Proceedings of the Eighth IASTED International Conference
SOFTWARE ENGINEERING AND APPLICATIONS
November 9-11, 2004, Cambridge, MA, USA

ARIES: REFACTORING SUPPORT ENVIRONMENT BASED ON CODE CLONE ANALYSIS

Yoshiki Higo[1] Toshihiro Kamiya[2] Shinji Kusumoto[1] Katsuro Inoue[1]
[1]Graduate School of Information Science and Technology, Osaka University
[2]PRESTO, Japan Science and Technology Agency
email : {y-higo,kamiya,kusumoto,inoue}@ist.osaka-u.ac.jp

ABSTRACT
Code clone has been regarded as one of factors that make software maintenance more difficult. A code clone is a code fragment in a source code that is identical or similar to another. For example, if we modify a code fragment which has code clones, it is necessary to consider whether we have to modify each of its code clones. Hence, removal of code clones makes maintainability and comprehensibility of source code more improved. We have proposed a method that detects refactoring-oriented code clone. In this paper, in order to improve the usefulness and applicability of the method in the actual software maintenance, we have extended our refactoring support method. Concretely, we have developed a characterization of code clones by some metrics, which suggest how to remove them. Then, we have developed refactoring support tool **Aries**. We expect Aries can support software maintenance more effectively.

KEY WORDS
Code Clone, Refactoring, Metrics, Object-Oriented, Tool, Software Maintenance

1 Introduction

Recently, maintaining software systems has been becoming more difficult as the size and complexity of software is increasing. Maintenance of software system is defined as 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[13]. Actually, it is reported that many software companies expend a lot of time and human cost for software maintenance.

It is generally said that code clone is one of factors that make software maintenance more difficult. E.g. Fowler said that number one in the stink parade is duplicated code in his book[6]. A code clone is a code fragment that is identical or similar to another. Code clones are introduced because of various reasons such as reusing code by 'copy-and-paste'. If we modify a code fragment and it has many code clones, it is necessary to consider whether we have to modify each of its code clones. Especially, for large scale software, such processes are very complicated and need much cost. So, efficient code clone detection is necessary and important in software development and maintenance.

So far, there exist many researches to automatically detect code clones[4][11][12] and remove code clones[2][3][10]. We have also suggested a refactoring method that can apply practical software development and maintenance[7]. But, we have not implemented the method as an actual software tool.

In this paper, in order to improve the usefulness and applicability of the method in the actual software maintenance, we have extended our refactoring support method. Concretely, we have developed a characterization of code clones by some metrics, which suggests how to remove them. Then, we have developed refactoring support tool **Aries**. Aries can detect refactoring-oriented code clones in practical time from large scale software. Moreover, Aries characterizes detected code clone using some metrics. In other word, Aries tells the user which code clones can be removed and how to remove them. So, the user can concentrate on modifying source code, which leads software development and maintenance to more effective ones. Through case studies for an open source software, we confirm the applicability of Aries.

2 Preliminaries

Here, we define some terminology regarding code clones. Next, we briefly explain our previous research results, a code clone detection tool **CCFinder**[9], and a refactoring method for code clones detected by CCFinder.

2.1 Code Clone

A clone relation is defined as an equivalence relation (i.e., reflexive, transitive, and symmetric relation) on code fragments[9]. A clone relation holds between two code fragments if (and only if) they are the same sequences. (Sequences are sometimes original character strings, strings without white spaces, sequences of token type, and transformed token sequences.) For a given clone relation, a pair of code fragments is called a **clone pair** if the clone relation holds between the fragments. An equivalence class of clone relation is called a **clone set**. That is, a clone set is a maximal set of code fragments in which a clone relation holds between any pair of code fragments. A code fragment in a clone set of a program is called a code clone or

simply a clone.

2.2 Detecting Code Clone

CCFinder detects code clones both within files and across files from programs and outputs the locations of the clone pairs on the programs. The length of minimum code clone is set by the user in advance. Clone detection of CCFinder is a process in which the input is source files and the output is clone pairs. The process consists of following four steps:

Step1 Lexical analysis: Each line of source files is divided into tokens corresponding to a lexical rule of the programming language. The tokens of all source files are concatenated into a single token sequence, so that finding clones in multiple files is performed in the same way as single file analysis.

