JOURNAL OF SOFTWARE MAINTENANCE AND EVOLUTION: RESEARCH AND PRACTICE
J. Softw. Maint. Evol.: Res. Pract. 2008; **20**:435–461
Published online 3 September 2008 in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/smr.394

Research

A metric-based approach to identifying refactoring opportunities for merging code clones in a Java software system

Yoshiki Higo*, †, Shinji Kusumoto and Katsuro Inoue

Department of Computer Science, Graduate School of Information Science and Technology, Osaka University 1-3, Machikaneyama-cho, Toyonaka, Osaka 560-8531, Japan

SUMMARY

A code clone is a code fragment that has other code fragments identical or similar to it in the source code. The presence of code clones is generally regarded as one factor that makes software maintenance more difficult. For example, if a code fragment with code clones is modified, it is necessary to consider whether each of the other code clones has to be modified as well. Removing code clones is one way of avoiding problems that arise due to the presence of code clones. This makes the source code more maintainable and more comprehensible. This paper proposes a set of metrics that suggest how code clones can be refactored. As well, the tool Aries, which automatically computes these metrics, is presented. The tool gives metrics that are indicators for certain refactoring methods rather than suggesting the refactoring methods themselves. The tool performs only lightweight source code analysis; hence, it can be applied to a large number of code lines. This paper also describes a case study that illustrates how this tool can be used. Based on the results of this case study, it can be concluded that this method can efficiently merge code clones. Copyright © 2008 John Wiley & Sons, Ltd.

Received 31 August 2007; Revised 2 July 2008; Accepted 16 July 2008

KEY WORDS: code clone; software maintenance; refactoring; metrics; object-oriented

*Correspondence to: Yoshiki Higo, Department of Computer Science, Graduate School of Information Science and Technology, Osaka University 1-3, Machikaneyama-cho, Toyonaka, Osaka 560-8531, Japan.
†E-mail: higo@ist.osaka-u.ac.jp

Contract/grant sponsor: Stage Project, the Development of Next Generation IT Infrastructure
Contract/grant sponsor: Japan Society for the Promotion of Science, Grant-in-Aid for Scientific Research (A); contract/grant number: 17200001
Contract/grant sponsor: Japan Society for the Promotion of Science, Grant-in-Aid for Young Scientists (Start-up); contract/grant number: 19800022

1. INTRODUCTION

As the size and complexity of software systems increase, maintenance tasks become more difficult and burdensome. Maintenance of software systems is defined as the *modification of a software product after delivery to correct faults*, *to improve performance or other attributes*, *or to adapt the products to a modified environment* [1]. Arthur stated that only one-fourth to one-third of all life-cycle costs are attributed to software development and Yip and Lam reported that 67% of the life-cycle costs are expended in the operation-maintenance phase of the life cycle [2,3].

It is generally thought that the presence of code clones is one factor that makes software maintenance more difficult. A **code clone** is a code fragment that has identical or similar source code fragments elsewhere in the code. If a code fragment with code clones is modified, it is necessary to consider whether each of the code clones should also be modified. To automatically detect code clones, several methods have been proposed and software tools have been implemented [4–8].

Refactoring has recently received a lot of attention. **Refactoring** is a set of operations to improve the maintainability, understandability, or other attributes of a software system without changing its external behavior. A code clone is one of the typical *Bad Smells* in the refactoring process [9]. Code clone detection can be perceived as the identification of code fragments to be refactored. From a practical standpoint, it is difficult to identify the code clones that are to be merged. Some code clones are simply unmergeable, or, if they are merged, they make the source code less understandable. Usually, large-scale software systems have complicated intertwining logics that make it difficult to identify the code clones that can be merged and how best to merge them.

In this paper, a new method for merging code clones is proposed. The method consists of two phases. The first phase is the quick detection of *refactoring-oriented code clones* from the source code. The second phase is the measurement of metrics indicating the manner in which *refactoring-oriented code clones* should be merged. The Aries software tool implements this method. Using Aries in the refactoring process, maintainers of the software system can readily know which and how code clones can be merged. This paper also presents a case study conducted using Aries. Based on the results of this case study, it was concluded that Aries performs the process successfully.

2. CODE CLONE

No consistent or precise definition of code clones is currently available. Code clones are often operationally defined by individual clone detection methods. The established detection techniques can be categorized as follows:

Line-based technique: Each line of the source code is compared with other lines. If consecutive lines of code are identical to other lines of code, they are detected as code clones.

Token-based technique: After the source code is divided into tokens, identical token sequences that are longer than a certain length are detected as code clones.

Abstract Syntax Tree (*AST*)*-based technique*: After building an AST from the source code, subtrees having the same structure are detected as code clones.

DOI: 10.1002/smr

Program Dependency Graph (*PDG*)*-based technique*: After building a PDG as a result of the semantic analysis of the source code, isomorphic subgraphs are detected as code clones.

Metric-based technique: After measuring several metrics from a certain unit, for example the function, method, or class, of the software system, units having identical or similar metrics values are detected as code clones.

Rysselberghe and Demeyer discussed the features of the three code clone detection techniques, namely line-based, token-based, and metric-based, from a refactoring perspective [10].

Evaluation of the metric-based technique: The *metric-based code clones* are well-suited to be refactored because they can be the target of refactoring operations in their entirety. However, they include many false positives. The smaller the functions, the more they tend to be detected as *metric-based code clones* because their metrics values are exactly or approximately the same, despite the fact that they cannot be refactored. Another problem is that if a part of a function is identical or similar to another part of a different function, they are not detected, since code clone detection is performed at the function level.

Evaluation of the line-based technique: The *line-based code clones* can be refactored with little effort. However, this method cannot detect code clones that are coded using different coding styles. Another, more important issue is that this method cannot detect code clones with different identifiers such as variable names, function names, or type names.

Evaluation of the token-based technique: The *token-based techniques* is a good detection method because it can detect code clones that have different formats, such as different indention, white space, and tab, and different identifiers. However, some of the *token-based code clones* are not suited for refactoring due to the following reasons:

- The token-based technique identifies duplicated token sequences as code clones. Hence, *token-based code clones* do not necessarily correspond to raw statements or expressions in the source code.
- There are sometimes semantic differences between *token-based code clones* belonging to the same clone set because the token-based technique replaces identifiers with special tokens before detection.

It is assumed that the detection unit of the metric-based technique and the identifier replacement of the token-based technique are suitable for refactoring despite having the potential for the existence of semantic differences. Thus, the proposal extracts the structural units of the programming language, for example, the method or loop, from identifier-replaced code clones. The targets of refactoring in the proposal are the extracted parts.

In addition, the AST-based technique is a good detection method from a refactoring perspective because it identifies cohesive parts of the program source code as code clones, such as whole methods or consecutive statements in the same scope. In many cases, it is probably easier to refactor *AST-based code clones* than *token-based code clones* or *line-based code clones*. However, comparing subtrees is expensive, so that it is difficult to simply apply the AST-based technique to large-scale software systems. Usually, the bigger the software scale, the more they need to be refactored since large-scale software includes many complicated logics that are intertwined with one another. However, it is difficult to manually detect the code to be refactored from the large number of lines of code. Scalability is an important property of the tools that detects the code to be refactored.

DOI: 10.1002/smr

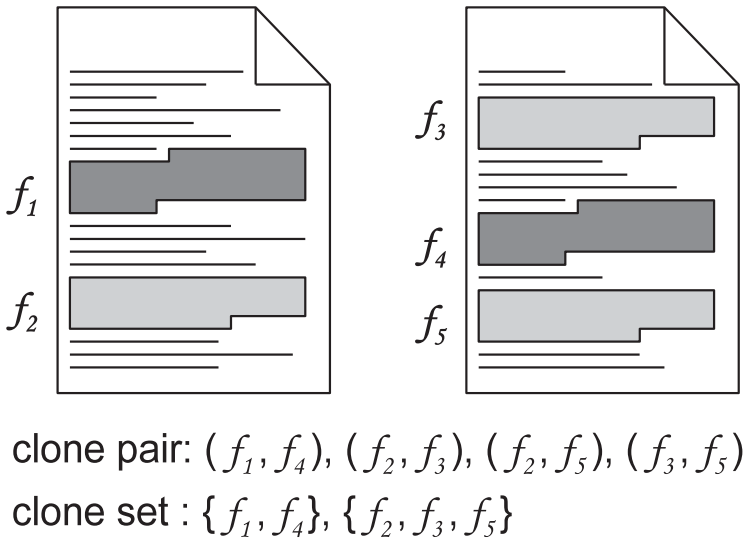

Figure 1. Clone Pair and Clone Set.

