# On Refactoring Support Based on Code Clone Dependency Relation

Norihiro Yoshida[1], Yoshiki Higo[1], Toshihiro Kamiya[2], Shinji Kusumoto[1], Katsuro Inoue[1]
[1]Graduate School of Information Science and Technology, Osaka University
{n-yosida, y-higo, kusumoto, inoue}@ist.osaka-u.ac.jp
[2]National Institute of Advanced Industrial Science and Technology
t-kamiya@aist.go.jp

## Abstract

*Generally, code clones are regarded as one of the factors that make software maintenance more difficult. A code clone is a set of source code fragments identical or similar to each other. From the viewpoint of software maintainability, code clones should be removed. However, sometimes there are dependency relations among each of which belong to the different code clone, and it is advisable to refactor all of such code clones at once. In this paper, we focus on the case that such code fragment corresponds to a method body in Java programs. We defined "chained method" as a set of methods that have dependency relations. A set of "chained methods" whose elements are each other's code clone is called "chained clone", and an equivalence class of "chained clone" is called a "chained clone set". We propose a refactoring support method for "chained clone set" by providing an appropriate refactoring pattern to them. Finally, we present the "chained clone set" refactoring support tool that we have developed, together with some case studies to show the usefulness of the proposed method.*

## 1 Introduction

**Refactoring**[7] is a disciplined technique for restructuring an existing body of code, altering its internal structure without changing its external behavior. That is to say, refactoring is a technique to improve the maintainability of software. On the other hand, it needs some experience to identify the part of software where refactoring should be applied. Such parts are called **bad smells**[7]. Duplicated code, long methods, long parameter list are examples of the bad smells.

Code clones are fragments of the code which are exactly the same as or similar to each other. Source codes including code clones are more difficult to maintain than ones without code clones. Especially, for large scale software, it is very important to find code clones and there are a lot of research studies to find code clones automatically[1, 2, 3]

We have also been developing a code clone detection tool **CCFinder**[9]. CCFinder detects code clones from program source codes and outputs the locations of them in source codes. Since the code clones detected by CCFinder are not necessarily appropriate for refactoring, we have proposed a code clone refactoring support environment, called **Aries**[8], using code clone analysis results by CCFinder. Aries extracts structural blocks in code clones (e.g. class, method body, if-statement) from Java programs and provides the user how each code clone could be removed. We have applied Aries to several open source software and remove many of them without changing external behavior.

Through applications of Aries to several Java applications, we found that a code clone which is a set of method bodies in Java programs sometimes has dependency relations among the methods in it. Also, for practitioners, it is effective to conduct refactoring all of such code clones at once. In this paper, we intend to refactor such code clones.

Here, we define **chained method** as a set of methods that hold dependency relations. For given *chained methods*, if each set of the corresponding methods is a code clone, we call the set of *chained methods* **chained clone**. Then, we have proposed a refactoring method for them. According to the characteristics of them, we provide an appropriate refactoring pattern to them. Finally, we have developed a refactoring support tool for them and applied it to several open source Java programs. As the results, we found actual *chained clones* in those programs and refactored them (i.e., merged the code fragments and applied the regression tests).

Section 2 describes code clones. Section 3 explains the proposed approach to refactor the *chained clone* and the refactoring support tool. Section 4 evaluates the applicability of the tool. Finally, Section 5 concludes this paper and discusses the future works.

## 2 Code Clone

Here, we define some terminologies regarding code clones. Next, we briefly explain our previous research results, a code clone detection tool CCFinder, and a refactoring support environment Aries for code clones detected by CCFinder.

### 2.1 Definition of 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. The Clone detection of CCFinder is a process in which the input is source files and the output is a lot of clone pairs. The process consists of the 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 aim 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 a position (line numbers) on the original source files.

CCFinder adopts suffix-tree algorithm, which is enable to analyze the system of millions line scale in practical use time. For the details of CCFinder, please refer to [9].

### 2.3 Refactoring support environment: Aries

We have proposed a refactoring support method and developed a support environment Aries for Java programs. In Aries, CCFinder is used as a code clone detection engine. But, since code clones detected by CCFinder are sequence of tokens, they are not necessarily suitable for refactoring. To solve this problem, Aries extracts *structural parts* from code clones as refactoring-oriented ones after they are detected by CCFinder. The followings are the *structural parts* extracted from Java programs.

