
Ant	7	327	7	327
JMeter	5	21	5	21
Compiler	27	183	27	183
Joda-Time	9	61	9	61

Then our technique detects clone sets, and the clone set of cloneA whose code fragments of cloneB is replaced to a method invocation is detected. Thus, our technique can target the clone set of clone A and calculate a reducible LoC by merging both clone sets.

The results of Subsection V-F and Subsection V-G, we can say that our technique can calculate reducible LoC of projects more accurately than the existing technique.

VI. Threats to validity

We experimented on seven OSS. However, if experiments are done on other projects, the results may be different from this study.

In java, method declarations consist of method body and an additional 2-line code. The additional code is composed of a method signature line and a close bracket line. However, other programming languages are not always fitted this rule. Therefore, such languages require different formulas of reducible LoC, and the result may be different from this study.

We implemented the proposed technique as a tool, but some of the implementations are still insufficient. If we improve our implementation, the more correct reducible LoC calculation can be expected.

Existing research proposes not only *Heuristic Method* but also *Complete Method* to estimate reducible LoC. In *Complete Method*, each clone is divided into multiple code fragments, each of which includes no partial overlapping fragment with any other clones. *Complete Method* is designed to aim at merging and removing all clones. However, the possibility of dividing clones is not considered in *Complete Method*. For example, when a code fragment is divided into two code fragments, and each of them is extracted as a method, the latter method cannot reference variables declared in the former method. Existing research explains that *Complete Method* gives an upper bound of the reducible LoC and do not care about the feasibility and usefulness of this merging in practice. Thus, we did not compare our proposed technique to *Complete method*.

VII. Conclusion

In this paper, we proposed a new technique to calculate lines of code that are reduced by removing clones. The proposed technique performs a loop of detecting clones, removing clones, compiling the edited source code, and testing it fully automatically. Thus, the proposed technique calculates reducible lines of code from only clones that actually can be removed while an existing technique estimates it without considering refactorability of clones.

The followings are our future work.

Utilizing other refactoring patterns

At this moment, the proposed technique utilizes *Extract Method* pattern to remove clones. However, there are various ways to remove clones: for example, *Pull Up Method* or *Form Template Method*. Considering other refactoring patterns to remove clones, calculated reducible LoC will get closer to the real value when developers appropriately remove clones.

Avoiding compilation errors

As described in Section V, we encountered many compilation errors in the experiment. However, most of such compilation errors are avoidable if we devise an implementation. If we do so, we will be able to get better reducible LoC.

Acknowledgment

This work was supported by MEXT/JSPS KAKENHI 17H01725.

References

[1] B. P. Lientz and E. B. Swanson, *Software Maintenance Management.* Addison-Wesley Longman Publishing Co., Inc., 1980.

[2] S. G. Eick, T. L. Graves, A. F. Karr, J. S. Marron, and A. Mockus, "Does code decay? Assessing the evidence from change management data," *IEEE Transactions on Software Engineering*, vol. 27, no. 1, pp. 1–12, 2001.

[3] T. M. Pigoski, *Practical Software Maintenance: Best Practices for Managing Your Software Investment*, 1st ed. Wiley Publishing, 1996.

[4] H. M. Sneed, "Planning the reengineering of legacy systems," *IEEE Software*, vol. 12, no. 1, pp. 24–34, Jan 1995.

[5] M. Fowler, *Refactoring: Improving the Design of Existing Code*, ser. Addison-Wesley Signature Series. Pearson Education, 1999.

[6] N. Yoshida, T. Ishizu, B. Edwards, III, and K. Inoue, "How Slim Will My System Be?: Estimating Refactored Code Size by Merging Clones," in *Proceedings of the 26th Conference on Program Comprehension*, 2018, pp. 352–360.

[7] 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*, vol. 28, no. 7, pp. 654–670, 2002.

[8] Y. Higo, S. Matsumoto, S. Kusumoto, T. Fujinami, and T. Hoshino, "Correlation Analysis between Code Clone Metrics and Project Data on the Same Specification Projects," in *Proc. of the 12th International Workshop on Software Clones*, 2018, pp. 37–43.

[9] L. Jiang, G. Misherghi, Z. Su, and S. Glondu, "DECKARD: Scalable and Accurate Tree-Based Detection of Code Clones," in *Proceedings of the 29th International Conference on Software Engineering*, 2007, pp. 96–105.

[10] J. R. Cordy and C. K. Roy, "The NiCad Clone Detector," in *2011 IEEE 19th International Conference on Program Comprehension*, 2011, pp. 219–220.

[11] "CCFinderX," http://www.ccfinder.net/.

[12] "Eclipse," https://www.eclipse.org/.

[13] "Eclipse java development tools," https://www.eclipse.org/jdt/.

[14] R. Komondoor and S. Horwitz, "Effective, automatic procedure extraction," in *11th IEEE International Workshop on Program Comprehension*, 2003, pp. 33–42.

[15] National Institute of Standards and Technology, "Secure Hash Standard," 2015.

[16] N. Tsantalis, D. Mazinanian, and S. Rostami, "Clone Refactoring with Lambda Expressions," in *Proceedings of the 39th International Conference on Software Engineering*, 2017, pp. 60–70.

[17] R. Just, D. Jalali, and M. D. Ernst, "Defects4J: A Database of Existing Faults to Enable Controlled Testing Studies for Java Programs," in *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, 2014, pp. 437–440.