社団法人 電子情報通信学会
THE INSTITUTE OF ELECTRONICS,
INFORMATION AND COMMUNICATION ENGINEERS

信学技報
TECHNICAL REPORT OF IEICE.
SS2004-1 (2004-05)

コードクローン情報を用いたリファクタリング支援ツール

肥後 芳樹† 神谷 年洋†† 楠本 真二† 井上 克郎†

† 大阪大学 大学院情報科学研究科 コンピュータサイエンス専攻 〒560-8531 豊中市待兼山町 1-3
†† 科学技術振興機構 さきがけ 〒560-8531 豊中市待兼山町 1-3
E-mail: †{y-higo,kamiya,kusumoto,inoue}@ist.osaka-u.ac.jp

あらまし 近年，ソフトウェアの保守を悪化させている一要因として，コードクローンが議論されている．コードクローンとはソースコード中に存在する同一，または類似したコード片のことである．例えば，あるコード片にバグが含まれていた場合，そのコード片のコードクローン全てについて修正の是非を考慮する必要がある．コードクローンを対象とする保守支援としては，ソフトウェア内に存在するコードクローンを把握，管理する方法と，ソフトウェアからコードクローンを取り除く（リファクタリング）方法の２つがあげられる．前者については我々はこれまでに，コードクローン分析環境 Gemini を開発し，さまざまな事例に対して適用してきた．また後者に関しては，これまでにいくつかの手法が提案されているが，解析時間のコストが高いなどの理由により，実際に社会で用いられているソフトウェアに対しては適用が難しかった．本論文では，実用的な時間でソースコード中からリファクタリングに適したコードクローンの検出手法を提案する．また，抽出したコードクローンの特徴をメトリクスを用いて数値化する．これにより，そのクローンの集約方法が予測でき，ユーザは効率的なリファクタリング作業ができると期待される．また提案手法を実装したツールを作成し，適用実験を行なうことで，本手法の有用性を確認した．
キーワード コードクローン，リファクタリング，ソフトウェア保守，オブジェクト指向

Code Clone Analysis Method for Practical Refactoring Support

Yoshiki HIGO†, Toshihiro KAMIYA††, Shinji KUSUMOTO†, and Katsuro INOUE†

† Graduate School of Information and Science Technology, Osaka University
1-3, Machikaneyama-cho, Toyonaka, Osaka, 560-8531, Japan
†† PRESTO, Japan Science and Technology Agency
1-3, Machikaneyama-cho, Toyonaka, Osaka, 560-8531, Japan
E-mail: †{y-higo,kamiya,kusumoto,inoue}@ist.osaka-u.ac.jp

Abstract Recently, 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. There are two ways of maintenance support for code clones. One is to comprehend and manage code clones, and the other is to remove them. For the former support, we have developed code clone analysis environment **Gemini**. For the latter support, several methods have proposed. But, it is difficult to apply them to industrial software because of various reasons such as high time complexity. In this paper, we propose a method that detects refactoring-oriented code clone in practical use time. And, we develop a characterization of code clones by some metrics, which suggest how to remove them. Then, we develop refactoring support tool **Cancer**. We expect Cancer can support software maintenance more effectively.
Key words Code Clone, Refactoring, Software Maintenance, Object-Oriented

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 [12]. Actually, it is re-

ported 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 [6]. 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 pros and cons of modification in its corresponding all 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.

There are two ways of maintenance support for code clones. One is to comprehend and manage code clones, and the other is to remove them. For the former support, there exist many researches to automatically detect code clones [4] [11]. We have also developed code clone detection tool **CCFinder** [9] and code clone analysis environment **Gemini** [13]. We have been delivering Gemini (including CCFinder) to more than 50 software organizations and evaluated the usefulness of them in the actual software maintenance. For the latter support, several code clone removal methods have been proposed [2] [3] [10]. But, it is difficult to apply them to industrial software because of various reasons such as high time complexity.

For supporting refactoring activity in practical, we developed **Cancer** to support the refactoring for code clone. Cancer can detect refactoring-oriented code clones in practical time from large scale software. Moreover, Cancer characterizes detected code clones using some metrics. In other word, Cancer 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 several open source software, we confirm the applicability of Cancer.

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].