Step2 Transformation: The token sequence is transformed, i.e., tokens are added, removed, or changed based on the transformation rules that aims at regularization of identifiers and identification of structures. Then, each identifier related to types, variables, and constants is replaced with a special token. This replacement makes code fragments with different variable names clone pairs.

Step3 Match Detection: From all the sub-strings on the transformed token sequence, equivalent pairs are detected as clone pairs.

Step4 Formatting: Each location of clone pair is converted into line numbers on the original source files.

CCFinder adopts suffix-tree algorithm, which is enable to analyze the system of milliions line scale in practical use time. The detail is written in [9].

2.3 Identifying Refactoring-Oriented Code Clone

The removal of code clones is generally referred as **refactoring**[6] or restructuring. The key idea of our method is to find a kind of cohesive code fragment (like *compound block* or *method bodies*) from the code clone fragments. Figure 1 shows an example. In this figure, there are two code fragments *A* and *B* from a program, and the code fragments with hatching are maximal clones between them. In code fragment *A*, some data are assigned to list data structure from the head successively. In code fragment *B*, they are done so from the tail successively. The `for` blocks in *A* and *B* have a common logic that handles a list data structure. There are, however, sentences before and after `for` block, that are not necessarily related with the `for` block from semantic point of view. Such semantically unrelated sentences often obstruct refactoring. In other word, extracting only `for` block as a code clone is more preferable from refactoring viewpoint in this example.

This method is implemented as a filter for the output of CCFinder. We named the filter **CCShaper**[7]. The extracting process using CCShaper consists of the following three steps:

Step1 Detect clone pairs using CCFinder.

Step2 Provide syntax information (body of method, loop and so on) to each block by parsing the source files where clone pair are detected in Step1 and investigating the positions of blocks.

Step3 Extract structural blocks in the code clone using the information of location of clone pairs and structural blocks. Intuitively, structural block indicates the part of code clone that is easy to move and merge.

CCShaper performs Steps 2 and 3. For example, CCShaper extracts the following types of code clone as refactoring-oriented code clones for Java language.

Declaration class { }, interface { }

Method method body, constructor, static initializer

Statement if, for, while, do, switch, try, synchronized

3 Advising Refactoring Pattern for Each Code Clone

In the previous method described in Section 2.3, we have only proposed the approach to extract the refactoring-oriented clones and did not consider how to remove them. So, the user has to decide how to remove the code clones by him/herself.

This paper describes the answer for this problem. We have introduced some metrics to determine how to remove them. Detected clones are quantitatively characterized by using the metrics which support the user how to remove them.

3.1 Refactoring for Code Clone Removal

We use existing refactoring pattern[6], especially “Extract Method” and “Pull Up Method”, to remove code clones. “Extract Method” means that a fragment of source code are extracted and redefined as a new method[6]. Originally, this pattern is applied to too long method or too complex part. Here, in order to remove code clones, we use “Extract Method” to extract code clone fragments as a common new method. “Pull Up Method” means that the same methods defined in child classes are pulled up to its parent class[6]. This pattern is performed because of various reasons such as design pattern. If two or more child classes which have a common parent class include a clone method, pulling up such methods means clone removal.

