

# Code Clone Analysis Environment for Software Maintenance

Yoshiki Higo<sup>1</sup>    Toshihiro Kamiya<sup>2</sup>    Shinji Kusumoto<sup>1</sup>    Katsuro Inoue<sup>1</sup>

<sup>1</sup>Graduate School of Information Science and Technology, Osaka University

<sup>2</sup>PRESTO, Japan Science and Technology Agency

{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, we have proposed a method that detects refactoring-oriented code clone. Through the application of them to software maintenance in several software companies, we got some feedback about them. In this paper, in order to improve the usefulness and applicability of the methods in the actual software maintenance, we extend both of our maintenance support methods. Concretely, we have developed a new code clone analysis tool **SuperGemini** that is improved the scalability by restructuring the architecture of Gemini. This improvement makes it possible to apply SuperGemini to industrial software practically. Also, as the extension of the refactoring method, we have developed a characterization of code clones by some metrics, which suggest how to remove them. Then, we have developed refactoring support tool **Cancer**. We expect SuperGemini and Cancer can support software maintenance more effectively.

## Categories and Subject Descriptors

D.1.m [Programming techniques]: Miscellaneous; D.2.6 [Software Engineering]: Programming Environments; D.2.8 [Software Engineering]: Metrics

## General Terms

Design, Languages, Management

## Keywords

Code clone, Refactoring, 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[14]. 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[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][13]. We have also developed code clone detection tool **CCFinder**[10] and code clone analysis environment **Gemini**[15]. We have been delivering Gemini (including CCFinder) to more than 50 software organizations and evaluated the usefulness of them in the actual software maintenance. Then, we have gotten valuable feedback from them. One of the problems to be solved in applying Gemini to industrial software is scalability. That means Gemini can’t deal with large scale software (more than 50000 LOC), effectively. So, the improvement of scalability of Gemini is essential problem. For the latter support, several code clone removal methods have been proposed[2][3][12]. 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 deal with the above problems, we develop two maintenance support systems based on code clone analysis. The first system is **SuperGemini** that have high scalability to cope with a huge size software in the actual software organization. The second system is **Cancer** to support the refactoring for code clone. Cancer can de-

tect refactoring-oriented code clones in practical time from large scale software. Moreover, Cancer characterizes detected code clone 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 SuperGemini and 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**[10], a code clone analysis system **Gemini** and a refactoring method for code clones detected by CCFinder. Finally, we show several problems to be solved that were known through applications of the tools in the actual software maintenance process.

### 2.1 Code Clone

A clone relation is defined as an equivalence relation (i.e., reflexive, transitive, and symmetric relation) on code fragments[10]. 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[10] 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 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.

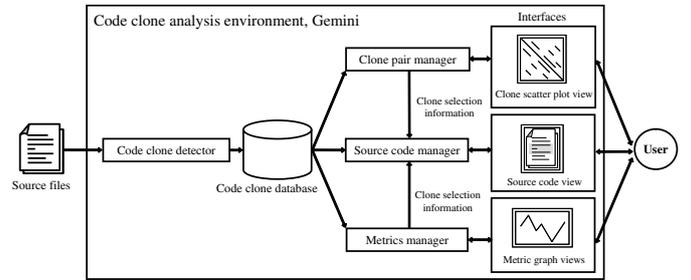


Figure 1: Architecture of Gemini

### 2.3 Gemini

Since CCFinder aims to detect code clones efficiently, the output from CCFinder is text format, which is difficult to analyze in practical maintenance process. So, we have implemented Gemini to support effective code clone analysis.

Gemini[15] is a GUI-based code clone analysis environment which uses CCFinder as a code clone detector. As shown in Figure 1, Gemini provides to the user the following view windows that enable an interactive code clone analysis: **Scatter Plot View**, **Metric Graph View**, **Source Code View**. *Scatter Plot View* shows visually where clone pairs exist in source files. It is very effective mechanism in early phase of code clone analysis since the state of distribution of code clone can be grasped at a glance. In the view, the user can select clone pairs by mouse dragging. The detail of the *Scatter Plot View* will be described later. *Metric Graph View* is used for the user to select clones by the quantitative characteristics of them. In the *Metric Graph View*, the user can easily select distinctive clone sets by setting the range of each metrics value. *Source Code View* works cooperating the *Scatter Plot View* on the *Metric Graph View*. The user can obtain actual source code corresponding to the clones selected in the other views.