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 CCFinder

CCFinder [9] detects code clones 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

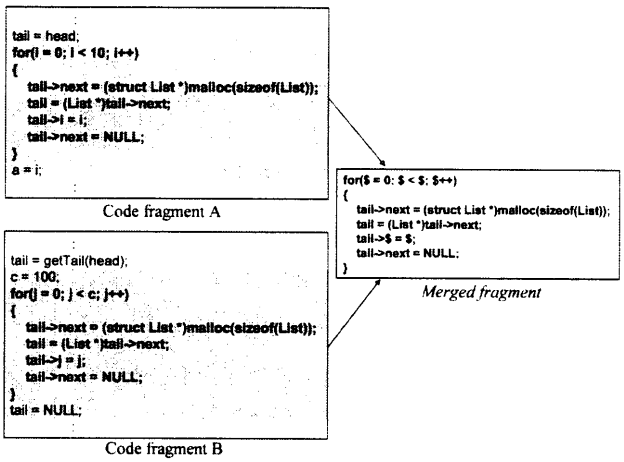


Fig 1 Example of merging two code fragments

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.

3. Proposed Method

3.1 Extraction of Refactoring-Oriented Code Clone

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 substituted 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.

We extract refactoring-oriented code clone from the output of CCFinder. For example, the following kinds of code clone are extracted as a refactoring oriented code clone for Java language.

Declaration : class { }, interface { }
Method : method body, constructor, static initializer
Statement : if, for, while, do, switch, try, synchronized

3.2 Code Clone Metrics for Determining Refactoring Pattern

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 plural child classes which have common parent class include clone method, pulling up such methods means clone removal.

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 substituted) 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 existing 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 several metrics.

For the variables which are defined outside the code clone fragment, we define two metrics $RVK(S)$, and $RVN(S)$. Here, we assume that clone set S includes code fragments $\{f_1, f_2, \cdots, f_n\}$. Code fragment f_i uses externally defined variables $\{v_{i_1}, v_{i_2}, \cdots, v_{i_{m_i}}\}$. Also, $RS(v_{i_j})$ denotes the total number of referred and substituted count of v_{i_j}.

$$RVK(f_i) = m_i,$$

$$RVN(f_i) = \sum_{i=1}^{m_i} RS(v_i)$$

and,

$$RVK(S) = (\sum_{i=1}^{n} RVK(f_i))/n,$$

$$RVN(S) = (\sum_{i=1}^{n} RVN(f_i))/n$$

Intuitively, $RVK(S)$ represents the number of externally defined variables used in the fragments of the clone set S. Additionally, $RVN(S)$ counts the number of usage of the variables used in the fragments of S. For the dispersion in class hierarchy, we defined a metrics $DCH(S)$. 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 of $\{C_1, C_2, \cdots, C_n\}$. Also, $D(C_k, C_h)$ represents the distance between class C_k and class C_h in the class hierarchy.

$$DCH(S) = max\ \{D(C_1, C_p), D(C_2, C_p), \cdots, D(C_n, C_p)\}$$

If the classes don't have common parent class,

$$DCH(S) = -1$$

The value of $DCH(S)$ also becomes larger as the degree of the dispersion of its clone set 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 children classes, the value of its $DCH(S)$ is set as 1. Exceptionally, if classes which have some fragments of a clone set 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.

4. Refactoring Support Tool: Cancer

Based on the proposed method, we have implemented a refactoring support tool **Cancer** with Java language. Figures 2(a) and 2(b) show snapshots of Cancer 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 it on the *Clone Set Viewer*.

4.1 Function of Each Component

Here, we explain some components on Cancer.