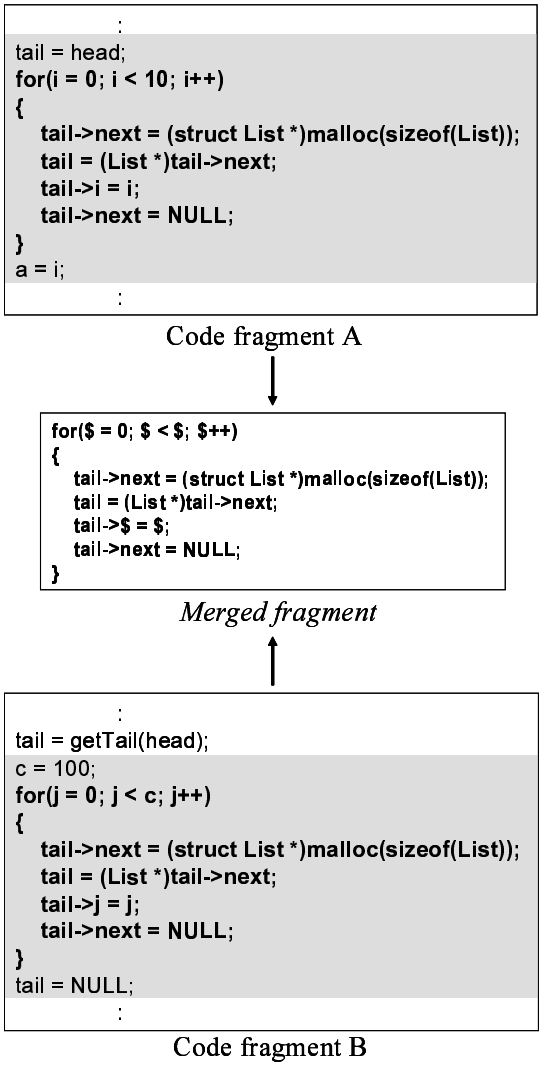


Figure 1. Example of merging two code fragments

3.2 Code Clone Metrics for Determining Refactoring Pattern

We attempt to refine detected code clones by measuring their characteristics to remove some of them. "Extract Method" is the extraction of a code fragment, so it is desirable that the target fragment has low coupling with the other surrounding fragments in the method, in other words, the variables defined outside the fragment aren't used (referred and assigned) in the fragment. If such variables are used, it is necessary to provide them as parameters for the new method. Therefore, we measure the amount of such variables.

On the other hand, "Pull Up Method" means moving identical methods in child classes to the parent class, so it is necessary that the child classes have common parent class. Therefore, we measure the dispersion of clones in the class hierarchy. The above characterizing makes it possible to determine how each clone can be removed. In order to make the decision, we introduce three metrics.

For the variables which are defined outside the code clone fragment, we define two metrics $NRV(S)$(*the Number of Referred Variables*), and $NAV(S)$(*the Number of Assigned Variables*). Here, we assume that clone set S includes code fragments $f_1, f_2, \cdots, f_n$. Code fragment f_i refers externally defined variables $rv_{i_1}, rv_{i_2}, \cdots, rv_{i_{s_i}}$, and assigns some values to variables $sv_{i_1}, sv_{i_2}, \cdots, sv_{i_{t_i}}$.

$$NRV(S) = \frac{1}{n}\sum_{i=1}^{n} s_i, \quad NAV(S) = \frac{1}{n}\sum_{i=1}^{n} t_i,$$

Intuitively, $NRV(S)$ represents the average of the number of externally defined variables referred in the fragments of the clone set S. Additionally, $NAV(S)$ represents the average of the number of assigned ones.

For the dispersion in class hierarchy, we defined a metric $DCH(S)$(*the Dispersion of Class Hierarchy*). As described above, the clone set S includes code fragments $f_1, f_2, \cdots, f_n$. C_i denotes the class which includes code fragment f_i.

Then, if the classes $C_1, C_2, \cdots, C_n$ have several common parent classes, C_p is defined as the class which lays the lowest position in class hierarchy among the parent classes $C_1, C_2, \cdots, C_n$. Also, $D(C_k, C_h)$ represents the distance between class C_k and class C_h in the class hierarchy.

$$DCH(S) = max\ \{D(C_1, C_p), \cdots, D(C_n, C_p)\}$$

The value of $DCH(S)$ also becomes larger as the degree of the dispersion of its clone set S becomes large. If all fragments of a clone set S are in the same class, the value of its $DCH(S)$ is set as 0. If all fragment of a clone set are in a class and its direct child classes, the value of its $DCH(S)$ is set as 1. Exceptionally, if classes which have some fragments of a clone set S don't have common parent class, the value of its $DCH(S)$ is set as -1. In detail, this metric is measured for only the class hierarchy where the target software exists because it is unrealistic that the user pulls up some methods which are defined in the target software classes to library classes like JDK.