Here, we briefly explain the *Scatter Plot*. Figure 2 shows an example of the *Scatter Plot*. Both the vertical and horizontal axes represent code fragments of source files. The following two sequences are used as sample code fragments in the scatter plot.

code fragment X: “ABCDCDEFBCDG”,  
code fragment Y: “ABCEFBCEBCD”

Here, symbols “A”, “B”, “C”, . . . are code fragments in an unit such as character, token, line, statement, function, etc (In Gemini, it is token). In Figure 2, each small black square means that corresponding two elements on the two axes are the same. So, a clone pair is shown as a diagonal line segment. If the same code fragments are arranged on the two axes, naturally, a diagonal line from the upper left to the lower right is drawn since such dot means comparison of token with itself, and the dots are symmetrical with a diagonal line.

### 2.4 Refactoring of code clones

We have also studied the removal of code clones from source code. 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 3 shows an example. In this figure, there are two code

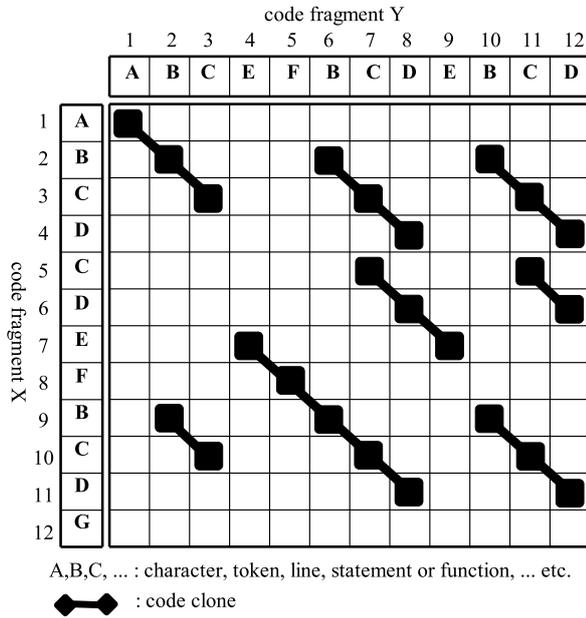


Figure 2: Scatter plot of code clones

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.

The proposed 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 kinds of code clone 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

In Step 1, time complexity  $O(nt)$  is required. Here,  $n$  is the token number included in target software, and  $t$  is the length of the longest code clone in it (the details are shown

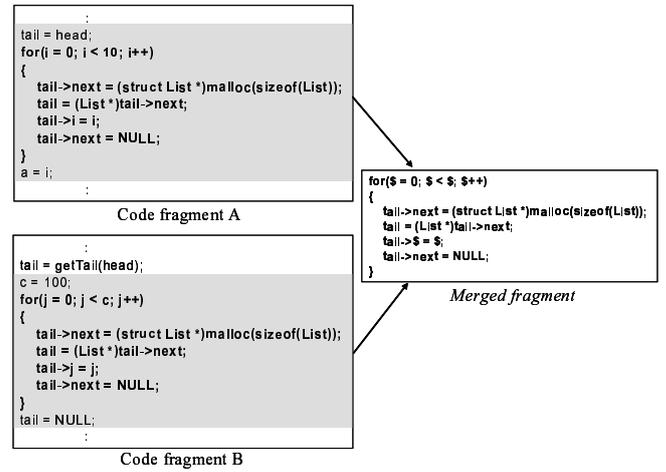


Figure 3: Example of merging two code fragments

in [10]). In Step 2, time complexity  $O(n)$  is required. Next, in Step 3, time complexity  $O(cs \log c)$  is required. Here,  $s$  is the number of target source files, and  $c$  is the average number of code clones in each file. Actually, the values of  $c$  and  $s$  are extremely smaller than the value of  $n$ . So, time complexity  $O(nt)$  is approximately required for detecting refactoring oriented code clones, which enables us to detect ones in practical time from large scale software.

There are several related studies about refactoring of code clones. Komondoor et al.[12] 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 re-ordered and intertwined clones[12] which cannot be detected 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] has proposed an approach to extract code clones using metrics. Since they consider the context of code clone, it is practical. But, the unit of the code clone is restricted to 'function' and 'method', which makes it difficult to perform refactoring to smaller unit of code clone.