4.1.1 Metric Graph View

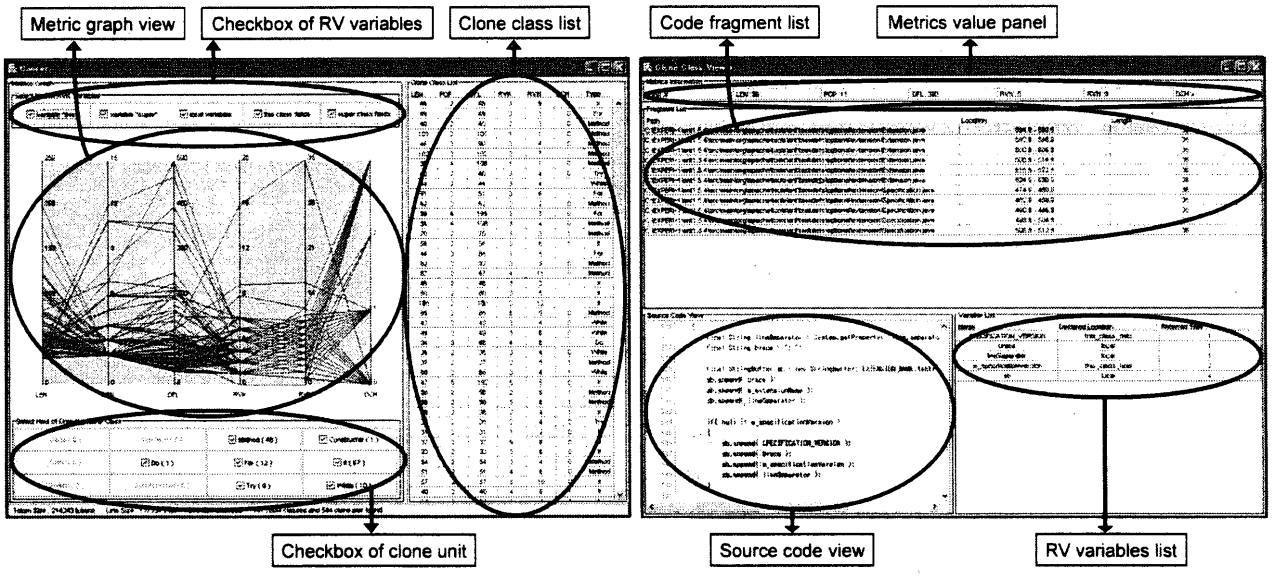
The *Metric Graph View* uses existing metrics, $LEN(S)$, $POP(S)$, and $DFL(S)$ [13] in addition to three metrics defined in Section 3.2. The followings are brief explanations of each metric.

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.

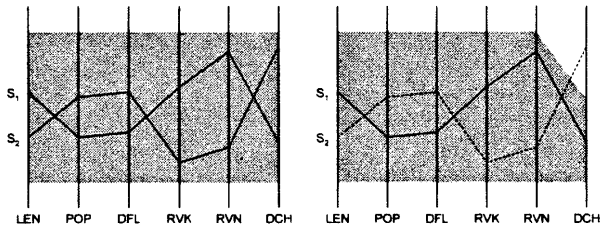
DFL(S) indicates an estimation of how many tokens would be removed from source files when the code fragments in a clone set S are reconstructed. This reconstruction is considered as the simplest case that all code fragments of S are replaced with caller statements of a new identical routine (function, 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



(a) Main Window (b) Clone Set Viewer

Fig 2 Snapshots of Cancer



(a) Before selection (b) After selection

Fig 3 Metric Graph

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*.

4. 1. 2 Checkbox of *RV* Variables

In the *Checkbox of RV Variables* in Figure 2(a), the user can decide which variables are counted as metrics $RVK(S)$ and $RVN(S)$. Currently, the variables are selected from the following five types.

- field members of its class,
- field members of parent class,
- "this" variable,
- "super" variable,
- local variables.

For example, if the user is going to perform "Extract Method" within a class, it is not necessary to count all kind 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 plural classes like "Pull Up Method", these ones should be counted.

4. 1. 3 Checkbox of Clone Unit

In the *Checkbox of Clone Unit*, the user can decide which kind of clone unit are shown in the *Metric Graph View*. Currently, the number of unit types are twelve as described in Section 3. 1. 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.

4. 1. 4 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.

4. 2 Refactoring Procedure

Now, we explain refactoring process using Cancer. 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 *Checkbox of Clone Unit*, 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 $RVK(S)$ is less than 1.

Since "Extract Method" is usually performed on a code fragment in a method, (EC1) is considered. Next, if all frag-

ments 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 used in a fragment, it is necessary to make them parameters of the new extracted method. Moreover, if some values are substituted to some of them, they have to be returned to method caller place to reflect the values of them. It is necessary to contrive like making new data class if plural value are substituted. The refinement process is as follows. At first, the user checks only 'statement' unit (do, if, for, switch, synchronized, try, while) checkbox on the *Checkbox of Clone Unit*, which is reflected to the *Metric Graph View*. Next, the user checks only 'local variable' on the *Checkbox of RV Variables* because other kind variables can be accessed as far as in the same class. Next, the user set the range of $DCH(S)$ as some value between 0 and 1($0 \leqq DCH(S) \leqq 1$), and the upper limit of $RVK(S)$ 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 Cancer, 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 thirty tokens as the length of minimum code clone of CCFinder(intuitively, thirty tokens correspond to about five LOC). The value thirty is the empirical value which was derived from our previous applications of CCFinder. We also set thirty tokens as the length of minimum clone of Cancer. Then, we tried to perform "Extract Method" and "Pull Up Method" to code clones detected by Cancer. We got 154 clone sets from Ant. The followings are the number of clones.