Burd *et al.* reported that CloneDR, which is an implementation of the AST-based technique, has good code clone detection precision[‡], but it could only detect a small part of the code clone set detected using other detection techniques. Moreover, CloneDR has high scalability despite using the AST-based technique. Generally, the AST-based technique requires $O(n^2)$ complexity (n is the number of subtrees) for the subtree comparisons. However, CloneDR puts subtrees into small groups by hashing them in the first traversal phase. The quadratic comparison is performed on each pair of subtrees in the same group. This approach can reduce the number of comparisons drastically, so that CloneDR can detect code clones efficiently from middle-scale or large-scale software systems. The PDG-based technique is the most expensive technique available, so much so that it is difficult to apply the technique to middle-scale or large-scale software systems.

2.1. Clone Pair and Clone Set

This section explains two terms related to the code clones. The first term is **Clone Pair**, which is a pair of identical or similar code fragments. The other is **Clone Set**, which is a set of code fragments that are identical or similar to each other.

Figure 1 illustrates an example. In this figure, there are five code fragments containing code clones. Code fragment f_1 is similar to code fragment f_4, and code fragments f_2, f_3, and f_5 are similar to each other. In this example, there are four clone pairs, (f_1, f_4), (f_2, f_3), (f_2, f_5), (f_3, f_5), and two clone sets, $\{f_1, f_4\}$, $\{f_2, f_3, f_5\}$.

[‡]However, it should be noted that Burd *et al.* did not evaluate code clone detection techniques from a refactoring perspective.

DOI: 10.1002/smr

3. METHOD FOR MERGING CODE CLONES

The method consists of three phases; **Detection Phase**, **Extraction Phase**, and **Measurement Phase**. Sections 3.1–3.3 explain each phase. After that, Section 3.4 describes two refactoring examples.

3.1. Detection phase

In this phase, code clones are detected using an existing detection technique. Based on the above discussion, it was decided to use either the token-based technique or the AST-based technique as the detection technique. The reason for the recommendation is that these techniques can detect code clones having different identifiers. As Balint *et al.* showed, after copying and pasting a code fragment, small modifications are often introduced to the copied code fragment. Thus, it is expected that such techniques can detect identifier-replaced code fragments. In the proposal, code clones detected using an existing detection technique are called **general code clones**.

3.2. Extraction phase

In this phase, the following granularity units are extracted from the *general code clones*:

Declaration: class, interface,

Function: method, constructor, static initializer,

Block: do, for, if, switch, synchronized, try, while.

The extracted units are called **refactoring-oriented code clones**. *Refactoring-oriented code clones* are more suitable for refactoring than *general code clones*. Figure 2 illustrates an example. In this figure, there are two code fragments *A* and *B* from a program. The code fragments with hatching parts are *general code clones*, which were detected using the token-based technique. In code fragment *A*, operations on *class name* are performed, whereas in code fragment *B*, operations

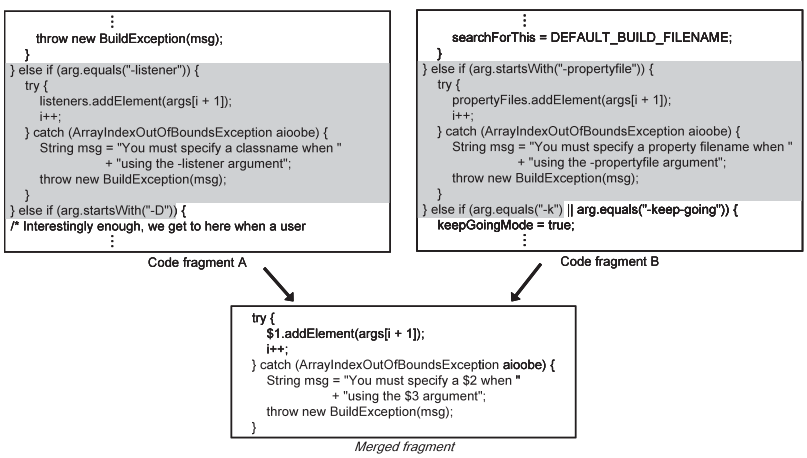

Figure 2. Example of merging two code fragments.

DOI: 10.1002/smr

on *property file name* are performed. The try-catch blocks in *A* and *B* have a common logic that handles a java.util.Vector data structure. There are, however, a few statements before and after the try-catch blocks that are not necessarily related to the try-catch blocks from a semantic standpoint. The presence of such semantically unrelated statements often obstructs refactoring. In other words, extracting only the try-catch blocks as code clones is more preferable to extracting else–if blocks from a refactoring viewpoint.

If the AST-based technique or the metric-based technique is used in the detection phase, *general code clones* themselves are any of the above granularity units, that is, *general code clones* are the same as the *refactoring-oriented code clones*. Therefore, the extraction phase can be omitted.

3.3. Measurement phase

After extracting the *refactoring-oriented code clones*, some metrics are measured for them. The metrics represent whether or not each of the code clones can be easily merged and how to merge them. Users determine, using the metrics, whether or not each of the *refactoring-oriented code clones* should be merged.

Sections 3.3.1 and 3.3.2 describe the proposed metrics for merging code clones. In addition, Section 3.4 shows two examples of merging code clones using these metrics.

3.3.1. Positional relationship in the class hierarchy

On source code written in Java language, the strategy for merging code clones depends on their positional relationship in the class hierarchy. Figure 3 illustrates some examples.

Case 1: If code clones are in a single class, they can be extracted as a new method in the same class (see Figure 3(a)).

Case 2: If code clones are in two or more classes derived from a single class, they can be pulled up to the common base class (see Figure 3(b)).

Case 3: If code clones are scattered in different classes, a new class is required to merge them (see Figure 3(c)). In Figure 3(c), new class *TelephoneNumber* was created, and the duplicated function was delegated to the new class. Besides, a new class that will be a common parent of duplicated classes can always be created unless the classes are already inherited from another class.

To automatically indicate an appropriate way to merge code clones, the positional relationship of the code clones belonging to clone set S is given using the quantitative metric, the Dispersion in the Class Hierarchy (**DCH(S)**).

Here, it is assumed that clone set S_1 includes the code fragments $f_1, f_2, \ldots, f_n$. A class including code fragment f_i is represented as c_i.

Case A: If classes $c_1, c_2, \ldots, c_n$ have one or more common base classes, c_p is defined as a class situated at the lowest position in the class hierarchy among the common base classes. $DCH(S_1)$ is the maximum number of hops from c_i $(1 \leq i \leq n)$ to c_p in the class hierarchy.

Case B: If the classes have no common base class, $DCH(S_1)$ is set to ∞.

The formula for the metric $DCH(S_1)$ can be represented as follows:

$$DCH(S_1) = \begin{cases} max\{D(c_1,\ c_p),\ D(c_2,\ c_p), \ldots, D(c_n,\ c_p)\} & \text{(Case A)} \\ \infty & \text{(Case B)} \end{cases} \quad (1)$$

DOI: 10.1002/smr

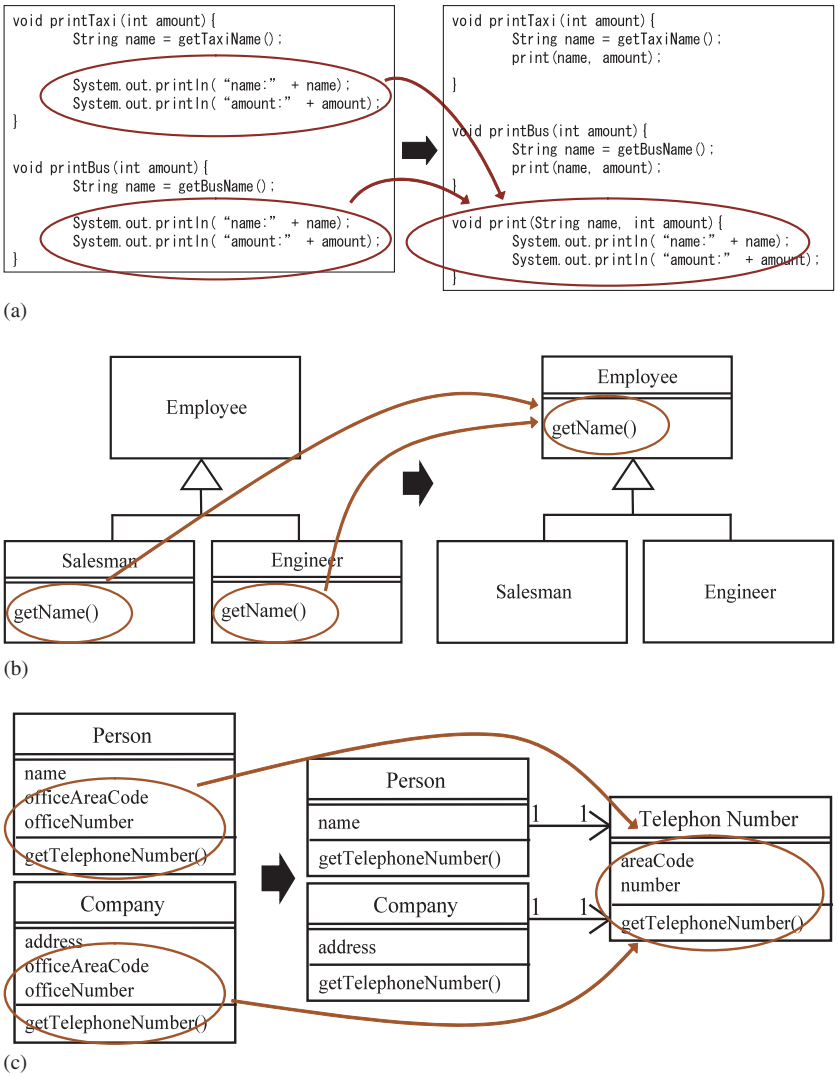

Figure 3. Examples for merging code clones: (a) Example of CASE1; (b) Example of CASE2; and (c) Example of CASE3.

where $D(c_i, c_p)$ is the number of hops from c_i $(1 \leq i \leq n)$ to c_p in the class hierarchy. If $c_i = c_p$, $D(c_i, c_p)$ is set to 0.