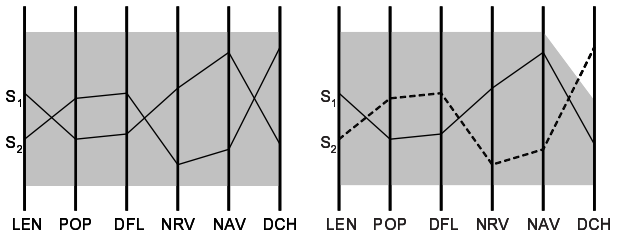
These three metrics can be used for not only "Extract Method" and "Pull Up Method", but also "Template Method" and "Parameterize Metghod" and so on.

4 Refactoring Support Tool: Aries

4.1 Overview

Based on the proposed method, we have implemented a refactoring support tool named **Aries** with Java language. For detection of code clones, Aries internally calls CCShaper[7]. Figures 2(a) and 2(b) show snapshots of Aries with the name of the windows.

Intuitively, the user specifies the distinctive clone set on the *Main Window*. Then, he/she analyzes the details of



(a) Before selection (b) After selection

Figure 3. Metric Graph

it on the *Clone Set Viewer*.

4.2 Functions

The user mainly uses the *Metric Graph View* to identify, filter, and select clone sets.

4.2.1 Metric Graph View

The *Metric Graph View* uses existing metrics, $LEN(S)$, $POP(S)$, and $DFL(S)$ [14] in addition to three metrics defined in Section 3.2. The existing metrics are defined as follows :

LEN(S) for clone set S is the maximum length of token sequence for each one in S.

POP(S) is the number of elements (code fragments) of a given clone set S. A clone set with a high value of $POP(S)$ means that similar code fragment appear in many places.

DFL(S) indicates an estimation of how many tokens would be removed from source files when the code fragments in a clone set S are reconstructed. This reconstruction is considered as the simplest case that all code fragments of S are replaced with caller statements of a new identical routine (function, method, template function, or so). After the reconstruction, $LEN(S) \times POP(S)$ tokens are occupied in the source files. In the newly reconstructed source files, they occupy $k \times POP(S)$ tokens (let k be the number of tokens for one caller statement) for caller statements and $LEN(S)$ tokens for callee routine.

Here, we explain the *Metric Gragh View* using an example shown in Figure 3. In the *Metric Graph View*, each metric has a parallel coordinate axe. Upper and lower limits are set per each metric. The hatching part is between upper and lower limits of each metric. A polygonal line is drawn per each clone set. In this example, values for the clone sets S_1 and S_2 are drawn. In the left graph(3(a)), all metric values of S_1 and S_2 are between upper and lower limits. So, these two clone sets are selected state. In the right graph(3(b)), the value of $DCH(S_2)$ is bigger than the upper limit of DCH, which means S_2 is unselected state. The *Metric Graph View* enables the user to select arbitrary clone set by changing upper and lower limits of each metric. And, the result of selection is reflected on the *Clone Set List*.

NRV/NAV Selector In the *NRV/NAV Selector*, Figure 2(a), the user can decide which types of variables are counted as metrics $NRV(S)$ and $NAV(S)$. Currently, the variables are selected from the following six types, field members of its class and parent classes and interfaces, "this" variable, "super" variable, and local variables.

For example, if the user is going to perform "Extract Method" within a class, it is not necessary to count all types of variables except local ones because these variables can be accessed anywhere in the same class. On the other hand, if the user is going to perform refactoring that crosses over two or more classes like "Pull Up Method", these ones should be counted.