## 2.5 Problems to be solved

We have so far delivered Gemini to more than fifty software organizations, and gotten many feedbacks from them. One of the problems which have been repeatedly pointed out is the low scalability of Gemini. The scatter plot of Gemini needs space complexity  $O(n^2)$  ( $n$  is the number of target source files) by its implementation. Moreover, if very many clones exist, the cost of plotting clones becomes huge and consequently the performance extremely deteriorates. From our experience and the related research[11], Gemini can perform smoothly if the LOC of target software is about 50,000 ~ 200,000 or less.

Also, there is another problem caused by the user-interface of Gemini. The *Scatter Plot View* is based on clone pair analysis. On the other hand, the *Metric Graph View* is based

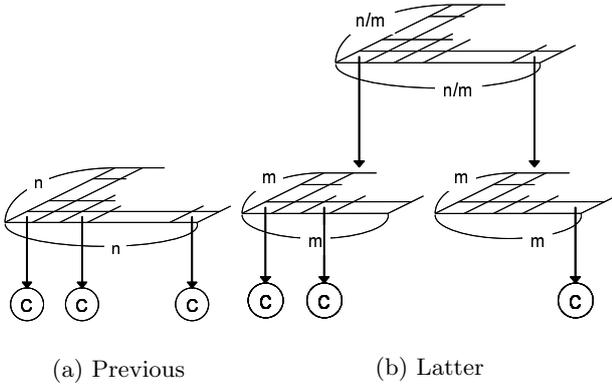


Figure 4: Data Structure of Scatter Plot

on clone set analysis. We considered that it is useful to use each view, separately. But, some users wanted to use the views cooperatively and then claimed that the operations were sometimes confused. So, the usability of them were not good.

With respect to the method of refactoring of code clones described in Section 2.4, we have just proposed the approach to extract the refactoring-oriented clones and did not consider how to remove them. So, the user have to decide how to remove the code clones by him/herself.

This paper describes the improvements for these problems. For the improvement of Gemini, we have added some new views, and changed architecture of it. As the results, the scalability of Gemini have been improved greatly. For the improvement of the refactoring method, 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. NEW CODE CLONE ANALYSIS ENVIRONMENT: SUPERGEMINI

#### 3.1 Key Idea for Improving Gemini

Through several experiments, we realized that the *Scatter Plot View* makes Gemini’s scalability lower extremely. The *Scatter Plot View*, by its nature, needs space complexity  $O(n^2)$  ( $n$  is the number of target source files) to draw clone pairs. Moreover, *Scatter Plot View* draws each clone pair respectively even if very many clones exist. Thus, it makes the cost of drawing very high.

So, we decided to contrive the implementation of the *Scatter Plot View*. At first, we reconstitute the data structure. Existing Gemini uses single 2-dimensional matrix as the data structure for the *Scatter Plot View*(Figure 4(a)). In Figure 4, © means clones existing in the cell corresponding to two axes. To cope with the problem, we contrive to use hierarchy 2-dimensional matrices. As shown in Figure 4(b), each cell of the top level matrix has a sub-matrix. If no code clones exist in files corresponding to this sub-matrix, it is not necessary to create the instance of this one, which leads to cutting down memory usage. Also, we revised the drawing component that if very many clones exist in some portions of source files, the portions are marked out collec-

tively. This contrivance leads to cutting down the cost of drawing clones.

With respect to the problem of user-interface, we changed the architecture of Gemini. Existing architecture is shown in Figure 1. In the revision of the architecture, we examined that there are following two categories of code clone analysis:

**File-Based Analysis:** It aims to evaluate how many code clones are included in each file. In this analysis, the user selects some files with interest. And, they check how many clones exist in those files or how those files are covered with clones, and so on.

**Clone-Based Analysis:** It aims to evaluate which code clones are distinguishing. This analysis, the user selects some clones with interest. And, they check where those clones exist in software, and so on.

So, we changed the Gemini’s architecture according to the categories.

Also, we introduce a new metric called  $LDL(C)$  to Gemini.  $LDL(C)$  means the rate how code clone  $C$  contains the same statement repeatedly. For example, the following code fragment (a code clone)  $C_1$  consists of three `System.out.println` statements.

```
System.out.println("The value of a is " + a);
System.out.println("The value of b is " + b);
System.out.println("The value of c is " + c);
```

In this case, a `System.out.println` statement is appeared three times in this fragment. So, the value of  $LDL(C_1)$  becomes 0.33. This metric enables the user to discriminate the clones, which include repeatedly the same statements, from other clones. Say, they are repeated **import** statements clones in Java language, or repeated **printf** and **scanf** statements clones in C language. The reason of introducing this metric also dues to the feedback from software companies.

