Identifying Refactoring Opportunities for Removing Code Clones with A Metrics-based Approach

Yoshiki Higo, Shinji Kusumoto and Katsuro Inoue
Graduate School of Information Science and Technology
Osaka University, Japan

1 Introduction

Recently, code clones have attracted a great deal of attention in software engineering, and various studies of code clone-related research have been performed (Roy et al., 2009). A code clone is a code fragment that is either identical or similar to other code fragments in the source code. Code clones are generated for various reasons, including copy-and-paste programming or implementing stereotyped code (Baxter et al., 1998).

The presence of code clones is generally considered to make it difficult to maintain the consistency of the code. For example, if we modify a code fragment with code clones in order to fix bugs or add new functionalities, it is necessary to determine whether the code clones should be modified simultaneously with the code fragment being modified. In addition, the presence of code clones has been reported to have a negative impact on software maintenance (Juergens et al., 2009). Monden et al. investigated the relations between code clones and the number of revisions of the source files on a very large legacy system written in COBOL (Monden et al., 2002). The results revealed that the following types of files tended to be revised more frequently than other types: (1) source files that are 80% or more duplicated, and (2) source files that includes large code clones (200 lines or more).

Refactoring is a set of operations to improve maintainability, understandability or other attributes of a software system without changing its external behavior. Refactoring has recently received a lot of attention. A code clone is one of the typical *Bad Smells* in the refactoring process (Fowler, 1999). Code clone detection can be perceived as the identification of code fragments to be refactored. From a practical standpoint, it is very hard to identify which code clones should 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, which makes it difficult to identify which code clones can be merged and how best to merge them.

In this chapter, we provide a new method for merging code clones. The method consists of 2 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 how the 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. As well, this chapter 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 Process of Code Clones Refactoring

Herein, we introduce a typical process for refactoring, which can be used for removing code clones. Reference (Mens & Tourwe, 2004), which is a survey of refactoring technologies, states 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 the consistency between the refactored program code and other software artifacts, such as documentation, design documents, requirements specifications, and tests.

Especially, STEPs 1 and 2 require specialized techniques on code clone removal. The remainder of this section describes existing techniques that support STEPs 1 and 2 for removing code clones.

2.1 STEP 1: Identify where the software should be refactored

In order to remove code clones, we have to know where code clones exist in a software system. A number of code clone detection tools have been developed before now. The detection tools reveal the position of the detected code clones in the source code. Since there is neither a generic nor a strict definition of code clone, each detection tool has its own unique definition for code clones. Consequently, different code clones are detected by different detection tools for the same source code. 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.
- **AST (Abstract Syntax Tree)-based technique** After building an AST from the source code, subtrees having the same structure are detected as code clones.
- **PDG (Program Dependency Graph)-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 program units, for example the functions, methods, or classes 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, that is, line-based, token-based, and metric-based, from a refactoring perspective (Rysselberghe & Demeyer, 2004).

- **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 are, the more they tend to be detected as 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 function is partly identical or similar to another part of a different function, they are not detected, since code clone detection is performed on 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, even greater 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 technique 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 token-based code clones are not suited to be refactored for the following reasons:

  - Token-based technique identifies duplicated token sequences as code clones, so that 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 group of code clones because of identifier replacement.

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. Therefore, the proposed method in this chapter 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.

Besides, 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 simple block. 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 is, the more they need to be refactored since large-scale software includes many complicated logics, which are intertwined with one another. However, it is difficult to manually detect code to be refactored from the large number of lines of code. Scalability is an important property of the tools that detects code to be refactored.

Burd and Bailey reported that CLONEDR, which is an implementation of the AST-based technique, has good code clone detection precision[1], but it could only detect a small part of the code clone set detected using other detection techniques (Burd & Bailey, 2002). 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

[1] However, it should be noted that they did not evaluate code clone detection techniques from a refactoring perspective.

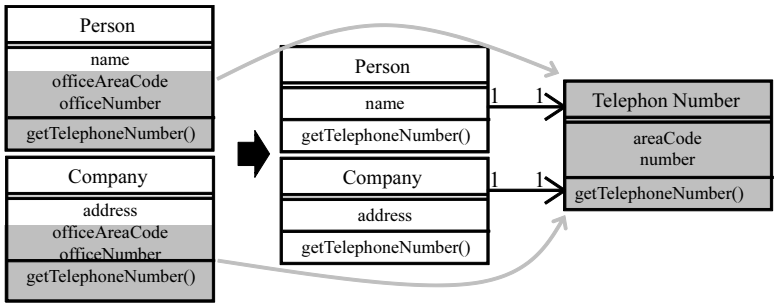


Figure 1: Example of Extract Class

```
void printTaxi(int amount){
        String name = getTaxiName();

        System.out.println("name:" + name);
        System.out.println("amount:" + amount);
}

void printBus(int amount){
        String name = getBusName();

        System.out.println("name:" + name);
        System.out.println("amount:" + amount);
}
```

```
void printTaxi(int amount){
        String name = getTaxiName();
        print(name, amount);
}

void printBus(int amount){
        String name = getBusName();
        print(name, amount);
}

void print(String name, int amount){
        System.out.println("name:" + name);
        System.out.println("amount:" + amount);
}
```

Figure 2: Example of Extract Method

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 CLONRDR 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.