**Declaration** class { }, interface { }

**Method** method body, constructor, static initializer

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

Also, Aries characterizes refactoring-oriented code clones using several metrics. Here, we explain a metric **DCH(S)**(the Dispersion in the Class Hierarchy). $DCH(S)$ represents position and distance relationship between code fragments in a clone set $S$ in the class hierarchy. Suppose that $S$ includes code fragments $f_1,\ f_2,\ \cdots,\ f_n$ and $C_i$ denotes the class which includes code fragment $f_i$. Then, if the classes $C_1,\ C_2,\ \cdots,\ C_n$ have some common parent classes, $C_p$ is defined as the nearest common parent class (Java employs a single inheritance so that nearest parent class is decided unique). Also, $D(C_k,\ C_h)$ represents the distance between class $C_k$ and class $C_h$ in the class hierarchy, then $DCH(S)$ is defined as the following formula:

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

The value of $DCH(S)$ becomes larger as the degree of the dispersion of $S$ becomes large. If all fragments of $S$ are in the same class, the value of its $DCH(S)$ is set to 0. If all fragment of a clone set are in a class or its direct child classes, the value of its $DCH(S)$ is set to 1. Exceptionally, if classes which have some fragments of a clone set $S$ don't have any common parent class, the value of its $DCH(S)$ is set to $\infty$.

Using $DCH(S)$, the user can predict how each code clone can be removed. Considering a clone set $S_1$, which is a set of refactoring-oriented code clones. The followings are guidelines of refactoring for each case of $DCH(S_1)$ value.

**the value is 0** : Code clones in $S_1$ could be easily removed using "Extract Method" pattern in the same class, since this value means all code clones in $S_1$ are in the same class.

**the value is greater than 0** : Code clones in $S_1$ should be removed using "Pull Up Method" pattern. Since this value means all classes having code clones in $S_1$ extends some common parent classes. The user can remove code clones by pulling up them to the common parent class.

**the value is $\infty$** : The code fragments of $S_1$ are widely spread in the class hierarchy of the target software. To remove code clones in $S_1$, the practitioners have to move code clones to some other classes like a utility class using "Move Method" pattern.

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.

Based on the $DCH(S)$, we propose new metrics for *chained clones* in the next section.

# 3 Proposed method

## 3.1 Motivation

We have applied Aries to several open source and commercial Java programs and then got some feedback from the practitioners. First one is about granularity of code clones. A clone set which is a set of method bodies or class definitions is effective for refactoring from the viewpoint of software maintenance. On the other hand, refactoring small code fragments is not useful because of several regions.

Second one is about dependency relations among code clones. Sometimes there are dependency relations among methods each of which belongs to a different clone set, and it is desirable to refactor all of such clone sets at once.

Figure 1 shows an example of clone sets that have dependency relations. Suppose that we apply "Pull Up Method" pattern to the clone set $\alpha$. Aries provides the user with the guidance that $a_1$, $b_1$ and $c_1$ could be merged into a method and be pulled up to the parent class $S$. Similarly, Aries provides the practitioner with guidance that $a_2$, $b_2$ and $c_2$ could be merged into a method and be pulled up to $S$. These guidances are useful. But in this case, there could be more sophisticated refactoring method.

Here, we focus on the six methods ($a_1$, $b_1$, $c_1$, $a_2$, $b_2$, $c_2$), they might be divided into two sets $\{a_1, b_1, c_1\}$ and $\{a_2, b_2, c_2\}$ based on the dependency relations. Also, the two sets could be merged at once depending on the dependency relations.

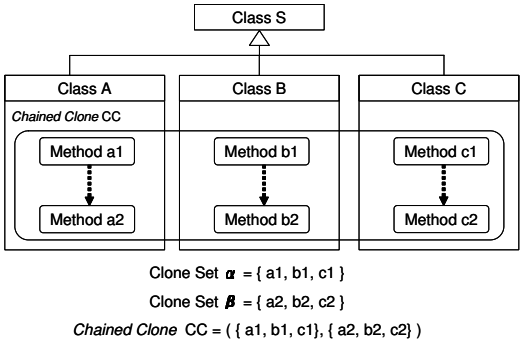



**Figure 1. Clone sets that have dependency relations**

In this paper, we define a set of code clones(clone pairs or clone sets) including dependency relations as *chained clone*(see Figure 1) and propose a refactoring support method based on patterns of the *chained clone*.

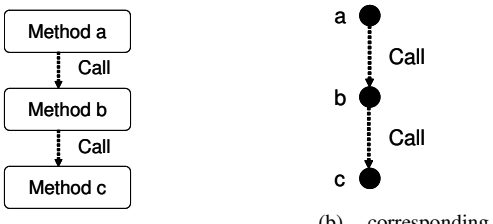
## 3.2 Chained Clone

Preparatory to define a *chained clone*, we define a terminology a *chained method*. A *chained method* means a set of methods that hold dependency relations. Here, we consider the following two types of dependency relations:

(1)Calling methods

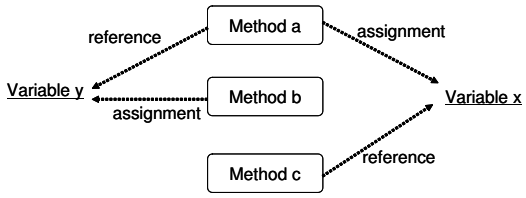
(2)Sharing variables (reference and assignment)

Figure 2(a) shows an example of a *chained method*, whose dependency relations are type(1). Here, the method $a$ calls $b$, and $b$ calls $c$. On the other hand, Figure 3(a) shows an example of a *chained method*, whose dependency relations are type(2). In this figure, $a$ assigns some values to $x$ and refers $y$, $b$ assigns some value to $y$, and $c$ refers $x$, respectively.

