ARIES: Refactoring Support Tool for Code Clone

Yoshiki Higo[1] Toshihiro Kamiya[2] Shinji Kusumoto[1] Katsuro Inoue[1]
[1]Graduate School of Information Science and Technorogy, Osaka University
[2]PRESTO, Japan Science and Technology Agency
+81(06)6850-6571
{y-higo,kamiya,kusumoto,inoue}@ist.osaka-u.ac.jp

ABSTRACT

In this paper, we explain our refactoring support tool **Aries**. *Aries* characterizes code clones by several metrics, and suggests how to remove them.

1. Introduction

It is generally said that code clone is one of the factors that make software maintenance difficult. A code clone is a code fragment that is identical or similar to another. 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. So, efficient code clone detection and removal is necessary and important in software development and maintenance.

In this paper, we describe refactoring support tool **Aries**. *Aries* supports removing code clones from source code. Concretely, *Aries* characterizes code clones by some metrics, and suggests how to remove them. In other words, *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.

2. Preliminaries

Here, we define some terminology regarding code clones. Next, we briefly explain our code clone detection, a code clone detection tool **CCFinder**[4].

2.1. Code Clone

A clone relation is defined as an equivalence relation (i.e., reflexive, transitive, and symmetric relation) on code fragments[4]. 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.

3-WoSQ '05, May 17, 2005, St Louis, Missouri, USA.

2.2. Detecting Code Clone

CCFinder detects code clones both within files and across files from programs to output 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 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 number) 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[4].

3. Refactoring for Code Clone

3.1. Extraction of Refactoring-Oriented Code Clone

The removal of code clones is generally referred as **refactoring**[2] 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,

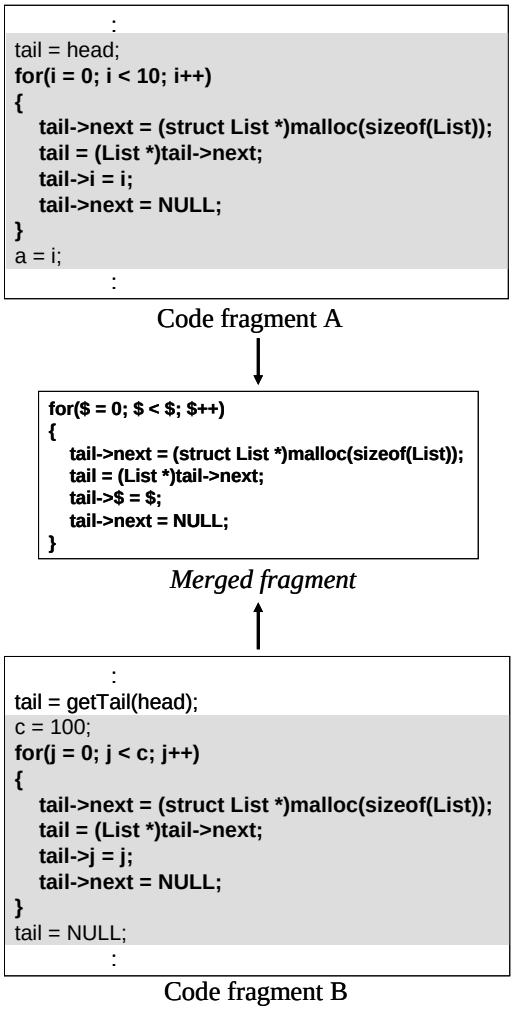

Figure1. Example of merging two code fragments

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 was implemented as a filter for the output of *CCFinder*. We named the filter **CCShaper**[3]. 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

3.2. Provision of Refactoring Pattern

CCShaper extracts the refactoring-oriented clones. Then, the user has to decide how to remove the code clones.
The rest of this section describes a solution to this problem. We have introduced some metrics to determine how to remove them. Extracted clones are quantitatively characterized by using the metrics which support the user how to remove them.

3.2.1. Refactoring for Code Clone Removal

We use existing refactoring patterns[2], especially "Extract Method" and "Pull Up Method", to remove code clones. "Extract Method" means that a fragment of source code is extracted and redefined as a new method[2]. Originally, this pattern is applied to too long method or too complex part. Here, 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[2]. 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.

3.2.2. Code Clone Metrics for Determining Refactoring Pattern

We attempt to identify which refactoring pattern is applicable to each code clone, by measuring its characteristics. For example, "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 it aren't used (referred and assigned) in it. 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 position and distance 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 $f_1, f_2, \ldots, f_n$. Code fragment f_i refers s_i-th variables which are defined outside it, and assigns to t_i-th variables which are also defined outside it. Then,

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

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 $f_1, f_2, \ldots, f_n$. C_i denotes the class which includes code fragment f_i.

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

$$DCH(S) = \max(D(C1,Cp), D(C2,Cp), \Lambda\ , D(Cn,Cp))$$

The value of DCH(S) also becomes larger as the degree of the dispersion of its clone set *S* becomes large. If all code fragments of a clone set *S* are in the same class, the value of its DCH(S) is set as 0. If all code 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 code fragment 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.
We define the upper and lower limits of these metrics for not only "Extract Method" and "Pull Up Method", but also "Form Template Method", "Parameterize Method" and so on. When the metric values of a clone set are within the limits, the corresponding refactoring pattern is applicable to the clone set.