2.2 STEP 2: Determine which refactoring should be applied to the identified places

Once we obtained code clones, we have to determine how to remove them. Fowler, who is a pioneer of refactoring, introduce many ways for refactoring in his book (Fowler, 1999). Some of them can be used for removing code clones, and herein we introduce them:

- **Extract Class/SuperClass** If you find a class level duplication, the "Extract Class" or "Extract SuperClass" patterns may be able to be performed for remove such a duplication. If duplicated classes extend different base classes, the duplication can be removed by "Extract Class". If not, "Extract SuperClass" should be better for removal. Figure 1 shows an example of "Extract Class". In this figure, new class "TelephonNumber" 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 duplicated classes are already inherited from another class.

- **Extract Method** If two or more methods in a single class are partly duplicated, the "Extract Method" is a simple and practical solution for removal. Figure 2 shows an example of "Extract Method".

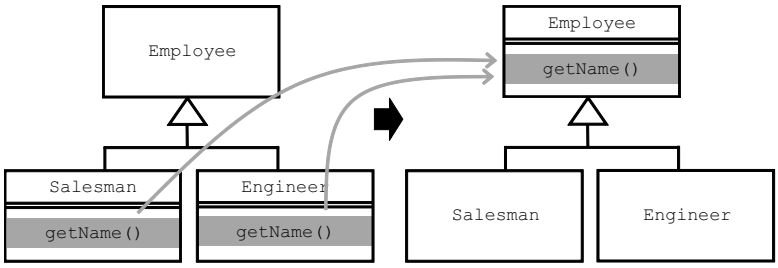


Figure 3: Example of Pull Up Method

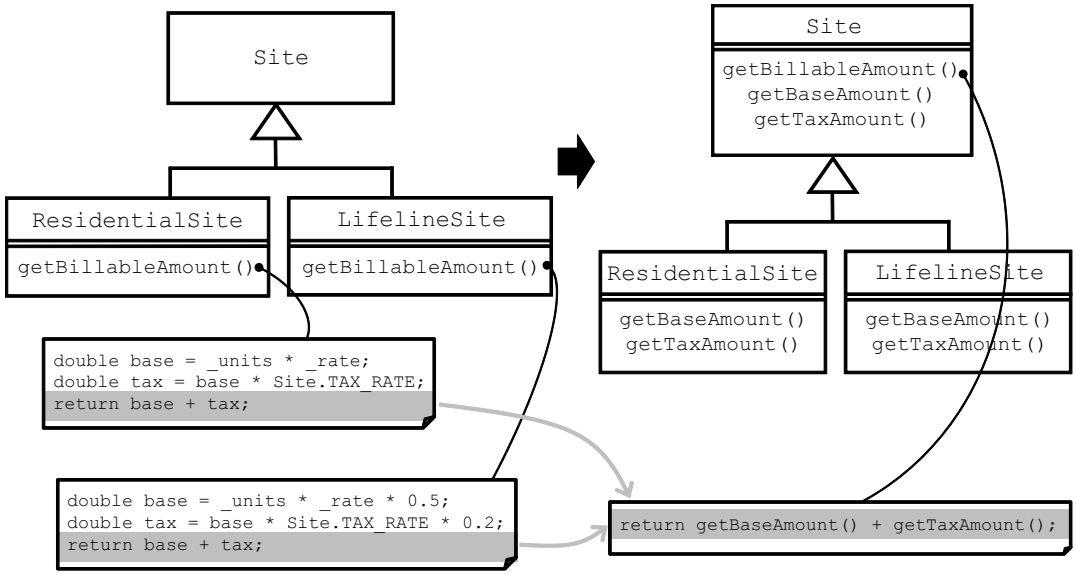


Figure 4: Example of Form Template Method

- **Pull Up Method** If there are duplicated methods in different classes that extend a common base class, the "Pull Up Method" is a good way for duplication removal. Figure 3 shows an example of "Pull Up Method".
- **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. Firstly, 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. 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 better 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 case, the “Move Method” is a good solution for removal.
- **Parameterize Method** If some methods in a single class are duplicated, the duplication may be removed by the “Parameterized Method”.
- **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.

3 Proposed Method for Merging Code Clones

This section provides a new method for merging code clones. The method consists of 3 steps, *Detection*, *Extraction*, and *Measurement*. Firstly Subsections 3.1, 3.2 3.3 explain each step, then, subsection 3.4 describes 2 refactoring examples.

3.1 STEP 1: Detection

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 (Balint et al., 2006). Thus, it is expected that such techniques can detect identifier-replaced code fragments. In the proposed method, code clones detected by an existing detection technique are called *general code clones*.