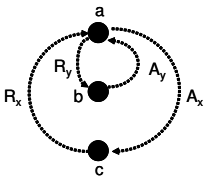



(a) example (b) corresponding graph

**Figure 2. Chained method whose dependency relations are calling methods**




(a) example




(b) corresponding graph

**Figure 3. Chained method whose dependency relations are sharing variables**

Next, we define the *chained clone*. At first, we transform the *Chained Method* into a labeled graph representation (named *chained method graph*). Here, a node represents a method and an edge represents a dependency relation. The types of labels for the dependency relation are as follows:

"Call": Calling methods,

"$A_i$": Sharing variable $i$ in terms of assignment, and

"$R_j$": Sharing variable $j$ in terms of reference.

Figures 2 and 3 include examples of a *chained method* and the corresponding graph representation.

For given *chained methods* $CM_1$ and $CM_2$, we transform them into *chained method graphs* $G_1$ and $G_2$. Then, for $G_1$ and $G_2$, if the following three conditions are satisfied, we call the pair of $CM_1$ and $CM_2$ as a *chained clone*.

(1) $G_1$ and $G_2$ are isomorphic,

(2) For each pair of the corresponding nodes between $G1$ and $G2$, a clone relation holds.

(3) In $G_1$ and $G_2$, a label of the corresponding edge is identical.

As in the case that as an equivalence class of a clone relation is called a clone set described in Section 2.1, an equivalence class of *a chained clone* is called a **Chained Clone Set**.

### 3.3 Typical Chained Clones and Applicable Refactoring Patterns

Here, we explain four cases of *chained clones*.

**Case 1**

Case 1 is a case that a *chained clone* is contained in a single class. We can merge them easily. Figure 4 shows an example of refactoring of Case 1. In this figure, the methods $a_{11}$ and $a_{12}$ are merged into the method $a_1$ and the methods $a_{21}$ and $a_{22}$ are merged into the method $a_2$ because they have a clone relation respectively.

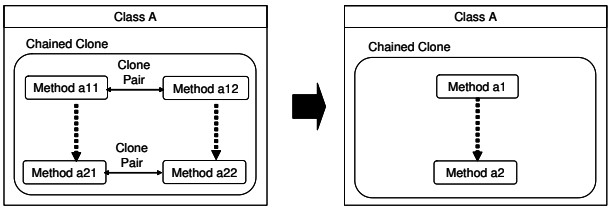



**Figure 4. Case 1**

**Case 2**

In Case 2, a *chained clone* satisfies the following two conditions:

- A clone relation is held among methods each of which has the same parent class.
- All method bodies of each *chained method* in the same class respectively.

In Case 2, each set of methods that have a clone relation can be merged into a new method in the parent class. It can be refactored by using "Pull Up Method" pattern. Figure 5 shows an example of refactoring in Case 2. Methods $a_1$ and $b_1$ are merged into the method $ab_1$ in the parent class $S$ and the methods $a_2$ and $b_2$ are merged into the method $ab_2$ in the parent class $S$.

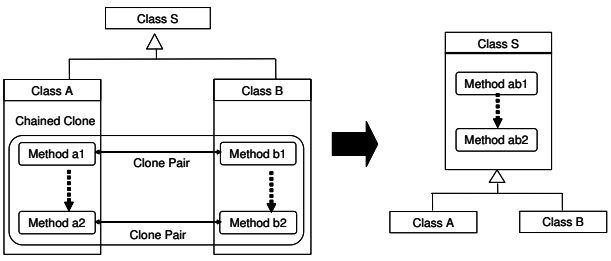



**Figure 5. Case2**

**Case 3**

In Case 3, the *chained clone* satisfies the following two conditions:

- A clone relation is held between methods each of which does not have the same parent class.

- All method bodies of each *chained method* in the same class respectively.

It is possible to conduct a refactoring by making a new parent class for the classes including the *chained methods*. That means "Extract Super Class" pattern is applied to the *chained methods* in a *chained clone* and then they are merged into new methods in the new parent class. Merging them into new methods in the new parent class, which means they can be refactored by applying "Extract Super Class" pattern. Figure 6 shows an example of a refactoring in Case 3. At first, a class $S$ is made as a parent class for classes $A$ and $B$. Then, methods $a_1$ and $b_1$ (that have a clone relation) are merged into the method $ab_1$ in $S$ and also methods $a_2$ and $b_2$ are merged into the method $ab_2$ in $S$.

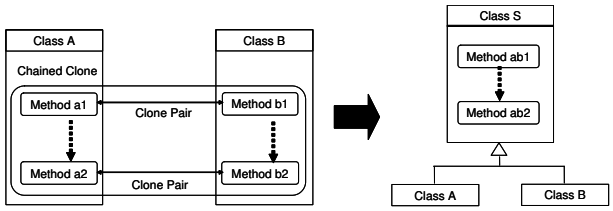



**Figure 6. Case 3**

**Case 4**

In Case 4, the *chained clone* satisfies the following conditions:

- A clone relation is held between methods each of which has the same parent class.
- *Chained methods* exist in different classes. That is, dependency relations exist among two or more classes.