$DCH(S_1)$ becomes large as the degree of the dispersion of S_1 becomes extended. If all code fragments of S_1 are in a single class, $DCH(S_1)$ is set to 0. If all code fragments of S_1 are in a class and its direct child classes, $DCH(S_1)$ is set to 1. Additionally, this metric is measured for only the

DOI: 10.1002/smr

class hierarchy of the target source code because it is unrealistic for it to pull up application code to libraries such as JDK.

3.3.2. Coupling between a code clone and it surrounding code

As described in Section 3.3.1, the basic strategy for merging code clones is migration of duplicated code to another place. To migrate implemented code, it is desirable that the duplicated code has low coupling with its surrounding code.

Assume that the *Extract Method*[§] is to be performed. To apply this method, the smaller the number of externally defined variables that are used (referenced and assigned) in the code fragment, the easier it is to migrate the code fragment to another place. If externally defined variables are used in the target code fragment, it is necessary to provide the variables as parameters to the extracted method. In order to automatically measure the ease of code migration, the degree of coupling is represented as 2 quantitative metrics, the number of referenced variables (**NRV(S)**), and the number of assigned variables (**NAV(S)**).

Here, it is assumed that clone set S_2 includes code fragments $f_1, f_2, \ldots, f_m$. Code fragment f_i references s_i number of variables defined externally, and it assignes to t_i number of variables defined externally. The formula of metrics $NRV(S_2)$ and $NAV(S_2)$ can be represented as follows:

$$NRV(S_2) = \frac{1}{m}\sum_{i=1}^{m} s_i, \quad NAV(S_2) = \frac{1}{m}\sum_{i=1}^{m} t_i \tag{2}$$

Intuitively, $NRV(S_2)$ represents the average of the externally defined variables referenced in the code fragments belonging to clone set S_2. In the same way, $NAV(S_2)$ represents the average of externally defined variables assigned in the same code fragments.

If *refactoring-oriented code clones* described in Section 3.1 are code clones that are structurally identical and either completely identical or have only parameterized differences, s_i and t_i $(1 \leq i \leq n)$ are always identical. In this case, these metrics can be presented as follows[¶]:

$$NRV(S_2) = s_1 = s_2 = \cdots = s_m, \quad NAV(S_2) = t_1 = t_2 = \cdots = t_m \tag{3}$$

If the *refactoring-oriented code clones* include some gaps, which are added or deleted statements, s_i/t_i can be different from s_j/t_j $(1 \leq i, j \leq n, i \neq j)$. In this case, the definitions of formula (2) for computing $NRV(S_2)$ and $NAV(S_2)$ must be used.

3.4. Examples of merging code clones

Metric $DCH(S)$ represents wherein the class hierarchy the common code can be factored out, and metrics $NRV(S)$ and $NAV(S)$ represent the couplings between code clones and their surrounding code. It is assumed that these metrics can represent the possibility of some refactoring patterns in

[§] Originally, the *Extract Method* is applied to a too long method or a part of a complicated function in order to improve the readability, understandability, and maintainability of a program [9]. It also can be applied to code clones to merge them.

[¶] In the implementation of the proposal, a code clone detection tool, CCFinder [6] is used. CCFinder detects only the structurally identical code clones, so that $NRV(S)$ and $NAV(S)$ are calculated using these formulas.

DOI: 10.1002/smr

which code clones are merged in another place of the class hierarchy. For example, the following refactoring patterns in Fowler's book [9] can be helped using these metrics:

Extract Class/*SuperClass*: If a class level duplication is found, the *Extract Class* or *Extract SuperClass* patterns may be performed. If duplicated classes extend different base classes, the duplication can be removed by *Extract Class*. If not, *Extract SuperClass* should be the better choice for removal. $DCH(S)$ is an indicator of which of the two patterns should be performed.

Extract Method: If some of the parts of the methods in a single class are duplicated, the *Extract Method* is a simple and practical solution for removal. $DCH(S)$ is an indicator for identifying duplication in a single class, and $NRV(S)$ and $NAV(S)$ are indicators for identifying duplication that can easily be removed. Section 3.4.1 introduces a typical set of conditions of the metrics for applying the *Extract Method*.

Pull Up Method: If there are duplicated methods in different classes that extend a common base class, the *Pull Up Method* is a good method for duplication removal. It is easy to identify such method level duplication with $DCH(S)$. If the value is 1 or more (not ∞), the duplication has a common base class.

Form Template Method: If the method level duplication is found in different classes that extend a common base class, the *Pull Up Method* pattern should be the first candidate for removal. However, if duplicated methods include different logics, it is difficult to apply the *Pull Up Method*. In such cases, the *Form Template Method* may be applicable.

Figure 4 illustrates an example of the *Form Template Method*. Before refactoring, there are duplicated methods in classes ResidentialSite and LifelineSite. The duplicated methods have different logics in computing the tax, so that it is difficult to apply the *Pull Up Method* to them. In such a case, the *Form Template Method* instead of the *Pull Up Method* is applied to the duplication. First, the different logic parts are extracted as new methods. Then, an abstract method is defined for the new methods in the common base class. Finally, the duplicated methods are pulled up to the common base class.

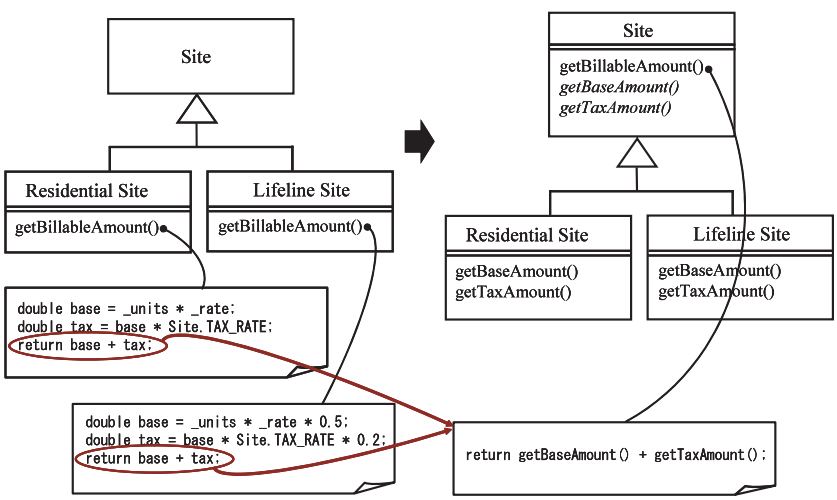

Figure 4. Example of form template method.

DOI: 10.1002/smr

Note that the proposed metrics simply identify 'duplicated methods in different classes that extend a common base class'. Users have to determine whether the *Pull Up Method* or the *Form Template Method* is a better choice for removal.

Move Method: If there are duplicated methods in different classes that have no common base class, it is difficult to apply refactorings using class hierarchy to them. In such cases, the *Move Method* is a good solution for removal. $DCH(S)$ is an indicator to identify such method duplication because value ∞ means that the duplication has no common base class.

Parameterize Method: If some methods in a single class are duplicated, the duplication may be removed by the *Parameterized Method*. This type of duplication can be identified with $DCH(S)$, since a value of 0 implies that all the duplication is in a single class.

Pull Up Constructor: This pattern is very similar to the *Pull Up Method* pattern. The only difference is that the refactoring target is not a method but a constructor.

The remainder of this subsection presents two sets of conditions of the granularity units and the corresponding metrics. The first set is for the *Extract Method* and the second one is for the *Pull Up Method*. Note that the sets described here are just examples and other conditions can be reasonable for using the *Pull Up Method* and the *Extract Method*.