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*[3]. Figure 2 shows 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 it on the *Clone Set Viewer*.

4.2. Function

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) [5] in addition to three metrics defined in Section 3.2.2. The existing metrics are defined as follows:
LEN(S): LEN(S) for clone set *S* is the average length of code fragment (the number of tokens) in *S*.

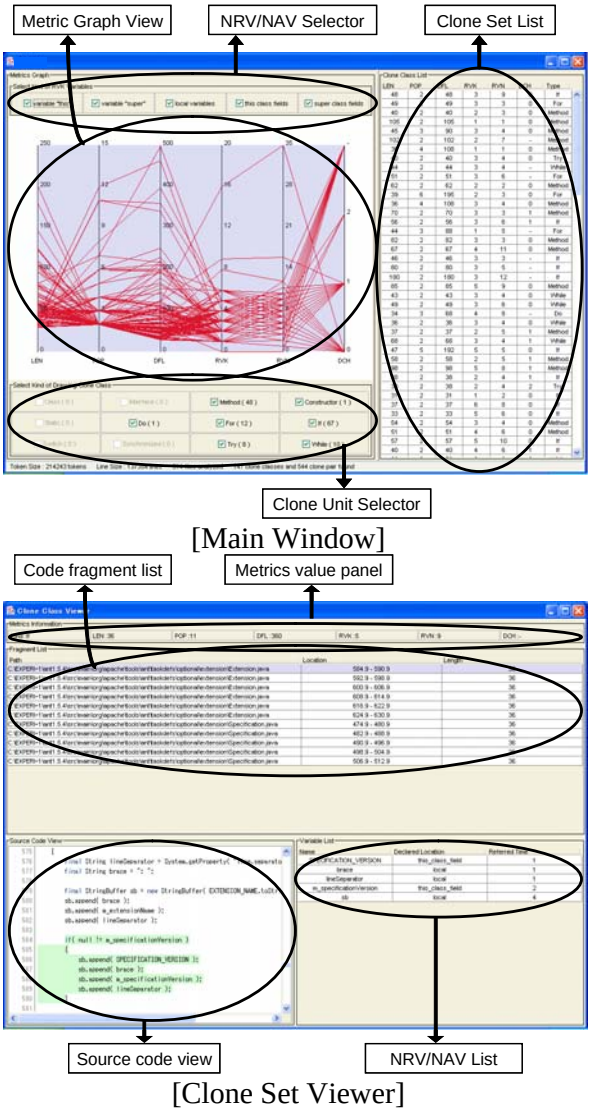

Figure2. Snapshots of Aries

POP(S): POP(S) is the number of code fragments of a given clone set *S*. A high value of POP(S) means that similar code fragments of *S* appear in many places.
DFL(S): 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). Before the reconstruction, LEN(S)xPOP(S) tokens are occupied in the source files. In the newly reconstructed source files, they occupy *k*xPOP(S) tokens (let *k* be the number of tokens for one caller statement) for caller statements and LEN(S) tokens for callee routine.

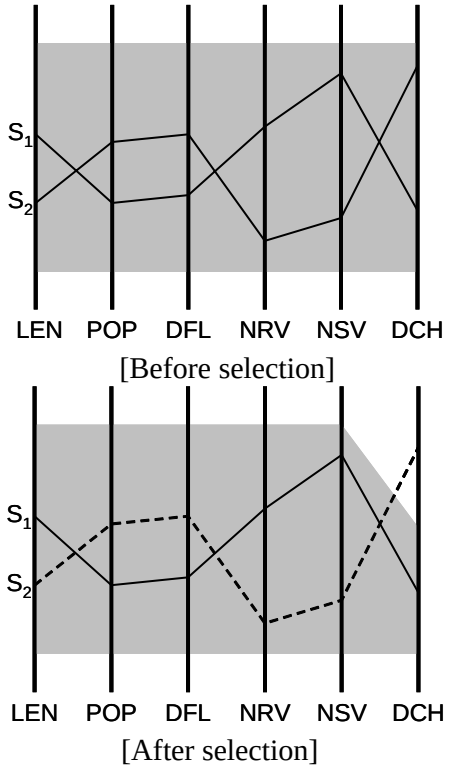

Figure3. Metric Graph

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 top graph, all metric values of S_1 and S_2 are between upper and lower limits. So, these two clone sets are said to be in *selected* state. In the bottom graph, the value of S_2 is bigger than the upper limit of DCH, which means S_2 is in *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, 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 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.

Clone Set List: The *Clone Set List* shows all clone sets which are filtered 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 4.2. 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.

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. Case Study

We have applied *Aries* to Ant[1], which is an open source program. We set 30 tokens as the minimum length of code clone and got 154 clone sets. Then, by using the proposed metrics, we identified clone sets that can be refactored by "Extract Method" or "Pull Up Method".
As the results, 52 clone sets could be merged by "Extract Method" and 12 ones could be merged by "Pull Up Method", respectively.

[1] Ant, |http://ant.apache.org|, 2003.
[2] M. Fowler, *Refactoring: improving the design of existing code*, Addison Wesley, 1999.
[3] 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.
[4] 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.
[5] Y. Ueda, T. Kamiya, S. Kusumoto, K. Inoue, *Gemini: Maintenance Support Environment Based on Code Clone Analysis*, 8th International Symposium on Software Metrics, pp.67-76, Ottawa, Canada, June, 2002.