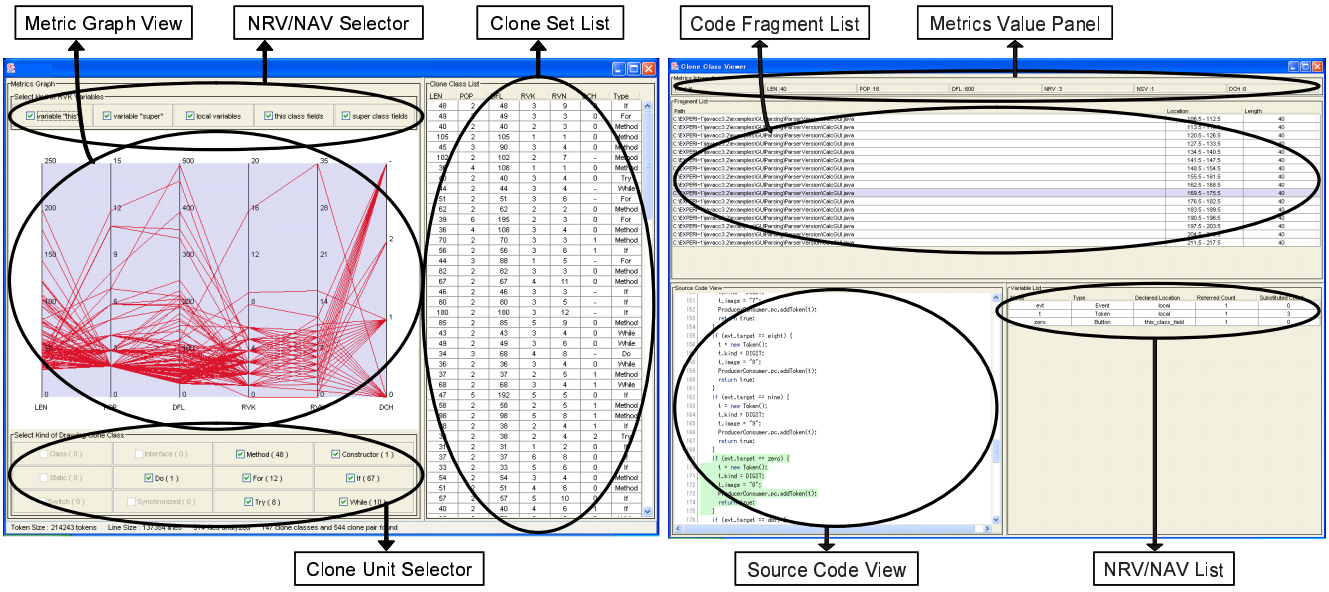
Clone Unit Selector In the *Clone Unit Selector*, the user can decide which types of clone unit are shown in the *Metric Graph View*. Currently, twelve types of clone units exist as described in Section 2.3. For example, if the user is going to perform "Pull Up Method", he/she should check only 'method' unit because the target of this pattern is the existing methods.

Clone Set List The *Clone Set List* shows all clone sets which are selected in the *Metric Graph View*. And the list can sort clone sets in ascending and descending sequence of each metric value. Double-clicking a clone set on this view is a trigger to run the *Clone Set Viewer* as shown in Figure 2(b). It shows more detail information of the selected clone set.

Metrics Value Panel The *Metrics Value Panel* shows the values of all metrics of clone set selected in the *Main Window*.

Code Fragment List The *Code Fragment List* shows the list of all code fragments included in the selected clone set. Each element of the list has three kinds of information, a path to each file which includes the code clone fragment, the location of the code clone in the file(the number of beginning line, beginning column, end line and end column), and the number of token included in the code clone fragment.

Source Code View The *Source Code View* works cooperatively with the *Code Fragment List*. The user can obtain the actual source code corresponding to the code clone fragment selected in the *Code Fragment List*. The fragment including the clones is emphatically displayed.



(a) Main Window

(b) Clone Set Viewer

Figure 2. Snapshots of Aries

NRV/NAV List The *NRV/NAV List* shows the list of all variables which are used and defined externally in the code fragment which is selected in the *Code Fragment List*. Each element of this list has three kinds of information, the name of its variable, the type of its variable and the count of used.

4.3 Refactoring Procedure

Now, we show example refactoring processes “Pull Up Method” and “Extract Method”.

If the user wants to perform “Pull Up Method”, the following conditions should be considered for example.

PC1 The target is ‘method’ unit code clone.

PC2 The value of $DCH(S)$ is more than 1.