3.4.1. Example 1: Extract Method

If a user wants to merge code clones using the *Extract Method*, then a typical set of conditions could be as follows:

EC1 (*Extract Method Condition 1*): The target granularities are the statement units;

EC2 (*Extract Method Condition 2*): $DCH(S)$ is 0;

EC3 (*Extract Method Condition 3*): $NAV(S)$ is 1 or less.

Since the *Extract Method* is directed to a part of a method, (EC1) is considered. If all code fragments are in a single class, it is easy to merge them, so that (EC2) is considered. The reason for (EC3) is that, if some values are assigned to externally defined variables in the code fragment, it is necessary to add parameters and a return-statement to the new method. It is necessary to contrive a new data class if two or more externally defined variables are assigned values.

The primary operations for the *Extract Method* are as follows:

EXTRACT: A set of operations for simply extracting a code fragment as a new method: for example, (1) cut the target code fragment, (2) paste the code fragment outside the method, and (3) add a simple signature (a method name with no parameter) to the code fragment.

PARAMETER: A set of operations for removing the direct access to the externally defined variables and instead passing them as input parameters: for example, (1) add parameters to the signature, and (2) replace the references to the externally defined variables with references to the parameters.

Table I. Classifying code clones satisfying (EC1) to (EC3).

	EXTRACT	PARAMETER	RETURN	OTHER
EG1	Required	—	—	—
EG2	Required	Required	—	—
EG3	Required	—	Required	—
EG4	Required	Required	Required	—
EG5	Required	Do no care	Do no care	Required

DOI: 10.1002/smr

RETURN: A set of operations for adding a return-statement to the extracted method to reflect the result of the assignment in it to the caller place: for example, (1) define a new local-variable, (2) replace the assignment to the externally defined variable with an assignment to the local-variable, and (3) add a return-statement for passing the value of the local-variable.

OTHER: Operations other than the above ones for extracting code fragments as methods. It is assumed that all code clones satisfying (EC1) to (EC1) can be removed with the above three kinds of operations: EXTRACT, PARAMETER, and RETURN. Some code clones may require other efforts to remove them, or it may be impossible to remove. As a matter of convenience, such code clones, by definition, require the OTHER operations.

EXTRACT operations are required by any of the code clones in performing the *Extract Method*, whereas needing the PARAMETER, RETURN, and OTHER operations depends on the internal logics of the code clones. If a clone set satisfies all the conditions (EC1) to (EC3), it is categorized into one of the following groups; (EG1) to (EG5). Table I represents the relationships among the five groups and their required operations.

EG1 (*Extract Method Group 1*): Clone sets that can be merged by just extracting the code fragments as a new method in the same class, that is, the code fragments use no externally defined variables. This group requires only the EXTRACT operations.

EG2 (*Extract Method Group 2*): Clone sets that can be merged by extracting the code fragments as a new method by adding parameters for the externally defined variables, that is, the code fragments reference one or more externally defined variables. This group requires the EXTRACT and PARAMETER operations.

EG3 (*Extract Method Group 3*): Clone sets that can be merged by extracting the code fragments as a new method by adding a return-statement, that is, the code fragments assign to an externally defined variable. This group requires the EXTRACT and RETURN operations.

EG4 (*Extract Method Group 4*): Clone sets that can be merged by extracting the code fragments as a new method by adding parameters and a return-statement, that is, the code fragments reference externally defined variables and assign values to one of them. This group requires the EXTRACT, PARAMETER, and RETURN operations.

EG5 (*Extract Method Group 5*): Clone sets that could potentially be merged, but require too much effort. This group, by definition, requires the OTHER operations. For this categorization, it is irrelevant whether the clone sets in (EG5) require the PARAMETER and RETURN operations or not. Thus, the corresponding cells in Table I are 'do no care'.

3.4.2. Example 2: Pull Up Method

If a user wants to merge code clones by performing the *Pull Up Method*, the following conditions are reasonable:

PC1 (*Pull Up Method Condition 1*): The target granularity is the method;

PC2 (*Pull Up Method Condition 2*): The value of $DCH(S)$ is 1 or more (not ∞);

PC3 (*Pull Up Method Condition 3*): The value of $NAV(S)$ is 0.

Usually, the *Pull Up Method* is performed on existing methods, which is the reason for (PC1). (PC2) requires all classes including code clones (duplicated method) to extend a common base class. (PC3) prevents replacing more than one external reference with local-variable assignments. A single method cannot return two or more different values as return-statements.

DOI: 10.1002/smr

The primary operations for the *Pull Up Method* are as follows:

MOVE: A set of operations for simply moving a code fragment to another place, for example, (1) cut the target method from the original place and (2) paste it in the common base class.

PARAMETER: A set of operations for removing direct accesses to variables that cannot be used in the common base class and, instead, pass them as input parameters, for example, (1) add parameters to the signature and (2) replace the references to the unavailable variables with references to the parameters.

OTHER: Operations other than the above ones.

As was seen in the case of the *Extract Method*, not all code clones satisfying (PC1) to (PC3) can be removed using the above two operations, EXTRACT and PARAMETER. Some code clones may require more effort to remove them, or it may be impossible to remove them. As a matter of convenience, such code clones are defined to require the OTHER operations.

MOVE operations are required by any of the code clones in performing the *Pull Up Method*, whereas the requirements of the PARAMETER and OTHER operations depend on the internal logic of the code clones. If the condition of PC3 is '*NAV*(*S*) *is 1 or less*', one more primary set of operations, RETURN, should be added.

RETURN: A set of operations for adding a return-statement to the moved method to reflect the result of the assignment in it to the caller place.

When *NAV*(*S*) is 1, there is an assignment to an externally defined variable. Such assignments have to be changed to local-variable assignments, and a return-statement has to be added for reflecting the assignment result to the caller place.

MOVE operations are required by any of the code clones in performing the *Pull Up Method*, whereas the requirements of PARAMETER and OTHER operations depend on the internal logic of the code clones. If a clone set satisfies all the conditions (PC1) to (PC3), it is categorized into one of the following groups, (PG1) to (PG3). Table II shows the relationships between classified groups and their required operations.

PG1 (*Pull Up Method Group 1*): Clone sets that can be merged by just moving the code fragments to the common base class, that is, the code fragments utilize no externally defined variables. This group requires only MOVE operations.

PG2 (*Pull Up Method Group 2*): Clone sets that can be merged by moving the code fragments to the common base class by adding parameters for referencing externally defined variables, that is, the code fragments reference one or more externally defined variables. This group requires the MOVE and PARAMETER operations.

We can choose either to delete existing methods including the code clones or change them to call using the new method from the inside. If the existing methods are deleted, it is necessary to change all of the caller places because the signature is changed.

PG3 (*Pull Up Method Group 3*): Clone sets that require ingenuity in merging them, that is, this group requires OTHER operations. As for the *Extract Method*, it is irrelevant whether clone sets

Table II. Classifying code clones satisfying (PC1) to (PC3).

	MOVE	PARAMETER	OTHER
PG1	Required	—	—
PG2	Required	Required	—
PG3	Required	Do no care	Required

DOI: 10.1002/smr

in (PG3) require the PARAMETER operations or not, so that the corresponding cell of Table II is 'do no care'.

4. ARIES: AN IMPLEMENTATION OF THE PROPOSAL

This section describes **Aries**, which is an implementation of the proposal. Aries assists users in merging code clones. However, the assistance is only a part of refactoring process. Mens and Tourwe [11], which is a good survey of refactoring technologies, state that refactorings are performed in the following steps:

Step 1: *Identify where the software should be refactored*,

Step 2: *Determine which refactoring should be applied to the identified places*,

Step 3: *Guarantee that the applied refactoring preserves the behavior*,

Step 4: *Apply the refactoring*,

Step 5: *Assess the effect of the refactoring on the quality characteristics of the software*, *such as complexity*, *understandability*, *or maintainability*, and the process, such as productivity, cost,

Step 6: *Maintain consistency between the refactored program code and other software artifacts*, *such as documentation*, *design documents*, *requirements specifications*, *and tests*.

Aries assists in Steps 1 and 2 of the above operating procedure by first determining which refactorings could be performed and suggesting, based on the conditions determined above, which type of refactoring to perform. The user must choose whether or not the refactoring actually occurs.

Aries consists of two components, an analysis component and a graphical user interface (GUI) component, which are shown in Figure 5. Sections 4.1 and 4.2 describe the two components of Aries. After that, Section 4.3 shows the process of merging code clones with Aries.

4.1. Analysis component of aries

The analysis component takes two inputs: source files and minimum token length. Source files are the target of refactoring and the minimum token length is a threshold of code clone detection. *Refactoring-oriented code clones* that are longer than the threshold are output to the code clone data file. The processing from input to output is completely automated.