In Case 4, each of the method having a clone relation can be merged into a new method in its parent class, which means they can be refactored by using "Pull Up method" pattern. Figure 7 shows an example of refactoring in Case 4. Methods $a_1$ and $b_1$ are merged into the method $ab_1$ in the parent class $S_1$ and also methods $a_2$ and $b_2$ are merged into the method $ab_2$ in the parent class $S_2$. However, the benefit of refactoring is smaller than Cases 1, 2 and 3.

### 3.4 Categorization of Chained Clones

In Section 3.3, we classified *chained clones* into four categories. Here, we propose a method to classify *chained clones* by using two metrics. Now, we consider the following two groups of methods for classifying *chained clones*.

**G1** The group of the methods having clone relations.

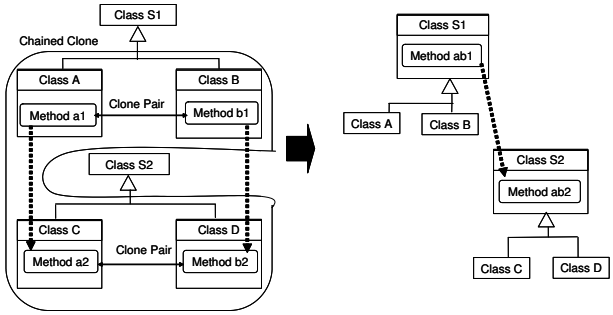



**Figure 7. Case 4**

**Table 1. Categorization of Chained Clones**

| G1 \ G2 | R1 | R2 | R3 |
|---|---|---|---|
| R1 | Category 11 | Category 12 | Category 13 |
| R2 | Category 21 | Category 22 | Category 23 |
| R3 | Category 31 | Category 32 | Category 33 |

**G2** The group of methods having dependency relations.

Next, we consider the distance and position relationship in the class hierarchy among methods in above two groups. Here, we classify the relations as follows:

**R1** All methods in group exist in the same class,

**R2** All methods in its group have a common parent class, and

**R3** Some methods in its group have no common parent class.

Combing the above groups and relations, we classify the *chained clones* into 9 categories shown in Table 1.

The following refactoring patterns can be applied to the categories 11, 21 and 31.

- Category 11

  Category 11 corresponds to Case 1. As shown in Figure 4, we can merge the *chained clone* in the same class.

- Category 21

  Category 21 corresponds to Case 2. As shown in Figure 5, we can apply "Pull Up Method" pattern to the *chained clone*.

- Category 31

  Category 31 corresponds to Case 3. As shown in Figure 6, we can apply "Extract Super Class" pattern to the *chained clone*.

On the other hand, it is difficult to refactor the *chained clone* all together for other categories.

This classification can be applied to the *chained clone set*. Figure 8 shows an example of a *chained clone set*. There is a common parent class $S$ among the methods that have clone relations. Also, each *chained method* is in the same class. So, it is classified into Category 21. Thus, "Pull Up Method" pattern can be applied to this *chained clone set* and we should get the same result as shown in Figure 6.

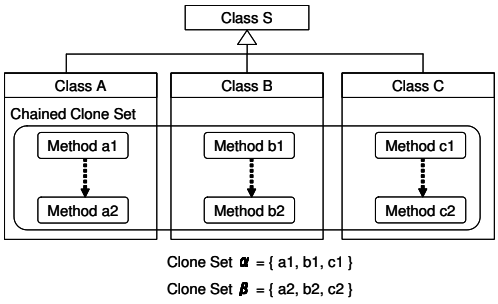



**Figure 8. Example of chained clone set**

Up to now, we explain a refactoring process for a *chained clone* that includes just two *chained methods*. Next, we describe a refactoring support method for *chained clone sets*.

### 3.5 Metrics to classify chained clone sets

Based on the above discussions, we propose two metrics for classifying *chained clone sets*. One is a metric to evaluate G1 and the other is to evaluate G2. The metrics represent the relations among the given methods in the class hierarchy. In order to investigate the relations, we measure the relations among the methods by using an idea of the metric $DCH(S)$ described in section 2.3.