Usually, ”Pull Up Method” is performed on existing methods, so (PC1) should be considered. And, the classes whose method includes target code clones have to inherit common parent class, so (PC2) should be considered. Next, the refinement process is as follows. At first, the user checks only ‘method’ unit checkbox on the *Clone Unit Selector*, which is reflected to the *Metric Graph View*. Next, the user sets the lower limit of $DCH(S)$ as more then 0. This operation is reflected to the *Clone Set List*. As the result, the list shows the clone sets which meet the conditions (PC1) and (PC2).

On the other hand, if the user wants to perform “Extract Method”, the following conditions should be considered for example.

EC1 The target is ‘statement’ unit code clones.

EC2 Tthe value of $DCH(S)$ is 0.

EC3 The value of $NAV(S)$ is less than 1.

Since “Extract Method” is usually performed on a code fragment in a method, (EC1) is considered. Next, if all fragments of clone set S exist in the same class, it is easy to merge them. So, (EC2) is considered. The reason to consider (EC3) is that if some variables which are externally defined are assigned in a fragment, it is necessary to make them parameters of the new extracted method, and to return them to its method caller place to reflect the values of them. It is necessary to contrive like making new data class if two or more values are assigned. The refinement process is as follows. At first, the user checks only ‘statement’ unit (do, if, for, switch, synchronized, try, while) checkbox on the *Clone Unit Selector*, which is reflected to the *Metric Graph View*. Next, the user checks only ‘local variable’ on the *NRV/NAV Selector* because other type variables can be accessed as far as in the same class. Next, the user sets the range of $DCH(S)$ as some value between 0 and 1($0 \leq DCH(S) \leq 1$), and the upper limit of $NAV(S)$ as less then 2. As the result of these operations, the *Clone Set List* shows only the clone sets which meet above three conditions (EC1), (EC2) and (EC3).

5 Case Study

5.1 Overview

In order to evaluate the usefulness of Aries, we have applied it to Ant 1.6.0[1]. It includes 627 files and the size is 180,000 LOC. In this case study, we set 30 tokens as the length of minimum code clone of CCShaper(intuitively, thirty tokens correspond to about five LOC). The value 30 is the empirical value which was derived from our previous studies. Then, we tried to perform "Extract Method" and "Pull Up Method" to code clones detected by Aries. It took 2 minutes to detect code clones, and we got 154 clone sets from Ant. The followings are the number of clones.

All detected clones	154
"Extract Method"	59
"Pull Up Method"	20

The conditions of "Extract Method" and "Pull Up Method" are the same as ones described in Section 4.3. In Sections 5.2 and 5.3, we describe the details of refactoring using Aries. Also, after removing several clone sets, we performed regression tests to confirm the behavior of Ant. In the regression test, we used totally 220 test cases included in Ant package. These test cases used JUnit[8], which is one of regression testing frameworks. So, we could easily perform all test cases and took about 4 minutes to perform all test cases.

5.2 "Extract Method"

As described above, we got 59 clone sets as the result using the "Extract Method" conditions described in Section 4.3. Then, we browsed and examined all source codes of each clone set, and classified them to the following four groups:

Group 1 Clone sets that can be removed only by extracting them and making a new method in the same class.

Group 2 Clone sets that can be removed by extracting them and making a new method with setting the externally defined variables as parameters of it because such variables are used in the clone.

Group 3 Clone sets that can be removed by extracting them and making a new method with setting the externally defined variables as parameters of it and with adding parameters of return statement to deliver the results to the variables used in the caller.

Group 4 Clone sets that can be removed but need a lot of effort.

Three clone sets were classified to Group 1. Figure 4 shows a source code of one of them. In this 'if-statement' clone, no externally defined variable was used. So, it was very easy to extract it as a new method in the same class.

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

Figure 4. Example of Extract Method in Group 1

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

Figure 5. Example of Extract Method in Group 2

thirty-four clone sets were classified to Group 2. Figure 5 shows a source code of one of them. In this 'if-statement' clone, the variable "javacopts" was a field member of its class, and the variable "genicTask" was a local variable. So, it was necessary to set "genicTask" as a parameter of a new method to extract this code clone in the same class.