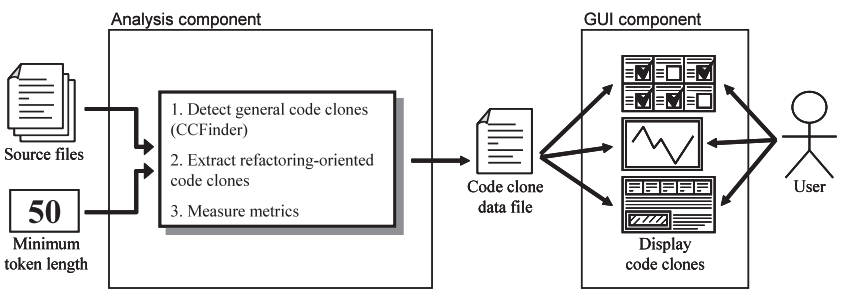

Figure 5. System overview.

DOI: 10.1002/smr

The analysis component utilizes an existing code clone detection tool, CCFinder. The reasons for using CCFinder are:

- CCFinder has high scalability. It can finish code clone detection within an hour even if the source code has millions of lines of code.
- CCFinder can handle Java language, as well as other popular programming languages.
- Many researchers and practitioners favor CCFinder, which suggests that this program is easy to use and has high reliability.

After obtaining the *general code clones* using CCFinder, the analysis component parses the source code and builds the ASTs. Th e granularity units (described in Section 3.2) located in the *general code clones* are identified by comparing the locations of granularity units with the *general code clones*. Identified granularity units are the *refactoring-oriented code clones*, and the metrics $DCH(S)$, $NAV(S)$, and $NRV(S)$ are measured using information from the ASTs. After measuring the metrics from all the *refactoring-oriented code clones*, the information is output to the code clone data file in XML format.

In this implementation, the target source files are parsed twice. The first time is by using the CCFinder, and the second time is for building the ASTs. This redundancy can be removed by implementing a code clone detection tool that detects the *refactoring-oriented code clones* with the measurement of the metrics $DCH(S)$, $NAV(S)$, and $NRV(S)$. However, it is difficult and costly to develop a practical detection tool like CCFinder or CloneDR, which is an implementation of the AST-based technique. The current implementation can analyze middle-scale or large-scale software systems quickly despite the fact that it parses twice. This implies that the implementation is appropriate for practical use.

4.2. GUI component of aries

The GUI component loads a code clone data file and visually displays the code clones stored in it. The GUI component supports interactive investigation of code clones for refactoring. Figure 6 shows snapshots of the GUI component. The Main Window (Figure 6(a)) is used to select the code clones as the refactoring targets. The following describes the details of the Main Window:

Metric Graph: The Metric Graph visually represents the features of code clones using the six metrics. Three of the metrics, which are $DCH(S)$, $NRV(S)$, and $NAV(S)$, are proposed in Section 3.3. The other three metrics, which are $LEN(S)$, $POP(S)$, and $DFL(S)$, were previously proposed in [12].

The Metric Graph allows users to filter out code clones that they are not interested in. Figure 7 illustrates how users filter out code clones using the Metric Graph. In the Metric Graph, each metric has a parallel coordinate axis. Users can specify the upper and the lower limits of each metric. The hatched region is the range bounded by the upper and the lower limits of the metrics. A polygonal line is drawn for each clone set. In this figure, two lines for clone sets S_1 and S_2 are drawn. In the left graph (Figure 7(a)), all the metric values of both the clone sets are in the hatched region, which implies that neither of them is filtered out. In the right graph (Figure 7(b)), $DCH(S_2)$ is larger than the upper limit of DCH; hence, S_2 is filtered out. The Metric Graph enables users to filter out clone sets based on their metric values.

NRV/NAV Selector: On the NRV/NAV Selector, users determine which types of variables are counted for the metrics $NRV(S)$ and $NAV(S)$. In the current implementation, variable types can be

DOI: 10.1002/smr

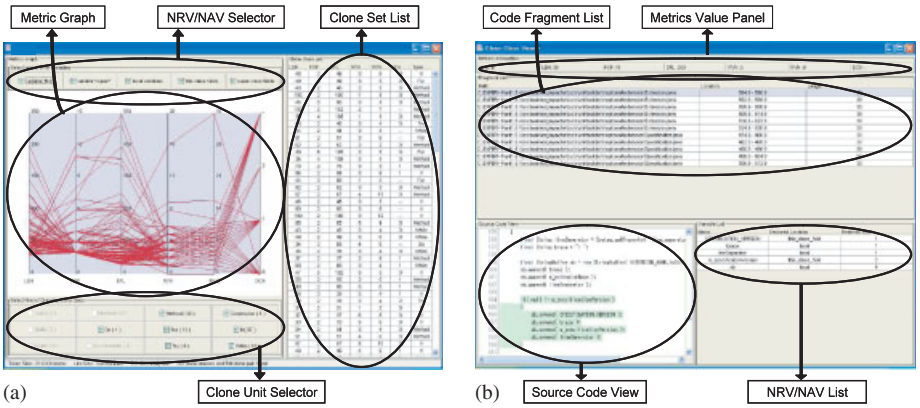

Figure 6. Snapshots of Aries: (a) Main Window and (b) Clone Set Viewer.

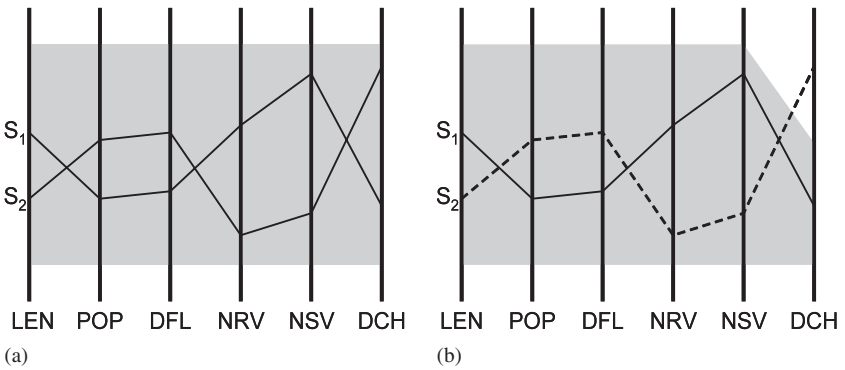

Figure 7. Metric Graph model: (a) before selection and (b) after selection.

selected from:

- field-member-of-its-class;
- field-member-of-parent-class;
- field-member-of-interface;
- local-variable;
- variable ‘this’;
- variable ‘super’.

For example, if users are going to perform the *Extract Method* within a single class, it is not necessary to count all types except local-variable because they can be accessed anywhere in the same class. On the other hand, if users are going to perform the *Pull Up Method*, other types should be counted because the refactoring pattern involves code migration across classes.

DOI: 10.1002/smr

Clone Unit Selector: On the Clone Unit Selector, users determine the kinds of granularity units that are to be displayed in the Metric Graph. As described in Section 4.1, there are 12 kinds of granularity units in Clone Unit Selector.

For example, if users are going to perform the *Pull Up Method*, they should select only the method-unit, because this pattern is a set of operations on the existing methods.

Clone Set List: The Clone Set List displays all clone sets that are not filtered out by the Metric Graph. In other words, code clones satisfying all of the conditions on granularity units and metrics are listed in the Clone Set List. The list has a sorting function that allows clone sets to be sorted in the ascending or descending order of any of the metrics.

Figure 6(b) is a snapshot of the Clone Set Viewer that is launched by double-clicking a clone set on the Clone Set List. It shows more detailed information for the selected clone set. The following are the parts of the Clone Set Viewer:

Metric Value Panel: The Metric Value Panel displays all the metric values of the clone set.

Code Fragment List: The Code Fragment List displays a list of all the code fragments included in the clone set. Each line of the list consists of three types of information: (1) the path to the file including the code fragment; (2) the location of the code fragment in the file, including begin line number, begin column number, end line number, and end column number; (3) the number of tokens included in the code fragments, which implies the length of the code fragment.

Source Code View: The Source Code View works cooperatively with the Code Fragment List. Users obtain the actual source code of a code fragment selected in the Code Fragment List. In Source Code View, the code fragment is emphatically displayed.

NRV/NAV List: The NRV/NAV List displays a list of all the variables counted for the metrics $NRV(S)$ and $NAV(S)$ of a code fragment selected in the Code Fragment List. The information consists of three elements, (1) name of the variable, (2) type of variable, and (3) number of uses.

4.3. Interactive analysis of refactoring-oriented code clones

Figure 8 illustrates the process of an interactive analysis of the *refactoring-oriented code clones* using Aries's GUI component. As a result of the interactive analysis, users can identify code clones that can be merged. The following describes what users have to do in each phase of the process. Module names in parentheses mean that the modules are used in the given phase.

PHASE 1 (*NRV/NAV Selector*, *Clone Unit Selector*): Users determine the types of variables that are to be used for measuring the metrics and the kinds of granularity units that are to be the targets of the refactoring operations. The reason for an appropriate choice is described in Section 4.2.