All detected clones	154
"Pull Up Method"	20
"Extract Method"	32

The conditions of "Pull Up Method"and "Extract Method" are the same as ones described in Section 4.2. In Section 5.3 and 5.2, we describe the details of refactoring using Cancer. 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 "Pull Up Method"

Next, we describe the results of applying 'Pull Up Method". As described above, we extracted 20 clone sets using the "Pull Up Method" conditions described in Section 4.2. Then, we browsed and examined all source codes of each code clone, and classified them to the following four groups:

Group 1 clone sets that can be removed only by moving them to the common parent class.

Group 2 clone sets that can be removed by moving them to common parent class after adding variables which are defined outside.

Group 3 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 methods 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 4 clone sets that need much contrivance to remove.

Here, no clone set was classified to Group 1.

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

Fig 4 Example of Pull Up Method in group 2

Ten clone sets were classified to Group 2. Figure 4 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.

```
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.");
    }
}
```

Fig 5 Example of Pull Up Method in group 3

Two clone sets were classified to Group 3. Figure 5 shows a source code of one of them. In this method clone, the variable "map" was externally defined, and some values were substituted to it(Method "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".

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

Fig 6 Example of Pull Up Method in group 4

Eight clone sets were classified to Group 4. Figure 6 shows a source code of one of them. In this method, the method "checkOptions" was called. This method was defined in the

same class(Methods, "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. In this case, "Template Method" pattern [6] could be applied. Procedure of this pattern appliance is the followings. At first, we moved the clone to the common parent class. Next, we defined an abstract method "checkOptions" in the common parent class.

5.3 "Extract Method"

As described above, we extracted 32 clone sets using the "Extract Method" conditions described in Section 4.2. 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.

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

Fig 7 Example of Extract Method in Group 1

Three clone sets were classified to Group 1. Figure 7 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 (javacopts != null && !javacopts.equals("")) {
    genicTask.createArg().setValue("-javacopts");
    genicTask.createArg().setLine(javacopts);
 }
```

Fig 8 Example of Extract Method in Group 2

Eighteen clone sets were classified to Group 2. Figure 8 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 is 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);
     }
}
```

Fig 9 Example of Extract Method in Group 3

Seven clone sets were classified to Group 3. Figure 9 shows a source code of one of them. In this 'if-statement' clone, the variable "iSaveMenuItem" was externally defined. Moreover, the value was substituted to it. So, it is necessary to set "iSaveMenuItem" as a parameter of a new method and add 'return statement' to reflect the result of substitution to the caller.

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

Fig 10 Example of Extract Method in group 4

Four clone sets were classified to Group 4. Figure 10 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 think that removal of them is strongly dependent on the skill of each programmer.

6. Conclusion

In this paper, we have proposed a new refactoring method of code clone, and implemented a refactoring support tool, Cancer. The code clone analysis algorithm used in Cancer is so fast that it can apply industrial huge scale software. Also, we have applied Cancer to Ant, and removed almost of refined clones.

As future works, we are going to perform more detail analysis for code clones. For example, distinction of reference and substitution of externally defined variables should be considered. Also, 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.

References

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

[2] M. Balazinska et al., "Advanced Clone-Analysis to Support Object-Oriented System Refactoring", *Proceedings the 7th Working Conference on Reverse Engineering*, 2000, 98-107.

[3] I. D. Baxter et al., *Clone Detection Using Abstract Syntax Trees*, Proc. of ICSM98, pages 368-377, Nov. 1998.

[4] S. Ducasse et al., *A Language Independent Approach for Detecting Duplicated Code*, Proc. of ICSM99, pages 109-118, 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. of Profes 2002, pp. 185-197 (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, (2002-7).

[10] R. Komondoor and S. Horwitz, *Using slicing to identify duplication in source code*, In Proc. of the 8th International Symposium on Static Analysis, Paris, France, July 16-18, 2001.

[11] J. Mayland, C. Leblanc, and E. M. Merlo *Experiment on the Automatic Detection of Function Clones in a Software System Using Metrics*, Proc. of IEEE Int'l Conf. on Software Maintenance (ICSM) '96, pages 244-253, Monterey, California, Nov. 1996.

[12] Pigoski T. M, *Maintenance*, Encyclopedia of Software Engineering, 1, John Wiley & Sons. 1994.

[13] Y. Ueda, T. Kamiya, S. Kusumoto, K. Inoue, *Gemini: Maintenance Support Environment Based on Code Clone Analysis*, 8th International Symposium on Software Metrics, June 4-7, 2002.