3.2 STEP 2: Extraction

The following syntactical 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 5 illustrates an example. In this figure, there are 2 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, while in code fragment *B*, operations 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 than extracting else-if blocks from a refactoring viewpoint.

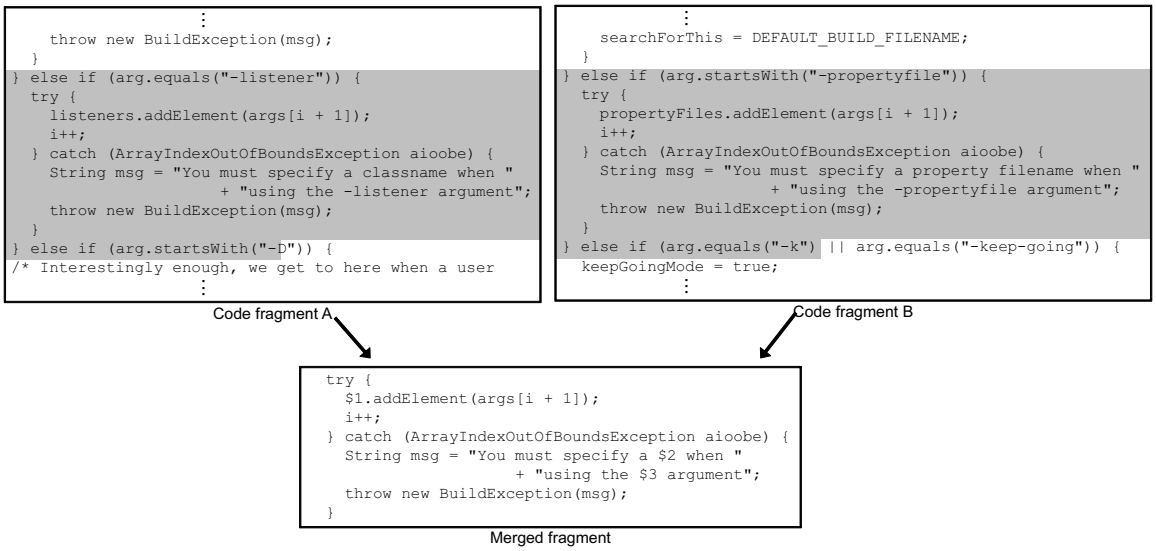


Figure 5: Example of merging 2 code fragments

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

3.3 STEP 3: Measurement

Several software metrics are measured for the refactoring-oriented code clones. Those 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.

Subsections 3.3.1 and 3.3.2 describe the proposed metrics which can be used to detemine a stragegy for merging code clones. Also, subsection 3.4 shows 2 examples of merging code clones using these metrics.

3.3.1 Locational Relationship in the Class Hierarchy

On source code written in Java language, the strategy for merging code clones depends on their locational relationship in the class hierarchy as follows:

- **CASE 1:** If code clones are in a single class, they can be extracted as a new method in the same class as shown in Figure 2.
- **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 as shown in Figure 3.
- **CASE 3:** If code clones are scattered in different classes, a new class is required to merge them as shown in Figure 1.

In order to automatically indicate an appropriate way to merge code clones, the locational 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, $\cdots$, f_n. A class including code fragment f_i is represented as c_i.

- **CASE A:** If classes c_1, c_2, $\cdots$, 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 as ∞.

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

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

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

3.3.2 Coupling between a Code Clone and its Surrounding Code

The principle of strategy for merging code clones is migration of duplicated code to another place. To migrate implemented code, it is desirable that the code has low coupling with its surrounding code. Assume that the Extract Method[2] is to be performed. To apply this pattern, 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).

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

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

$$NAV(S_2) = \frac{1}{m}\sum_{i=1}^{m} t_i \tag{3}$$

Intuitively, NRV(S_2) represents the average of 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 are code clones that

[2]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 (Fowler, 1999). It also can be applied to code clones to merge them.

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[3]:

$$NRV(S_2) = s_1 = s_2 = \cdots = s_m, \quad (4)$$

$$NAV(S_2) = t_1 = t_2 = \cdots = t_m. \quad (5)$$

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 such a case, the definitions of formulae 2 and 3 for computing NRV(S_2) and NAV(S_2) must be used.

3.4 Examples of Merging Code Clones

Locational relationship metric, DCH(S), represents where in the class hierarchy the common code can be factored out to, and coupling metrics NRV(S) and NAV(S) represent the couplings between code clones and their surrounding code. These metrics can represent the possibility of some refactoring patterns in which code clones are merged in another place of the class hierarchy.

The remainder of this subsection presents 2 sets of conditions of the syntactical 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 we want to merge code clones by the "Extract Method", then a typical set of conditions could be as follows:

- **EC1 (Extract Method Condition 1):** The target syntactical units 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" can be categorized 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, (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.