PHASE 2 (*Metric Graph*): Users filter code clones by changing the lower and upper limits of the metrics. The code clones satisfying the metrics conditions are listed in the Clone Set List.

In the current implementation, the Metric Graph has pre-defined conditions for the following refactoring patterns:

- *Extract Class/Method/SuperClass*,
- *Form Template Method*,
- *Move Method*,
- *Parameterize Method*,
- *Pull Up Constructor/Method*.

DOI: 10.1002/smr

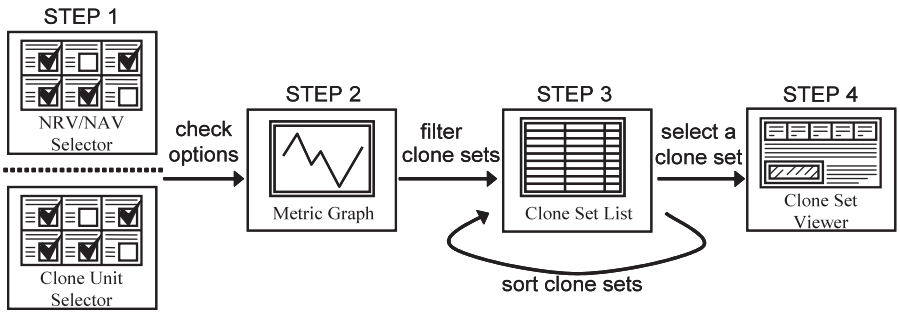

Figure 8. Process of merging code clones using Aries.

Each of the predefined conditions is a combination of the following:

- A set of variable types utilized for measuring metrics $NRV(S)$ and $NAV(S)$.
- A set of granularity units, which are the refactoring targets.
- Upper and lower limits of all of the metrics in the Metric Graph.

The predefined conditions are implemented as right-click menus in the Metric Graph. By choosing a predefined pattern, users can get clone sets satisfying those conditions. Predefined patterns make it easier to filter code clones.

PHASE 3 (*Clone Set List*): Users select a clone set. The detailed information for the selected clone set is available in the Clone Set Viewer. This list can sort clone sets in ascending or descending order of arbitrary metric, which may help users to select the code clones that they are interested in.

PHASE 4 (*Clone Set Viewer*): Users determine whether to merge the code clones or not, based on the detailed information of the clone set selected in the previous phase. If the users decide not to merge the code clones, they can use either CASE 1 or CASE 2:

Case 1: If they want to analyze another code clone with the same conditions, they can go back to PHASE 3;

Case 2: If they want to analyze another code clone with different conditions, they go back to either PHASE 1 or PHASE 2.

5. CASE STUDY

5.1. Target and configuration

Ant [13] (version 1.6.0) was chosen as the target for two reasons. First, Ant is written in Java language. As previously mentioned, the current implementation can only handle Java language. Second, the Ant package includes many test cases that can be used to confirm that Ant's external behavior has not been changed due to refactoring.

Ant 1.6.0 includes 627 source files, and its size is about 180 000 lines of code (LOC). In this case study, 30 tokens were used as the minimum token length of code clone, that is, 30 tokens correspond to about five lines of code.

DOI: 10.1002/smr

Aries detected 154 clone sets consisting of the *refactoring-oriented code clones* within 2 min. In this case study, the detected clone sets were filtered using the conditions of the *Extract Method* and the *Pull Up Method* described in Section 3.4. Fifty-nine of the code clones satisfied the conditions of the *Extract Method*, and 18 satisfied the conditions of the *Pull Up Method*.

The 59 clone sets were manually assigned to groups (EG1) to (EG5) described in Section 3.4. Similarly, the 18 clone sets were assigned to groups (PG1) to (PG3). After classification, all of the clone sets were classified into three groups, (EG1), (EG2), and (EG4), and a group, (PG2), for a total of 62 clone sets were merged.

After merging each clone set, regression tests were performed to confirm that the refactoring efforts had not changed the external behavior of Ant. In the regression test process, all 220 test cases included in the Ant package were used. These test cases were conducted with JUnit [14], which is one of the unit testing frameworks. Since it only took 4 min to perform all the test cases, it was very easy to perform regression tests.

The remainder of this section describes the details of both pattern applications.

5.2. Result of the Extract Method

As described above, 59 clone sets were obtained as a result of filtering with the conditions of the *Extract Method* described in Section 3.4. The source code of all clone sets was examined, and each of the clone sets was classified into one of five groups (EG1) to (EG5), which are described in Section 3.4. Then, the individual clone set in groups (EG1), (EG2), and (EG4) was merged.

Three clone sets were classified as (EG1). Figure 9(a) shows a code fragment included in (EG1). In this if-statement clone, no externally defined local-variable was utilized, so that it was very easy to merge the clone set into a new method in the same class.

Thirty-four clone sets were classified as (EG2). Figure 9(b) shows a code fragment included in (EG2). In this if-statement clone, variable javacopts was a field-member-of-its-class and variable genicTask was a local-variable; hence, it was necessary to set genicTask as a parameter of the extracted method to merge this clone set into the same class.

No clone set was classified as (EG3).

Fifteen clone sets were classified as (EG4). Figure 9(c) shows a code fragment included in (EG4). In this if-statement clone, variable iSaveMenuItem was externally defined. The code fragment included a reference and an assignment to the variable; hence, it was necessary to set iSaveMenuItem as a parameter of the extracted method, and to add a return-statement to reflect the result of the assignment to the caller place.

Seven clone sets were classified as (EG5). Figure 9(d) shows a code fragment included in (EG5). In this if-statement clone, there were two return-statements, so that an unreasonable amount of work would be required to extract this code. In the implementation of Ant 1.6.0, the return-statements are instructions for getting out from the existing method. If the code fragment is naively extracted as a new method, the return-statements become instructions for getting out from the extracted method. Simple method extraction changes the role of the return-statements. In this case study, these seven clone sets were not merged, since merging them would be highly dependent on the skill of an individual programmer.

Thus, in the case of the *Extract Method*, 52 of 59 clone sets could be refactored using only simple operations, such as extracting a method, adding parameters, and adding a return-statement.

DOI: 10.1002/smr

```
if (!isChecked()) {
    // make sure we don't have a circular reference here
    Stack stk = new Stack();
    stk.push(this);
    dieOnCircularReference(stk, getProject());
}
```

(a)

```
if (javacopts != null && !javacopts.equals("")) {
    genicTask.createArg().setValue("-javacopts");
    genicTask.createArg().setLine(javacopts);
}
```

(b)

```
if (iSaveMenuItem == null) {
   try {
      iSaveMenuItem = new MenuItem();
      iSaveMenuItem.setLabel("Save BuildInfo To Repository");
   } catch (Throwable iExc) {
      handleException(iExc);
   }
}
```

(c)

```
if (name == null) {
    if (other.name != null) {
        return false;
    }
} else if (!name.equals(other.name)) {
    return false;
}
```

(d)

Figure 9. Examples of the *Extract Method*: (a) example of the *Extract Method* in (EG1); (b) example of the *Extract Method* in (EG2); (c) example of the *Extract Method* in (EG4); and (d) example of the *Extract Method* in (EG5).

DOI: 10.1002/smr

5.3. Result of the Pull Up Method

As described above, 18 clone sets were obtained as a result of filtering with the conditions of the *Pull Up Method*. The source code of all code clones was examined to classify the code clones into three groups, (PG1) to (PG3), which are described in Section 3.4. In this case study, no clone set was classified into (PG1).

Ten clone sets were classified as (PG2). Figure 10(a) shows a code fragment included in (PG2). In this method clone, method getCommentFile was invoked twice, and the method was defined in the same class. Aries pointed out the invocation as a reference of the external defined variable although the variable 'this' was omitted in the source code. Variables this and FLAG_COMMENTFILE, which were field-members-of-its-class, were externally defined, so that this clone set was pulled up to the common base class after adding two parameters.

Eight clone sets were classified as (PG3). Figure 10(b) illustrates a code fragment included in (PG3). This method invoked method checkOptions, which was defined in the same class. Other invoked methods were defined in the common base class. Variable commandLine, which was an argument of checkOptions, was defined in this code fragment. Each class includes the code fragments of this clone set defined method checkOptions, and each of the code fragments invoked the method checkOptions defined in the same class. Different method invocations prevented this clone set from merging with the *Pull Up Method*. However, it was assumed that the *Form Template Method* pattern could be applied to this group. This would be done by first moving the code fragment to the common base class and then defining an abstract method named checkOptions in the common base class.

In the case of the *Pull Up Method*, 10 of 18 clone sets could be refactored using only simple operations, moving the method to a common base class and adding parameters.

6. DISCUSSION

This section discusses this proposal from several viewpoints.