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

Figure 6. Example of Extract Method in Group 3

Fifteen clone sets were classified to Group 3. Figure 6 shows a source code of one of them. In this 'if-statement' clone, the variable "iSaveMenuItem" was externally defined. Moreover, the value was assigned to it. So, it was necessary to set "iSaveMenuItem" as a parameter of a new method and add 'return statement' to reflect the result of assignment to the caller.

Seven clone sets were classified to Group 4. Figure 7 shows a source code of one of them. In this 'if-statement' clone, some 'return-statements' were used. So, a lot of effort would be necessary to extract it. In this case study, we didn't remove these four clone sets because we thought that removal of them was strongly dependent on the skill of each programmer.

5.3 "Pull Up Method"

Next, we describe the results of applying 'Pull Up Method". As described above, we got 20 clone sets as the result using the "Pull Up Method" conditions described in Section 4.3. Then, we browsed and examined all source codes of each code clone, and classified them to the following four groups:

Group 5 Clone sets that can be removed only by moving

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

Figure 7. Example of Extract Method in group 4

them to the common parent class.

Group 6 Clone sets that can be removed by moving them to common parent class after adding variables which are defined outside.

Group 7 Clone sets that can be removed by moving them to common parent class and adding a new method which needs parameters of outside variables and return statement. Existing method which includes the pull-uped clones can be deleted or changed so that they call the new method from the inside. If they are deleted, it is necessary to change all its caller places.

Group 8 Clone sets that need much contrivance to remove.

Here, no clone set was classified to Group 5.

```
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());
    }
}
```

Figure 8. Example of Pull Up Method in group 6

Ten clone sets were classified to Group 6. Figure 8 shows a source code of one of them. In this 'method' clone, the variable "this" was omitted at calling method "getCommentFile" which was defined in the same class. The variables "this" and "FLAG_COMMENTFILE", which are field members of the same class, are externally defined. To adapt "Pull Up Method" pattern, with adding two parameters, we pulled up them to the common parent class.

Two clone sets were classified to Group 7. Figure 9 shows a source code of one of them. In this method clone, the variable "map" was externally defined, and some values were assigned to it. Here, the methods named "setError" was defined in the common parent class. So, to pull up this clone set to the common parent class, it was necessary to add a parameter and return statement for the variable "map".

Eight clone sets were classified to Group 8. Figure 10 shows a source code of one of them. In this method,

```
public void verifySettings() {
    if (targetdir == null) {
        setError("The targetdir attribute is required.");
    }
    if (mapperElement == null) {
        map = new IdentityMapper();
    } else {
        map = mapperElement.getImplementation();
    }
    if (map == null) {
        setError("Could not set <mapper> element.");
    }
}
```

Figure 9. Example of Pull Up Method in group 7

```
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());
    }
}
```

Figure 10. Example of Pull Up Method in group 8

the method "checkOptions" was called. This method was defined in the same class. Here, the methods named "getProject", "getViewPath" and "getLocation" were defined by using common parent class. And, the variable "commandLine", which was a parameter of this method, was defined and used in the clone. So, this method caller made it difficult to apply "Pull Up Method" to this clone set. But, the method "checkOptions" was defined in each child class. We can apply "Template Method" pattern[6] on them. At first, we move the clone to the common parent class. Next, we define an abstract method "checkOptions" in the common parent class.

6 Related Works

There are several related studies about refactoring of code clones. Komondoor et al.[10] has proposed a refactoring method using program slicing. In this method, a program dependence graph is constructed by analyzing target source codes. Identical or similar parts are detected as code clone. This detection is greatly precise because of considering control and data flow of program. Moreover, it can detect reordered and intertwined clones[10] which cannot de-

tected by CCFinder. But, time complexity of constructing program dependence graph is $O(m^2)$(m is the number of statement and expression included in target source codes), which makes it difficult to apply this method to large scale software.

Also, Balazinska et al.[2] have proposed an approach to extract code clones using several metrics. Considering the context of code clone, it seems to be practical. On the other hand, since the unit of the code clone is restricted to 'function' and 'method', it makes difficult to perform refactoring for smaller unit.