[3]In the implementation of the proposal, a code clone detection tool, CCFINDER (Kamiya et al., 2002) is used. CCFINDER detects only the structurally identical code clones, so that NRV(S) and NAV(S) are calculated using these formulas.

- **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, (3) add a return-statement for passing the value of the local variable.

- **OTHER:** Other operations 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 3 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 the PARAMETER, RETURN, and OTHER operations depends on the internal logics of them. If a clone set satisfies all the conditions (EC1) to (EC3), it is categorized into one of the following groups, (EG1) to (EG5). Table 1 represents the relationships between the 5 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 such 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 1 are "do no care".

	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

Table 1: Classifying code clones satisfying (EC1) to (EC3)

3.4.2 Example 2: Pull Up Method

If a user wants to merge code clones with the "Pull Up Method", the following conditions are reasonable:

- **PC1 (Pull Up Method Condition 1)** The target syntactical unit 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.

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.

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** Other operations 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 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.

The MOVE operations are required by any of the code clones in performing the "Pull Up Method", whereas 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 2 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 was 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 in (PG3) require the PARAMETER operations or not, so that the corresponding cell of Table 2 is "do no care".

3.5 Features of the Proposed Method

Herein, we discusse the proposed method from several viewpoints.

3.5.1 Syntactical Units of the Refactoring-oriented Code Clones

In the proposed method, the refactoring-oriented code clones are the syntactical units of programming language, which is for the following reasons:

- All syntactical units have their own variable scope, which means that all variables declared in a syntactical unit can be accessed only in it. This is a big advantage when a syntactical unit is moved to another place in the software system because these variables can be ignored.

- Identification of syntactical units from the source code requires only a lightweight analysis, like matching opening braces and closing ones or building ASTs. The proposed method builds ASTs and identifies locations of all syntactical units. The reason for building ASTs is that the proposed method collects the variable information for measuring metrics DCH(S), NRV(S), and NAV(S), in addition to locating the syntactical units. To get only the locations of all syntactical units, matching opening and closing braces is sufficient.

For more precisely identifying refactorable code clones, the PDG-based technique is better. 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 applied to middle-scale or large-scale software systems. On the other hand, the proposed method can detect the refactoring-oriented code clones in a practical timeframe, even though the precision of the proposed method is less than that of the PDG-based technique.

	MOVE	PARAMETER	OTHER
PG1	required	-	-
PG2	required	required	-
PG3	required	do no care	required

Table 2: Classifying code clones satisfying (PC1) to (PC3)

3.5.2 Renamed Identifiers in the Refactoring-oriented Code Clones

The proposed method measures the metrics DCH(S), NAV(S), and NRV(S) from the refactoring-oriented code clones. However, these metrics are not sufficient 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 are going to perform refactorings with the proposed method 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.

3.5.3 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. For example, if there is a return-statement in a clone instance, special techniques are required to extract it as a new method. Existing return-statements are instructions for getting out from existing methods. If a code clone including a return-statement are naively extracted, the return-statement will be instructions for getting out from the extracted method. The proposed method has no special technique for handling such a code clone.

3.5.4 Users's Responsibility in Merging Code Clones with ARIES

Prior to using the proposed method, users have to learn and understand 3 metrics DCH(S), NRV(S), and NAV(S). These represent features of code fragments that 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.

The proposed method enables users to know which code clones satisfy the conditions their conditions. The proposed method 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 the proposed method.

4 ARIES: Refactoring Support System based on Code Clone Analysis

This section introduces ARIES, which is an implementation of the proposed method. ARIES assists users in merging code clones. However, the assistance is only a part of refactoring process. The assistance of ARIES covers only STEPs 1 and 2, which are described in Section 2. The user must choose whether the suggested refactorings should be performed or not. ARIES consists of 2 components, an analysis component and a GUI component, which are shown in Figure 6. Subsections 4.1 and 4.2 describe the components. After that, subsection 4.3 shows the process of merging code clones with ARIES.

4.1 Analysis Component of ARIES

The analysis component takes 2 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. The analysis component utilizes an existing code clone detection

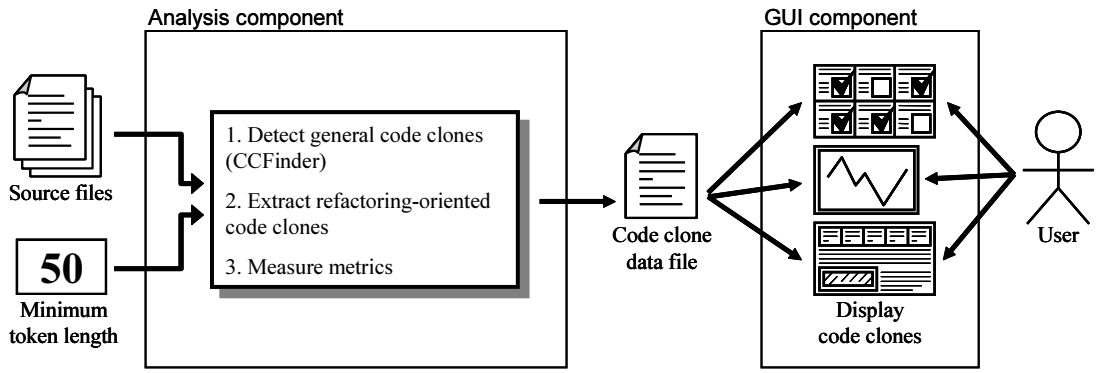


Figure 6: System Overview

tool, CCFINDER (Kamiya et al., 2002). 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.