6.1. Granularity units of the refactoring-oriented code clones

The reasons that the *refactoring-oriented code clones* provided by Aries are the granularity units of the programming language are as follows:

- All granularity units have their own variable scope, which means that all variables declared in a granularity unit can be accessed only in it. This is a big advantage when a granularity unit is moved to another place in the software system because these variables can be ignored.
- Identification of granularity units from the source code requires only a lightweight analysis, like matching opening braces and closing ones or building ASTs. Aries builds ASTs and identifies locations of all the granularity units. The reason for building ASTs is that Aries collects the variable information for measuring metrics $DCH(S)$, $NRV(S)$, and $NAV(S)$, in addition to locating the granularity units. To get only the locations of all the granularity units, matching opening and closing braces is adequate.

DOI: 10.1002/smr

```
private void getCommentFileCommand(Commandline cmd) {
   if (getCommentFile() != null) {
      /* Had to make two separate commands here because
         if a space is inserted between the flag and the
         value, it is treated as a Windows filename with
         a space and it is enclosed in double quotes (").
         This breaks clearcase.
      */
      cmd.createArgument().setValue(FLAG_COMMENTFILE);
      cmd.createArgument().setValue(getCommentFile());
   }
}
```

(a)

```
public void execute() throws BuildException {
   Commandline commandLine = new Commandline();
   Project aProj = getProject();
   int result = 0;

   // Default the viewpath to basedir if it is not specified
   if (getViewPath() == null) {
      setViewPath(aProj.getBaseDir().getPath());
   }

   // build the command line from what we got. the format is
   // cleartool checkin [options...] [viewpath ...]
   // as specified in the CLEARTOOL.EXE help
   commandLine.setExecutable(getClearToolCommand());
   commandLine.createArgument().setValue(COMMAND_CHECKIN);

   checkOptions(commandLine);

   result = run(commandLine);
   if (Execute.isFailure(result)) {
      String msg = "Failed executing: " +
                   commandLine.toString();
      throw new BuildException(msg, getLocation());
   }
}
```

(b)

Figure 10. Examples of the *Pull Up Method*: (a) example of the *Pull Up Method* in (PG2) and (b) example of the *Pull Up Method* in (PG3).

DOI: 10.1002/smr

For identifying refactorable code clones more precisely, the PDG-based technique is a better option. The technique can detect semantically identical code clones even if they are non-contiguous in the source code. However, the PDG-based technique is more expensive, and, thus, cannot be realistically used to code clones from middle-scale or large-scale software systems. On the other hand, Aries can detect the *refactoring-oriented code clones* in a practical time frame, even though the precision of Aries is less than that of the PDG-based technique.

6.2. Renamed identifiers in the refactoring-oriented code clones

Aries measures the metrics $DCH(S)$, $NAV(S)$, and $NRV(S)$ from the *refactoring-oriented code clones*. However, these metrics are not adequate conditions for representing refactoring possibility. *Refactoring-oriented code clones* can be renamed code clones, that is, identifiers utilized in them can be different from each other. In the case where only variable names are different, they can be merged because the logic is identical. However, in the case where the variable types are different, it is difficult to merge them because the logic is different. Users who intend to perform refactorings with Aries have to investigate whether the provided code clones can really be merged, in addition to the semantic perspective investigation, such as whether merging them can reduce maintenance costs in the future.

6.3. Internal logic of the refactoring-oriented code clones

The difficulty of merging *refactoring-oriented code clones* depends on the internal logic of the code clones and can be shown using an Ant case study. As shown in the case of (EG5), if the return-statements are in the target code fragment, it is necessary to make an effort to handle the return-statements. In the implementation of Ant 1.6.0, the return-statements are instructions for getting out from the existing method. If the target code fragments are naively extracted, the return-statements will be instructions for getting out from the extracted method. In the case where the target code fragments include break-statements or continue-statements[‖], special techniques are required to extract them as a case of return-statements.

6.4. Users' responsibility in merging code clones with Aries

Before utilizing Aries, users have to understand the three metrics $DCH(S)$, $NRV(S)$, and $NAV(S)$ that were proposed in this paper. These represent the features of code fragments that the existing metrics do not. If users do not understand these metrics, then they will be unable to properly filter the *refactoring-oriented code clones* based on their requirement.

Using Aries, users know the code clones that satisfy the conditions their conditions. Aries shows the refactoring candidates based not on **adequacy** but on **possibility**. Thus, users have to consider the adequacy of merging the candidates themselves. Users determine whether or not they merge the code clones by reference to the various kinds of information provided by Aries.

[‖] In Java language, break-statements and continue-statements can have labels, which makes the execution sequence complicated.

DOI: 10.1002/smr

7. RELATED WORKS

This section describes other research efforts related to code clone refactoring**. However, refactoring is not a silver bullet for code cloning. This paper also presented some techniques for managing code clones since, for various reasons, some code clones cannot or should not be merged.

7.1. Techniques for merging code clones

Fowler, a pioneer in the field of refactoring, mentioned in his book that the *number one in the stink parade is duplicated code* [9]. He also presented some sets of operations for merging code clones. The operations described in Figure 3 are taken from his book. This book helped to formulate the need for the metric $DCH(S)$.

CloneDr, which is an implementation of the AST-based detection technique, presents not only the locations of code clones but also the forms of merged code fragments [5]. The forms help users to understand the operations that are required to merge code clones. However, the tool does not care about the positional relationship between code clones in the class hierarchy. Users themselves have to investigate where the merged code fragment can be placed, whereas the proposed method itself renders to the users the required information. Metric $DCH(S)$ is an indicator of the location of the merged code fragment. If the value is 0, the merged code fragment can be placed in the same class. If the value is 1, the merged code fragment can be placed in the direct base class.

Balazinska *et al.* proposed a refactoring method for the duplicated methods [15]. Their method provides the differences between code clones, which helps users to determine whether code clones can be merged or not. In addition, their method measures the coupling between a duplicated method and its surrounding code.

Cottrell *et al.* implemented a tool that visualizes the detailed correspondences between a pair of classes. The classes are generalized to form an intermediate, AST-like structure that distinguishes between what is common and what is specific to each class. The specific instructions will influence the degree of relatively between the classes. The tool works after users identify two classes that should be merged. This tool can be combined with Aries into a single refactoring process, since the two tools support different steps of the refactoring process.

Jarzabek proposed and implemented XVCL, which is a framework to merge duplicated parts of a software system into so-called meta components [16,17]. The technique can handle complicated code clones that include abstraction functions built into the programming language. Meta components include merged code fragments and instructions for deploying the code fragments into the raw source code. Some papers reported that XVCL plays an important role in merging code clones [18,19].

Komondoor *et al.* proposed an algorithm for procedure extraction. The inputs to the algorithm are as follows:

- The CFG (control-flow graph) of a procedure,
- A set of nodes in the CFG.

**About the validity of this proposal, please refer to Section 2.

DOI: 10.1002/smr

The goal of the algorithm is to revise the CFG with the following conditions:

- The set of nodes that are extractable from the revised CFG,
- The revised CFG is semantically equivalent to the original CFG.

In implementing this algorithm, they introduced certain heuristics for enhancing scalability. Although the algorithm has a worst-case exponential time complexity, their experimental results indicated that it may work well in practice. However, the algorithm can be applied only to a single code clone. Different techniques are needed to determine how two or more code clones can be extracted as a single procedure while preserving semantics.

7.2. Techniques for managing code clones

Kim *et al.* performed experiments on the repositories of open-source software systems to investigate how code clones appear and disappear [20]. The experimental results revealed the following points, which partly motivated the proposed simultaneous modification support method [21]:

- Some code clones are short-lived. Refactoring (merging) them does not improve their maintainability.
- Most long-living code clones are not suited to be refactored because there is no abstraction function of the programming language that can handle them.

Kapser and Godfrey also suggested that, based on their experience, code clones are not always harmful [22]. They reported several situations where code duplication is a reasonable or even beneficial way to handle large-scale complex software systems. For example, when developing a new driver for a certain hardware family, there may already be drivers to handle some other hardware families. However, there are often considerable differences in the functionalities or features between the families. It is very risky and unrealistic to merge code clones in the drivers. Nevertheless, if a bug were found in the driver of a certain hardware family, it would be very likely that there are the same bugs in the drivers of the hardware families having similar features to this family. Thus, it would be necessary to find and correct each of the bugs in the drivers without overlooking any of them. In cases like this, simultaneous modification can be a great support for software maintenance.

Balazinska *et al.* reported that differences between code fragments tend to hinder applying re-engineering actions (refactorings) to them [23]. It was shown that strictly identical or superficially different code clones are easier to re-engineer than code clones including other types of differences. In other words, code clones including some gaps are difficult to refactor. However, CCFinder can detect only identical or identifier-replaced code clones, that is, it cannot detect gapped code clones. It may be more effective to use another clone detection technique that can detect gapped code clones in the context of simultaneous modification. Some of these techniques/tools are metric-based detection [8] and CP-Miner [24].