We propose new metrics based on $DCH(S)$. First one is a metric **DCHS(CCS)** for evaluating G1. Suppose a *chained clone set* $CCS$ and each of *chained methods* consists of $m$ methods. Here, $CCS$ is divided into $n$ subset of clone sets, $S_1, S_2, \cdots, S_n$ (where $S_1 \cup S_2 \cup \cdots \cup S_n = CCS, S_i \cap S_j = \emptyset, 1 \leq i \leq n, 1 \leq j \leq n, i \neq j$). Also, clone set $S_i$ includes $m$ methods $M_1, M_2, \cdots, M_m$. Then, $DCHS(CCS)$ is defined as the following formula:

$$DCHS(CCS) = max\{DCH(S_1), \cdots, DCH(S_n)\}$$

Similarly, based on $DCH(S)$, we define a metric to evaluate G2. Suppose a *chained clone set* $CCS$ consists of $n$ *chained methods*, $CM_1, CM_2, \cdots, CM_n$ and each *chained method* $CM_i$ consists of methods $M_1, M_2, \cdots, M_m$. Then, **DCHD(CCS)** is defined as the following formula:

$$DCHD(CCS) = max\{DCH(CM_1), \cdots, DCH(CM_n)\}$$

Figure 9 shows how to calculate $DCHS(CCS)$. The *chained clone set* $CCS$ belongs to Category 21. $CCS$ includes three *chained methods* $CM_i$ each of which includes method $a_i$, $b_i$ and $c_i$ $(i = 1, 2)$. Here, $CCS$ is divided into two subsets of clone sets, $S_1$ and $S_2$ (where $S_1 \cup S_2 = CCS, S_1 \cap S_2 = \emptyset$). Then, the value of $DCH(S_1)$ and $DCH(S_2)$ is 1 because both $S_1$ and $S_2$ have the same direct parent class $S$. So, the value of $DCHS(CCS)$ becomes 1.

Next, using Figure 10, we explain how to calculate $DCHD(CCS)$. The *chained clone set* $CCS$ belongs to Category 21. $CCS$ includes three *chained methods* $CM_1, CM_2, CM_3$. Here, the value of $DCH(CM_1)$, $DCH(CM_2)$ and $DCH(CM_3)$ becomes 0 because each of *chained method* is in the same class, so that the value of $DCHD(CCS)$ becomes 0.

Based on $DCHS(CCS)$ and $DCHD(CCS)$, *chained clone sets* are classfied as in Table 2.

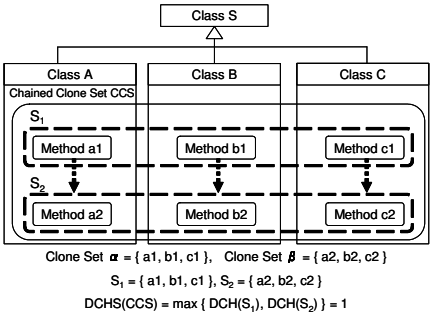



**Figure 9. Calculation of DCHS(CCS) metrics**

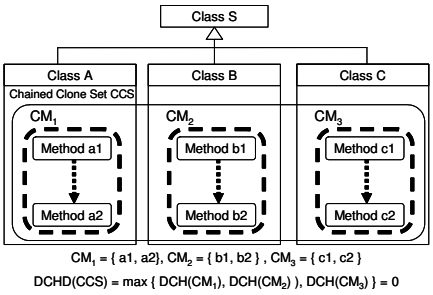



**Figure 10. Calculation of DCHD(CCS) metrics**

### 3.6 Implementation

We have implemented a proposed refactoring support method that detects *chained clones* from Java programs, calculates the proposed metrics and provides refactoring patterns to the user. Actually, we added the following functionalities to Aries.

**Table 2. Classifying Chained Clone Set Based on Proposed Metrics**

| Category | DCHS(CCS) | DCHD(CCS) |
|---|---|---|
| 11 | 0 | 0 |
| 21 | more than 1 | 0 |
| 31 | $\infty$ | 0 |
| other | Any | more than 1 |

**(F1)** Detection of *chained clone sets*,

**(F2)** Calculation of the proposed metrics for *the chained clone sets*, and

**(F3)** Provision of information of detected *chained clone sets* and metrics values through GUI interface.

As for (F1), using the information of clone sets detected by CCFinder, we analyze dependency relations among them and identify the *chained clone set*.

As for (F2), the metrics $DCHS(CCS)$ and $DCHD(CCS)$ are proposed based on the metric $DCH(S)$. So, we implemented (F2) by extending the functionality of calculating $DCH(S)$ in Aries.

As for (F3), we have added the following three GUI views to Aries(see Figure 11).

**Chained Clone Set Selection View :** It shows the list of *chained clone sets* and metrics of them. This view shows several metrics[1] including the proposed metrics ($DCHS(CCS)$ and $DCHD(CCS)$). The category of each *chained clone set* is shown for the user to understand which refactoring pattern could be applied to it.

**Chained Clone Set View :** It shows the code fragments (methods) that are included in the *chained clone* set that the user selected in the *Chained Clone Set Selection View*. This view consists of the *Code Clone List* and the *Dependency Relation View*. The *Dependency Relation View* is a table including *chained clone sets* and a list of variables which are used in each *chained clone set*.

**Source Code View :** It shows the source codes of a code fragment (method) that are selected in the *Chained Clone Set View* by the user.