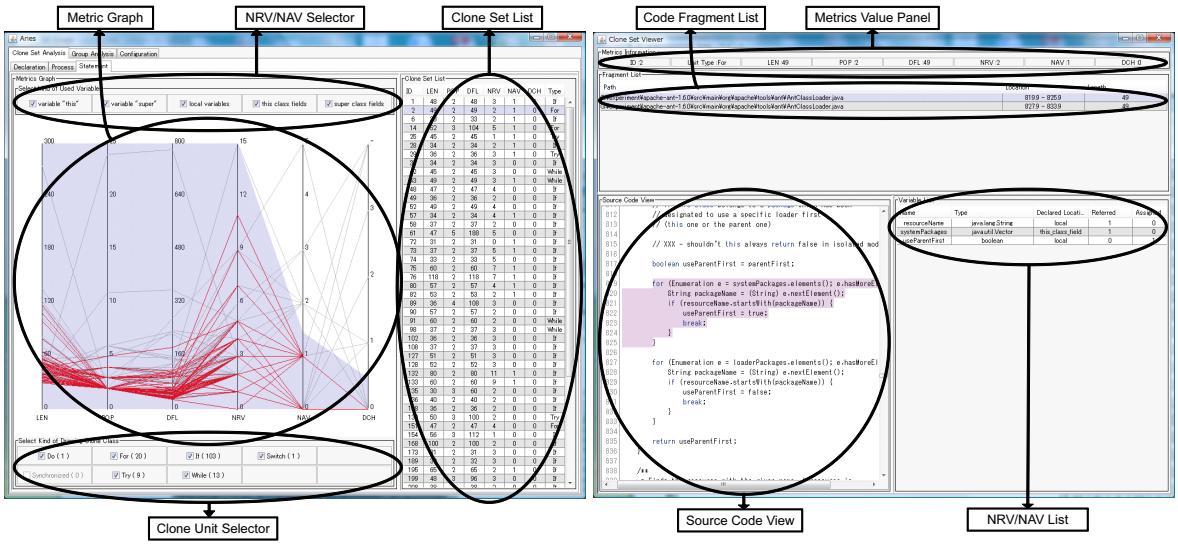
Firstly the general code clones are detected by CCFINDER, and then the analysis component parses the source code and builds the ASTs. The syntactical units (described in subsection 3.2) located in the general code clones are identified by comparing the locations of syntactical units with the general code clones. Identified syntactical 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 using 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 that the fact that it parses twice. This implies that, the implementation is sufficient 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 7 shows snapshots of the GUI component. The Main Window (Figure 7(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 6 metrics. Three of the metrics, which are DCH(S), NRV(S), and NAV(S) were proposed in Section 3.3. The



(a) Main Window

(b) Clone Set Viewer

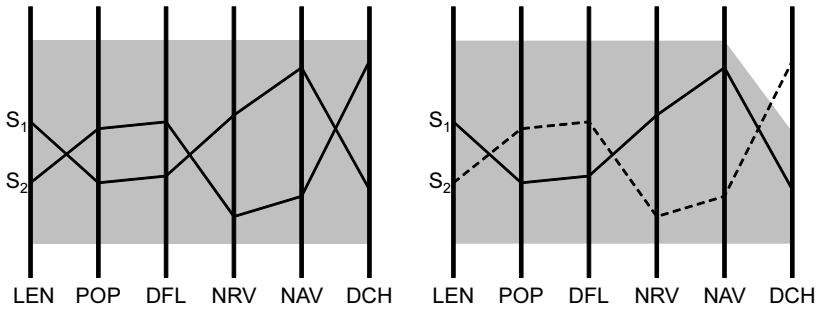
Figure 7: Snapshots of ARIES

other three metrics, which are LEN(S), POP(S), and DFL(S), were previously proposed in (Higo et al., 2007a). They are defined as follows:

- **LEN(S)** LEN(S) means the average size of code clones included in a clone set S. ARIES uses CCFINDER as a code clone detection engine. CCFINDER is a token-based detection tool. Consequently, The size of a code clone is the number of tokens forming the code clone. Large code clones should be a typical target of code clone refactoring.
- **POP(S)** POP(S) means the number of code clones included in a clone set S. A high POP(S) value means that duplicate code of S appear in many places in the system. Identifying code clones occurring frequently is useful for code clone refactoring.
- **DFL(S)** DFL(S) represent a value estimating how many tokens would be removed from source files when the code clones included in a clone set S are merged as a single method.