Toomim *et al.* have proposed a simultaneous modification method on code clones included in the same clone set [25]. In their method, there is a database of code clone information in the backend of the editor program. When a code fragment included in a clone set is modified, other code fragments in the clone set are also simultaneously modified. Duala-Ekoko and Robillard have

DOI: 10.1002/smr

also proposed a simultaneous editing method [26]. The method identifies corresponding lines in code fragments that are similar to each other based on the Levenshtein distance[††] after the code fragments are detected. An implementation of the method has been fully integrated in Eclipse. At present, the method can handle only clone pairs. It cannot handle clone sets consisting of three or more code fragments. Neither method can be utilized in real software development or maintenance because they have a critical drawback, that is, they cannot identify the differences between code fragments to be edited simultaneously that contain small differences between the code fragments. Their methods work well only on identical code clones, which do not include different identifiers or re-ordered statements.

Mann suggested that it should be effective to track 'copy and paste' actions [27] that would enable the user to know from where arbitrary code fragments were derived and, then, simultaneously modify all of the code fragments derived from the same source. 'Copy and paste' is one source of code clones in the source code. For example, Kim *et al.* reported that developers perform block- or method-level copy-and-paste actions approximately 4 times per hour [28]. Balint *et al.* reported that inconsistencies often occur between the 'copied-and-pasted' code [29]. Tracking 'copy and paste' can potentially identify many code clones. The advantage of this method is that any type of code clone can be identified as long as the code clones are generated as a result of a 'copy and paste' action. Each code clone detection technique depends on its detection algorithm. In other words, it cannot detect all types of code clones. For this reason, tracking 'copy and paste' might be a good support for simultaneous modification.

8. CONCLUSION

In this paper, a new refactoring method for code clones was proposed and implemented as the software tool, Aries. The code clone detection of Aries is fast enough to apply it to middle-scale or large-scale software systems. A case study using Aries was performed on the open-source software system, Ant. Quantitative metrics for code clones proposed in this paper adequately characterized the code clones in Ant, and most of the code clones recommended by Aries could be refactored with simple operations.

In the future, more detailed analyses for code clones will be implemented. For example, there is a need to consider the effectiveness of refactoring. Currently, code clones are filtered based on whether they **can** or cannot be merged. If code clones are filtered based on whether they **should** or should not be merged, the refactoring support will become more effective.

ACKNOWLEDGEMENTS

This work has been conducted as a part of the EASE Project, Comprehensive Development of e-Society Foundation Software Program, and Grant-in-Aid for Exploratory Research (186500006), both supported by the Ministry of Education, Culture, Sports, Science and Technology of Japan. It has also been performed under Grant-in-Aid for Scientific Research (A)(17200001) supported by the Japan Society for the Promotion of Science.

[††]The Levenshtein distance is a method for comparing strings S_a and S_b based on the number of insertions, deletions, and substitutions required to transform S_a to S_b.

DOI: 10.1002/smr

REFERENCES

1. IEEE. *Standard for Software Maintenance*. IEEE Standard 1219, 1998.
2. Arthur L. *Software Evolution*: *The Software Maintenance Challenge*. Wiley: New York NY, 1988.
3. Yip SWL, Lam T. A software maintenance survey. *Proceedings of the 1st Asia-Pacific Software Engineering Conference*, December 1994; 70–79.
4. Baker BS. Parameterized duplication in strings: Algorithms and an application to software maintenance. *SIAM Journal on Computing* 1997; **26**(5):1343–1362.
5. Baxter I, Yahin A, Moura L, Anna M, Bier L. Clone detection using abstract syntax trees. *Proceedings of International Conference on Software Maintenance 98*, March 1998; 368–377.
6. Kamiya T, Kusumoto S, Inoue K. CCFinder: A multilinguistic token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering* 2002; **28**(7):654–670.
7. Komondoor R, Horwitz S. Using slicing to identify duplication in source code. *Proceedings of the 8th International Symposium on Static Analysis*, July 2001; 40–56.
8. Mayrand J, Leblanc C, Merlo E. Experiment on the automatic detection of function clones in a software system using metrics. *Proceedings of the International Conference on Software Maintenance 96*, November 1996; 244–253.
9. Fowler M. *Refactoring*: *Improving the Design of Existing Code*. Addison-Wesley: Reading MA, 1999.
10. Rysselberghe F, Demeyer S. Evaluating clone detection techniques from a refactoring perspective. *Proceedings 19th IEEE International Conference on Automated Software Engineering*, September 2004; 336–339.
11. Mens T, Tourwe T. A survey of software refactoring. *IEEE Transactions on Software Engineering* 2004; **30**(2):126–139.
12. Higo Y, Kamiya T, Kusumoto S, Inoue K. Method and implementation for investigating code clones in a software system. *Information and Software Technology* 2007; **49**(9–10):985–998.
13. Ant. Available at: http://ant.apache.org/ [5 August 2008].
14. JUnit. Available at: http://www.junit.org/ [5 August 2008].
15. Balazinska M, Merlo E, Dagenais M, Lagüe B, Kontogiannis K. Advanced clone-analysis to support object-oriented system refactoring. *Proceedings 7th IEEE International Working Conference on Reverse Engineering*, November 2000; 98–107.
16. XML-Based Variant Configuration Language—Technology for Reuse. Available at: http://xvcl.comp.nus.edu.sg/ [5 August 2008].
17. Jarzabek S. *Effective Software Maintenance and Evolution*: *Reused-based Approach*. CRC Press, Taylor & Francis: Boca Raton, London, 2007.
18. Jarzabek S, Shubiao L. Eliminating redundancies with a 'composition with adaptation meta-programming' technique. *Proceedings of ESEC-FSE'03 European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering*, September 2003; 237–246.
19. Jarzabek S, Li S. Unifying clones with a generative programming technique: A case study. *Journal of Software Maintenance and Evolution*: *Research and Practice* 2006; **18**(4):267–292.
20. Kim M, Sazawal V, Notkin D, Murphy GC. An empirical study of code clone genealogies. *Proceedings of the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, September 2005; 187–196.
21. Higo Y, Ueda Y, Kusumoto S, Inoue K. Simultaneous modification support based on code clone analysis. *Proceedings of the 14th Asia-Pacific Software Engineering Conference*, December 2007; 262–269.
22. Kapser C, Godfrey MW. 'Cloning considered harmful' considered harmful. *Proceedings of the 13th Working Conference on Reverse Engineering*, October 2006; 19–28.
23. Balazinska M, Merlo E, Dagenais M, Lagüe B, Kontogiannis K. Measuring clone based reengineering opportunities. *Proceedings 6th IEEE International Symposium on Software Metrics*, November 1999; 292–303.
24. Li Z, Lu S, Myagmar S, Zhou Y. CP-Miner: Finding copy-paste and related bugs in large-scale software code. *IEEE Transactions on Software Engineering* 2006; **32**(3):176–192.
25. Toomim M, Begel A, Graham S. Managing duplicated code with linked editing. *Proceedings IEEE Symposium on Visual Languages and Human-Centric Computing*, September 2004; 173–180.
26. Duala-Ekoko E, Robillard MP. Tracking code clones in evolving software. *Proceedings of the 29th International Conference on Software Engineering*, May 2007; 158–167.
27. Mann ZA. Three public enemies: Cut, copy, and paste. *IEEE Computer* 2006; **39**(7):31–35.
28. Kim M, Bergman L, Lau T, Notkin D. An ethnographic study of copy and paste programming practices in OOPL. *Proceedings of 2004 International Symposium on Empirical Software Engineering*, August 2004; 83–92.
29. Balint M, Girba T, Marinescu R. How developers copy. *Proceedings 14th IEEE International Conference on Program Comprehension*, June 2006; 56–68.

DOI: 10.1002/smr

AUTHORS' BIOGRAPHIES

Yoshiki Higo received his master's degree and PhD degree in information science and technology from Osaka University in 2004 and 2006, respectively. At present he is an assistant professor at the Osaka University. His research interests include code clone analysis, software metrics, and refactoring support techniques. He is a member of the IEEE, IPSJ, and IEICE.

Shinji Kusumoto received his BE, ME, and DE degrees in information and computer sciences from Osaka University in 1988, 1990, and 1993, respectively. At present he is a professor in the Graduate School of Information Science and Technology at Osaka University. His research interests include software metrics and software quality assurance techniques. He is a member of the IEEE, the IEEE Computer Society, IPSJ, IEICE, and JFPUG.

Katsuro Inoue received his BE, ME, and DE degrees in information and computer sciences from Osaka University, Japan, in 1979, 1981, and 1984, respectively. He was an assistant professor at the University of Hawaii at Manoa from 1984–1986. He was a research associate at Osaka University from 1984–1989, an assistant professor from 1989–1995, and has been a professor since 1995. His interests are in various topics of software engineering such as software process modeling, program analysis, and software development environment. He is a member of the IEEE, the IEEE Computer Society, and the ACM.

DOI: 10.1002/smr