In the actual refactoring process, at first, the user selects one of the *chained clone sets* in the *Chained Clone Set Selection View* based on the category of the *chained clone*. Then, he/she browses the list of methods included in the selected *chained clone set* through the *Chained Clone Set View*. Finally, he/she selects one of them and examines source codes of the selected methods whether the corresponding refactoring pattern can be applied to it.

[1] With respect to metrics LEN, POP, DFL, please refer to [8]

# 4 Evaluation

## 4.1 Overview

We have conducted case studies to evaluate the usefulness of the proposed method. In the evaluation, we focused on the following points:

- How many *chained clone sets* exist in an actual Java programs?
- Is it possible to classify *chained clone sets* by using the proposed metrics and to apply appropriate refactoring patterns to them?

The target software products are the following three Java programs.

- ANTLR 2.7.4 (47,000 LOC, 285 classes)
- Tomcat 5.5.4 (320,000 LOC,1214 classes)
- JBoss 3.2.6 (640,000 LOC, 3364 classes)

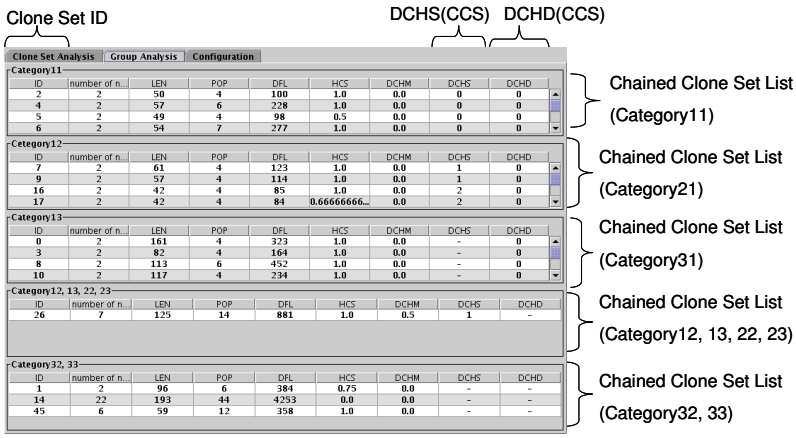
ANTLR is a language tool that provides a framework for constructing recognizers, compilers, and translators from grammatical descriptions containing Java, C# or C++ actions. Tomcat is the servlet container that is used in the official reference implementation for Java Servlet and Java Server Pages technologies. JBoss is a leading Open Source, standards-compliant, J2EE based application server.

## 4.2 Detected chained clone sets

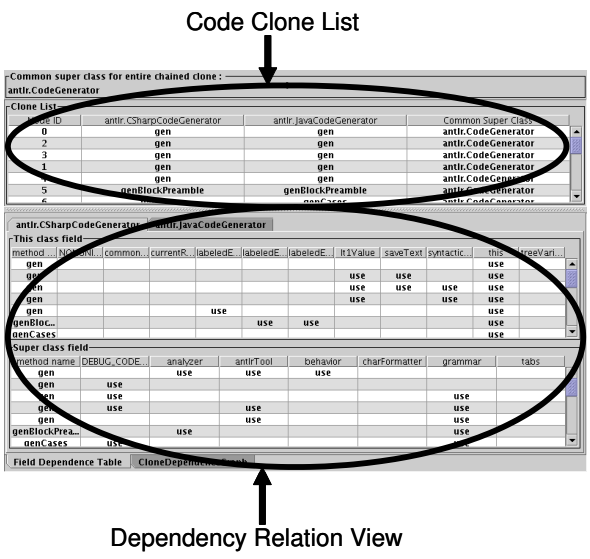
Tables 3, 4 and 5 show the results including the number of *chained clone sets* and the maximum/minimum number of methods included in them.

**Table 3. Chained clone sets in ANTLR**

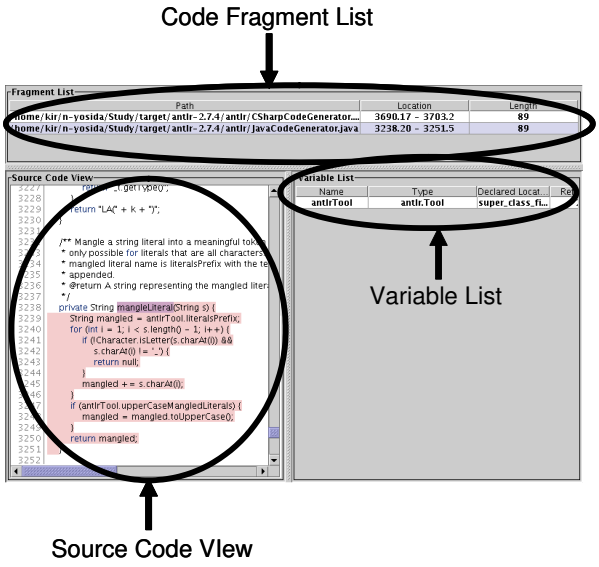
| Category | # of chained clone sets | Number of methods | |
|---|---|---|---|
| | | max | min |
| 11 | 3 | 4 | 4 |
| 21 | 6 | 40 | 6 |
| 31 | 1 | 4 | 4 |
| other | 0 | - | - |
| Total | 10 | - | - |