The Metric Graph allows users to filter out code clones that they are not interested in. Figure 8 illustrates how users filter out code clones using the Metric Graph. In Metric Graph, each metric has a parallel coordinate axis. Users can specify the upper and lower limits of each metric. The hatched region is the range bounded by the upper and 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 8(a)), all the metric values of both clone sets are in the hatched region, which implies that neither of them is filtered out. In the right graph (Figure 8(b)), DCH(S_2) is larger than the upper limit of DCH, so that 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 selected



(a) Before selection (b) After selection

Figure 8: Metric Graph Model

from:

- field-member-of-its-class (including special variables “this” and “super”);
- field-member-of-parent-class;
- field-member-of-interface;
- local-variable;

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.

- **Clone Unit Selector** On the Clone Unit Selector, users determine which kinds of syntactical units are displayed in the Metric Graph. As described in subsection 4.1, there are 12 kinds of syntactical units in Clone Unit Selector. For example, if users are going to perform the “Pull Up Method”, they should select only method-unit, because this pattern is a set of operations on 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 syntactical units and metrics are listed in the Clone Set List. The list has a sorting function, which allows clone sets to be sorted in ascending or descending order of any of the metrics.

Figure 7(b) is a snapshot of the Clone Set Viewer, which 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 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 code fragments included in the clone set. Each line of the list consists of 3 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.

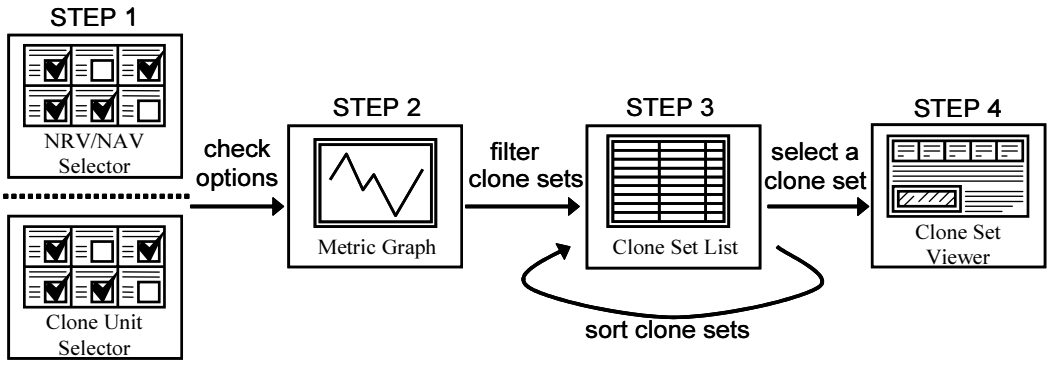


Figure 9: Process of merging code clones using ARIES

- **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 3 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 9 illustrates the process of an interactive analysis of the refactoring-oriented code clones using ARIES's GUI component. Through the interactive analysis with the GUI component, 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 step.

- **STEP 1 (NRV/NAV Selector, Clone Unit Selector)** Users determine which types of variables are used for measuring the metrics and which syntactical units are the targets of the refactoring operations. The reason for an appropriate choice are described in subsection 4.2.
- **STEP 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".

  Each of the pre-defined conditions is a combination of the followings.

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

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

- **STEP 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 they are interested in.

- **STEP 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 decided not to merge the code clones, they can go back to the previous steps: if they want to analyze another code clone with the same conditions, they can go back to PHASE 3; if they want to analyze another code clone with different conditions, they go back to either PHASE 1 or PHASE 2.

5 Experiment

5.1 Target and Configuration

We chose ANT[4] (1.6.0) as the experimental target for 2 reasons. Firstly, ANT is written in Java language. As previously mentioned, the current implementation can only handle Java language. Secondly, the ANT package includes many test cases that can be used to confirm that ANT 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 experiment, 30 tokens was used as the minimum token length of code clone. 30 tokens approximately correspond to about 5 lines of code.

The merging process was performed with the following steps:

- **STEP 1** Identifies code clones that are the target of refactoring with ARIES.
- **STEP 2** Classify the code clones into groups (EG1) to (EG5) and groups (PG1) to (PG3).
- **STEP 3** Selects a clone set from the identified code clones, and remove it by “Extract Method” or “Pull Up Method”.
- **STEP 4** Performs regression tests.
- **STEP 5** Goes back to STEP 3 if there is one or more clone set where STEP 3 is performed.

In STEP 1, ARIES detected 154 clone sets consisting of the refactoring-oriented code clones within 2 minutes. In this case study, the detected clone sets were filtered using the conditions of the “Extract Method” and the “Pull Up Method” described in subsection 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”. These 59 and 18 clone sets are the targets of refactoring in this experiment.