JDT on Eclipse[5] is well-known tool for supporting refactoring activity. JDT semi-automatically modifies source code according to the user's operation that specifies where should be refactored and how to modify them. All influenced parts by its refactoring are also modified automatically. If a method name is changed, all caller parts of the method are modified. The user doesn't need to perform troublesome modification manually. We consider that Aries with such tools can totally support refactoring activity. Aries can specify where should be refactored and how to modify them. He/She refactors, with JDT, the specified part using the pattern indicated by Aries.

7 Conclusion

In this paper, we have proposed a new refactoring method for code clones, and implemented a refactoring support tool, Aries. The code clone analysis algorithm used in Aries is so fast that it can apply industrial large-size software system. Also, we have applied Aries to Ant. Proposed metrics well characterized code clones, and most of refined them could be removed.

As future works, we are going to perform more detail analysis for code clones. For example, we are going to consider the effectiveness of refactoring. Currently, we refine code clones based on the judgment whether they can be removed or not. If we can judge whether the code clones should be removed or not, the supporting of the refactoring will become more effective.

Acknowledgment

This work is partly supported by Japan Science and Technology Corporation, Research and Development for Applying Advanced Computational Science and Technology, Grant-in-Aid for Young Scientists (B)(No:15700031) of Japan Society for the Promotion of Science, and the Comprehensive Development of e-Society Foundation Software program of the Ministry of Education, Culture, Sports, Science and Technology.

References

[1] Ant, `http://ant.apache.org`, 2003.

[2] M. Balazinska, E. Merlo, M. Dagenais, B. Lague and K. Kontogiannis, "Advanced clone-analysis to support object-oriented system refactoring", *Proc. the 7th Working Conference on Reverse Engineering*, pp98-107, Brisbane, Australia, Nov. 2000.

[3] I.D. Baxter, A. Yahin, L. Moura, M.S. Anna, and L. Bier, *Clone Detection Using Abstract Syntax Trees*, Proc. International Conference on Software Maintenance 98, pp368-377, Bethesda, Maryland, Mar. 1998.

[4] S. Ducasse, M. Rieger, and S. Demeyer, *A Language Independent Approach for Detecting Duplicated Code*, Proc. International Conference on Software Maintenance 99, pp109-118, Oxford, England, Aug. 1999.

[5] Eclipse, `http://www.eclipse.org`, 2004.

[6] M. Fowler, *Refactoring: improving the design of existing code*, Addison Wesley, 1999.

[7] Y. Higo, Y. Ueda, T. Kamiya, S. Kusumoto and K. Inoue, *On software maintenance process improvement based on code clone analysis*, Proc. 4th International Conference on Product Focused Software Process Improvement, pp.185-197, Rovaniemi, Finland, Dec. 2002.

[8] JUnit, `http://www.junit.org`, 2003.

[9] T. Kamiya, S. Kusumoto, and K. Inoue, *CCFinder: A multi-linguistic token-based code clone detection system for large scale source code* IEEE Transactions on Software Engineering, vol.28, no.7, pp.654-670, Jul. 2002.

[10] R. Komondoor and S. Horwitz, *Using slicing to identify duplication in source code*, Proc. the 8th International Symposium on Static Analysis, pp.40-56, Paris, France, Jul. 2001.

[11] J. Krinke, *Identifying Similar Code with Program Dependence Graphs* , In Proc. of the 8th Working Conference on Reverse Engineering, pp. 301-309, Suttgart, Germany, Oct. 2001.

[12] J. Mayland, C. Leblanc, and E.M. Merlo *Experiment on the automatic detection of function clones in a software system using metrics*, Proc. International Conference on Software Maintenance 96, pp.244-253, Monterey, California, Nov. 1996.

[13] T.M. Pigoski, *Practical Software Maintenance*, John Wiley and Sons, New York, Oct. 1996.

[14] Y. Ueda, T. Kamiya, S. Kusumoto, K. Inoue, *Gemini: Maintenance Support Environment Based on Code Clone Analysis*, 8th International Symposium on Software Metrics, pp67-76, Ottawa, Canada, June, 2002.