(a) Chained Clone Set Selection View




(b) Chained Clone Set View




(c) Source Code View

**Figure 11. Snapshots**

**Table 4. Chained clone sets in Tomcat**

| Category | # of chained clone sets | Number of methods | |
|---|---|---|---|
| | | max | min |
| 11 | 8 | 12 | 4 |
| 21 | 4 | 12 | 4 |
| 31 | 30 | 14 | 4 |
| 33 | 3 | 14 | 4 |
| other | 0 | - | - |
| Total | 45 | - | - |

**Table 5. Chained clone sets in JBoss**

| Category | # of chained clone set | Number of methods | |
|---|---|---|---|
| | | max | min |
| 11 | 16 | 13 | 4 |
| 21 | 17 | 8 | 4 |
| 31 | 13 | 29 | 4 |
| 23 | 1 | 14 | 14 |
| 33 | 3 | 44 | 12 |
| other | 0 | - | - |
| Total | 50 | - | - |

Totally, we could actually find the *chained clone sets* from the three Java programs. Next, we analyzed the result of each program.

**ANTLR**

We can see that 6 *chained clone sets* belonging to Category 21 are larger than other categories. Also, the number of methods in the *chained clone sets* of this category is relatively large (See Table 3). This reason is that ANTLR has similar functionalities for each language ( Java, C#, C++) and thus some of them inevitably become code clones. Actually, there is the *CodeGenerator* class constructing a parser, and it has three child classes (named *JavaCodeGenerator*, *CppCodeGenerator* and *CSharpCodeGenerator*) corresponding to these languages. There are a lot of code clones among these classes.

**Tomcat**

Table 4 shows that the number of Category 31 is relatively larger compared to the other categories. We consider that it is due to the characteristic that Tomcat consists of a lot of components and there are many types of connector that requests to browsers or Web servers.

**JBoss**

In JBoss, the number of *chained clone sets* is the largest among there programs. Also, it is peculiar that there is a *chained clone* of Category 23 (See Table 5). Figure 12 shows a class diagram of it. This *chained clone* includes another *chained clone* belonging to Category21.

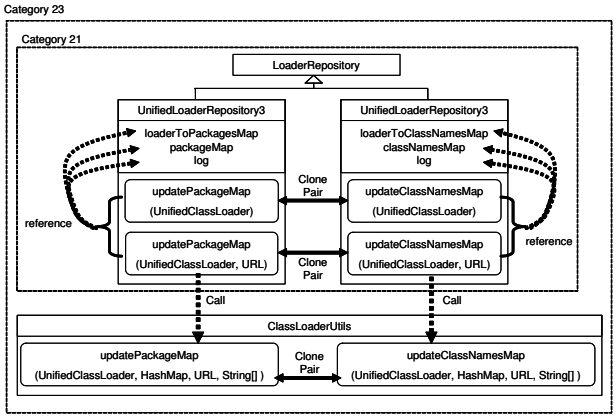
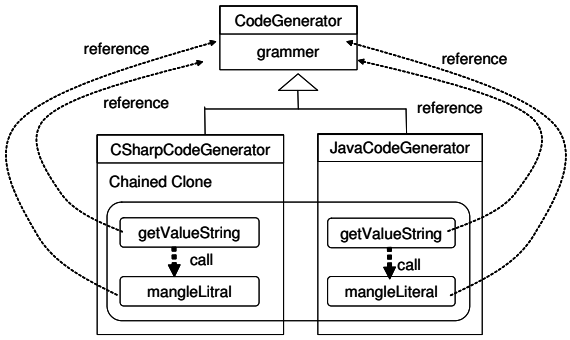



**Figure 12. Example of Category23**

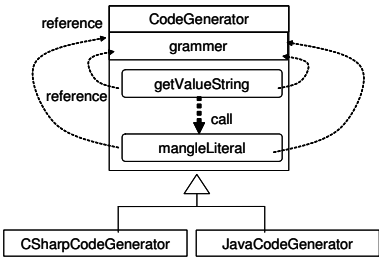
## 4.3 Refactoring for chained clone set

Next, we tryed to remove the *chained clone sets* extracted from ANTLR and Tomcat without chaging external behavior. At first, we modified source codes of them using refactoring patterns. Then, we conducted regression tests to confirm the original functionality being unchanged.

Figure 13 (a) shows an example of a *chained clone* (Category 21) from ANTLR. We have applied the "Pull Up Method" pattern to it and got the result shown in Figure 13 (b). You can see that code clones are correctly merged and also the coupling complexity among the parent and child classes is decreased.