[4] http://ant.apache.org/

In STEP2, the 59 clone sets were manually assigned to groups (EG1) to (EG5) described in subsection 3.4. Similarly, the 18 clone sets were assigned to groups (PG1) to (PG3).

In STEPs 3 and 4, all of the clone sets classified into 3 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[5], which is one of the unit testing frameworks. Since it only took 4 minutes to perform all the test cases, it was very easy to perform regression tests.

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

5.2 Result of the "Extract Method"

We obtained 59 clone sets as a result of filtering with the conditions of the "Extract Method" described in subsection 3.4. The source code of all clone sets was examined, and each of the clone sets was carefully classified into either of 5 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 10(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 10(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, so that 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 10(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, so that 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 10(d) shows a code fragment included in (EG5). In this if-statement clone, there were 2 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 7 clone sets were not merged, since merging them would be highly dependent on the skill of an individual programmer.

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.

5.3 Result of the "Pull Up Method"

We obtained 18 clone sets 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 3 groups, (PG1) to (PG3), which are described in section 3.4. In this experiment, no clone set was classified into (PG1).

Ten clone sets were classified as (PG2). Figure 11(a) shows a code fragment included in (PG2).

[5] http://www.junit.org/

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

(a) Example of the "Extract Method" in (EG1)

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

(b) Example of the "Extract Method" in (EG2)

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

(c) Example of the "Extract Method" in (EG4)

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

(d) Example of the "Extract Method" in (EG5)

Figure 10: Examples of the "Extract Method"

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 2 parameters.

Eight clone sets were classified as (PG3). Figure 11(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

```
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) Example of the "Pull Up Method" in (PG2)

```
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) Example of the "Pull Up Method" in (PG3)

Figure 11: Examples of the "Pull Up Method"

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 Related Work

This chapter proposed a method for code clone refactoring. This section introduces other techniques that support code clone refactoring. However, refactoring is not a silver bullet for code cloning. Thus, we also introduce several techniques that manage code clones without refactoring.

6.1 Techniques for Code Clone Refactoring

Fowler, a pioneer in the field of refactoring, mentioned that the "number one in the stink parade is duplicated code" (Fowler, 1999). He also presented some sets of operations for merging code clones. The operations described in Figures 1, 2, 3, and 4 are from his book. This book helped 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 forms of merged code fragments (Baxter et al., 1998). The forms help users understand what operations are required to merge code clones. However, the tool does not care about the positional relationship between code clones in the class hierarchy. Users have to investigate where the merged code fragment can be placed by themselves, whereas the proposed method gives 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.

Bakazinska et al. proposed a refactoring method for the duplicated methods (Balazinska et al., 2000). Their method provides the differences between code clones, which helps users to determine whether code clones can be merged or not. Also, their method measures the coupling between a duplicated method and its surrounding code. In their method, code clones are removed by using two design patterns "Strategy" and "Templete Method", which are not supported by the proposed method. Consequently, the proposed method and their method are complemental approaches.

Cottrell et al. implemented a tool that visualizes the detailed correspondences between a pair of classes (Cottrell et al., 2007). 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 2 classes that should be merged. This tool can be combined with ARIES into a single refactoring process, since the 2 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 (Jarzabek, 2007). 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. Several research efforts reported that XVCL plays an important role in merging code clones (Jarzabek & Shubiao, 2003; Jarzabek & Li, 2006).

Komondoor et al. proposed an algorithm for procedure extraction (Komondoor & Horwitz, 2000). The inputs to the algorithm are (1) the CFG (control-flow graph) of a procedure and (2) a set of nodes in the CFG. 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.

The implementation of this algorithm adopts 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 with preserving semantics.

6.2 Techniques for Managing Code Clone

Kim et al. performed experiments on the repositories of open source software systems to investigate how code clones appear and disappear (Kim et al., 2005). The experimental results revealed the following points, which partly motivated the proposed simultaneous modification support method (Higo et al., 2007b).

- 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 (Kapser & Godfrey, 2006). 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 (Balazinska et al., 1999). 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 (Mayrand et al., 1996) and CP-MINER (Li et al., 2006).

Toomim et al. have proposed a simultaneous modification method on code clones included in the same clone set (Toomim et al., 2004). 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 et al. have also proposed a simultaneous editing method (Duala-Ekoko & Robillard, 2007). The method identifies corresponding lines in code fragments that are similar to each other based on the Levenshtein distance[6] 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.

[6]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.

Mann suggested that it should be effective to track 'copy and paste' actions (Mann, 2006), 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 (Kim et al., 2004). As well, Balint et al. reported that inconsistencies often occur between the 'copied-and-pasted' code (Balint et al., 2006). 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.

7 Conclusion

In this chapter, a new refactoring method for code clones was provided 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 provided in this chapter adequately characterized the code clones in ANT, and most of the code clones recommended by ARIES could be refactored with simple operations.

Acknowledgments

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.

References

Balazinska, M., Merlo, E., Dagenais, M., Lagüe, B., & Kontogiannis, K. (1999). Measuring Clone Based Reengineering Opportunities. In Proc. of the 6th IEEE International Symposium on Software Metrics (pp. 292–303).