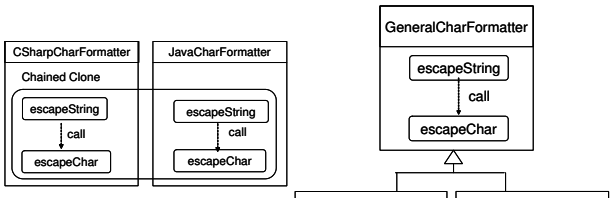

(a) Before refactoring




(b) After refactoring

**Figure 13. Refactoring in Category 21**

Figure 14 (a) shows an example of *chained clone* (Category31) detected from ANTLR. For this *chained clone*, we have applied "Extract Super Class" pattern and refactored without changing external behavior shown in Figure 14 (b).



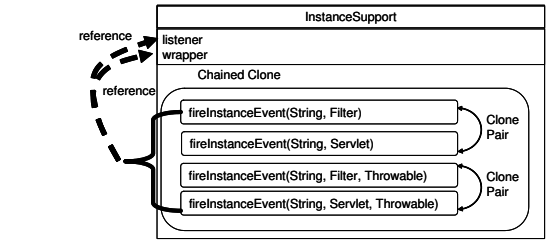

(a) Before refactoring

(b) After refactoring

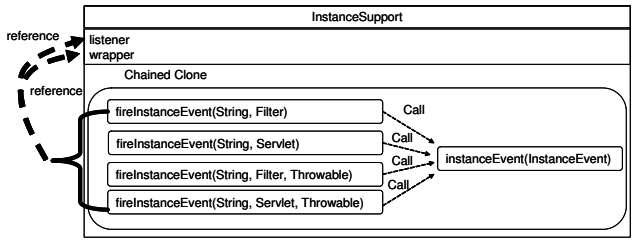
**Figure 14. Refactoring for Category 31**

Figure 15 (a) shows a *chained clone* (Category11) detected from Tomcat. Methods that are included in the *chained clone* are all the same name, but the number and type of these arguments are different. Also, methods having the same name have clone relations. The applicable refactoring pattern of Category11 is "Extract Method" pattern.

For this *chained clone*, we have applied "Extract Method" pattern and got the result shown in Figure 15(b).




(a) Before refactoring




(b) After refactoring

**Figure 15. Refactoring for Category 11**

## 5 Conclusion

In this paper, we focus on a refactoring for a *chained clone* that consists of sets of the methods with dependency relations. At first, we defined *chained clones*. Next, we classified *chained clones* according to their characteristic. Then, we discussed the applicable refactoring patterns to each category of *chained clones* and proposed metrics to automatically identify the category of them. Finally, we have implemented the proposed method to find and classify them by using metrics and added its to our code clone refactoring environment Aries. As the results of the case studies, we found a few dozen in *chained clones* in actual Java programs. As the result, we could remove most of them in Category 11, 12 and 31. We conclude that the proposed refactoring method for *chained clones* works effectively in refactoring for them.

As shown in Figure 12, some *chained clones* include another *chained clones* belonging to other category. We consider that it is necessary to provide information about the internal structure of *chained clones*. Also, we are going to apply our proposed method to some commercial Java programs and confirm whether there exists *chained clones* and whether we can apply appropriate refactoring patterns to them.

## Acknowledgment

This work is supported in part by MEXT, the Comprehensive Development of e-Society Foundation Software program, and JSPS.KAKENHI(17200001).

## References

[1] B. S. Baker, A Program for Identifying Duplicated Code, *In Proc. Computing Science and Statistics*, 24, pp.49-57, Mar. 1992.

[2] M. Balazinska, E. Merlo, M. Dagenais, B. Lague, and K. Kontogiannis, Advanced clone-analysis to support object-oriented system refactoring, *In Proc. of WCRE 2000*, pp.98-107, Nov. 2000.

[3] I.D. Baxter, A. Yahin, L. Moura, M.S. Anna, and L. Bier, Clone Detection Using Abstract Syntax Trees, *In Proc. of ICSM 1998*, pp.368-377, Mar. 1998.

[4] R. Komondoor, and S. Horwitz, Using Slicing to Identify Duplication in Source Code, *In Proc. of SAS 2001*, pp.40-56, Jul. 2001.

[5] J. Krinke, Identifying Similar Code with Program Dependence Graphs, *In Proc. of WCRE 2001*, pp.301-309, Oct. 2001.

[6] S. Ducasse, M. Rieger, and S. Demeyer, A Language Independent Approach for Detecting Duplicated Code, *In Proc. of ICSM 1999*, pp.109-118, Aug. 1999.

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

[8] Y. Higo, T. Kamiya, S. Kusumoto, and K. Inoue: ARIES: Refactoring Support Environment Based on Code Clone Analysis, *In Proc. of SEA 2004*, pp.222-229, Nov. 2004.

[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] Y. Ueda, T. Kamiya, S. Kusumoto, and K. Inoue, Gemini: Maintenance Support Environment Based on Code Clone Analysis, *In Proc. of METRICS 2002*, pp.67-76, Jun. 2002.

[11] ANTLR, http://www.antlr.org

[12] Tomcat, http://jakarta.apache.org/tomcat/

[13] JBoss, http://www.jboss.org