Balazinska, M., Merlo, E., Dagenais, M., Lagüe, B., & Kontogiannis, K. (2000). Advanced Clone-Analysis to Support Object-Oriented System Refactoring. In Proc. of the 7th IEEE International Working Conference on Reverse Engineering (pp. 98–107).

Balint, M., Girba, T., & Marinescu, R. (2006). How Developers Copy. In Proc. of the 14th IEEE International Conference on Program Conprehension (pp. 56–68).

Baxter, I., Yahin, A., Moura, L., Anna, M., & Bier, L. (1998). Clone Detection Using Abstract Syntax Trees. In Proc. of International Conference on Software Maintenance 98 (pp. 368–377).

Burd, E. & Bailey, J. (2002). Evaluating Clone Detection Tools for Use during Preventative Maintenance. In Proc. of the 2nd IEEE International Workshop on Source Code Analysis and Manipulation (pp. 36–43).

Cottrell, R., Chang, J. J., Waler, R. J., & Denzinger, J. (2007). Determing Detailed Structural Correspondence for Generalization Tasks. In Procs of the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundation of Software Engineering (pp. 165–174).

Duala-Ekoko, E. & Robillard, M. P. (2007). Tracking Code Clones in Evolving Software. In Proc. of the 29th International Conference on Software Engineering (pp. 158–167).

Fowler, M. (1999). Refactoring: improving the design of existing code. Addison Wesley.

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

Higo, Y., Ueda, Y., Kusumoto, S., & Inoue, K. (2007b). Simultaneous Modification Support Based on Code Clone Analysis. In Proc. of the 14th Asia-Pacific Software Engineering Conference (pp. 262–269).

Jarzabek, S. (2007). Effective Software Maintenance and Evolution: Reused-based Approach. CRC Press Taylor and Francis.

Jarzabek, S. & Li, S. (2006). Unifying Clones with a Generative Programming Technique: a Case Study. Journal of Software Maintenance and Evolution: Research and Practice, 18(4), 267–292.

Jarzabek, S. & Shubiao, L. (2003). Eliminating Redundancies with a "Composition with Adaptation" Meta-programming Technique. In Proc. of ESECFSE'03 European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering (pp. 237–246).

Juergens, E., Deissenboeck, F., Hummel, B., & Wagner, S. (2009). Do code clones matter? In Proc. of the 30th International Conference on Software Engineering.

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

Kapser, C. & Godfrey, M. W. (2006). "Cloning Considered Harmful" Considered Harmful. In Proc. of the 13th Working Conference on Reverse Engineering (pp. 19–28).

Kim, M., Bergman, L., Lau, T., & Notkin, D. (2004). An Ethnographic Study of Copy and Paste Programming Practices in OOPL. In Proc. of 2004 International Symposium on Empirical Software Engineering (pp. 83–92).

Kim, M., Sazawal, V., Notkin, D., & Murphy, G. C. (2005). An Empirical Study of Code Clone Genealogies. In Proc. of the ACM SIGSOFT Symposium on the Foundations of Software Engineering (pp. 187–196).

Komondoor, R. & Horwitz, S. (2000). Semantics-preserving procedure extraction. In Proc. of the 27th ACM SIGPLAN-SIGACT on Principles of Programming Languages (pp. 155–169).

Li, Z., Lu, S., Myagmar, S., & Zhou, Y. (2006). CP-Miner: Finding Copy-Paste and Related Bugs in Large-Scale Software Code. IEEE Transactions on Software Engineering, 32(3), 176–192.

Mann, Z. A. (2006). Three Public Enemies: Cut, Copy, and Paste. IEEE Computer, 39(7), 31–35.

Mayrand, J., Leblanc, C., & Merlo, E. (1996). Experiment on the Automatic Detection of Function Clones in a Software System Using Metrics. In Proc. of the International Conference on Software Maintenance 96 (pp. 244–253).

Mens, T. & Tourwe, T. (2004). A survey of software refactoring. IEEE Transactions on Software Engineering, 30(2), 126–139.

Monden, A., Nakae, D., Kamiya, T., Sato, S., & Matsumoto, K. (2002). Software Quality Analysis by Code Clones in Industrial Legacy Software. In Proc. of the 8th IEEE International Software Metrics Symposium (pp. 87–94).

Roy, C. K., Cordy, J. R., & Koschke, R. (2009). Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. Science of Computer Programming, 74(7), 470–495.

Rysselberghe, F. & Demeyer, S. (2004). Evaluating Clone Detection Techniques from a Refactoring Perspective. In Proc. of the 19th IEEE International Conference on Automated Software Engineering (pp. 336–339).

Toomim, M., Begel, A., & Graham, S. (2004). Managing Duplicated Code with Linked Editing. In Proc. of IEEE Symposium on Visual Languages and Human-Centric Computing (pp. 173–180).