#### 3.2 Implementation of SuperGemini

We have implemented above extensions as a new code clone analysis environment, **SuperGemini**. SuperGemini fundamentally doesn’t provide clone pair information to the user. *Scatter Plot View* isn’t used to display clone pair information, but used to show how many code clones exist between each file, which is used in file-based analysis. Figure 5 shows the architecture of SuperGemini. As shown in this figure, several new views, *Directory Tree View*, *Clone Set List*, and *File List* are included in SuperGemini. The *Directory Tree View* is mainly used with the *Scatter Plot View*. The *Scatter Plot View* shows clones which are under the directory or file indicated on the *Directory Tree View*. The *Clone Set List* is used in clone-based analysis. The *Clone Set List* shows clone sets selected on the *Metric Graph View*. Also, the *Clone Set List* has a sort function, which sorts clone sets according to ascending or descending order of each metric. We consider that the user operates these views as follows.

1. The user roughly specifies clone sets in the *Metric Graph View* with his/her interest metrics.
2. The user sorts the selected clone sets on the *Clone Set List* based on the metrics and specify the distinctive clone set.
3. The user browses actual code fragment of the specified clone set one by one.

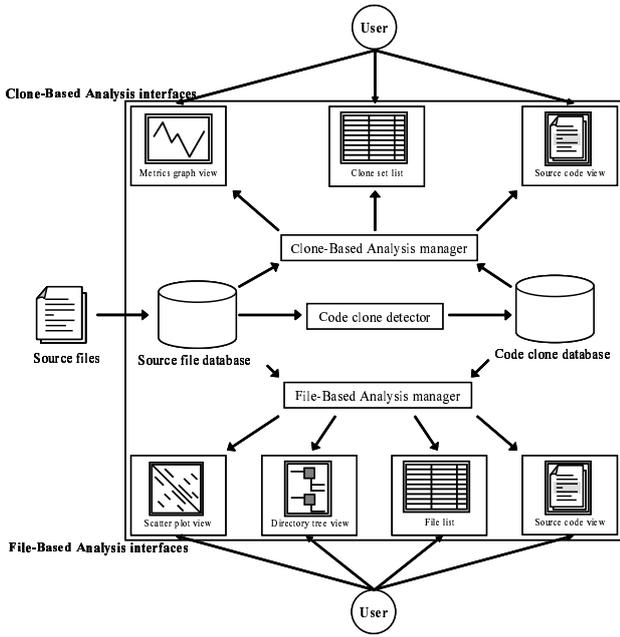


Figure 5: Architecture of SuperGemini

The *File List* is used in file-based analysis. The *File List* has the following information for each file.

- the number of tokens included in the file,
- the number of lines included in the file,
- the number of code clones included in the file,
- the rate how the file is covered with code clones,
- the list of code clones included in the file, and
- the list of files which include the same code clones with the file.

The *File List* provides the user quantitative clone information of each file. The user also can sort files based on each information. Say, it is very easy for the user to browse the source code of the file which includes the largest number of code clones. Here, we classify clone sets as the following three types based on its degree of dispersion in the file system.

**Dense:** In case that all the element of a clone set are included in the same file,

**Middle:** In case that all the element of a clone set are not included in the same file but in the same directory,

**Scattered:** In case that all the element of a clone set are included in neither the same file nor the same directory.

The reason why we use this classification is that the way how to deal with code clones is different from their degree of dispersion[11]. The *Source Code View* has also been adjusted to this classification. That is, the code clones shown in the *Source Code View* are highlighted by the different colors based on each dispersion. So, the user can figure out the degree of dispersion of them at a glance.

## 4. REFACTORING METHOD FOR CODE CLONE

### 4.1 Key Idea

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.

### 4.2 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 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  $\{f_1, f_2, \dots, f_n\}$ . Code fragment  $f_i$  uses externally defined variables  $\{v_{i_1}, v_{i_2}, \dots, v_{i_{m_i}}\}$ . 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,

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

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

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

Then, if the classes  $\{C_1, C_2, \dots, C_n\}$  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  $\{C_1, C_2, \dots, C_n\}$ . Also,  $D(C_k, C_h)$  represents the dis-

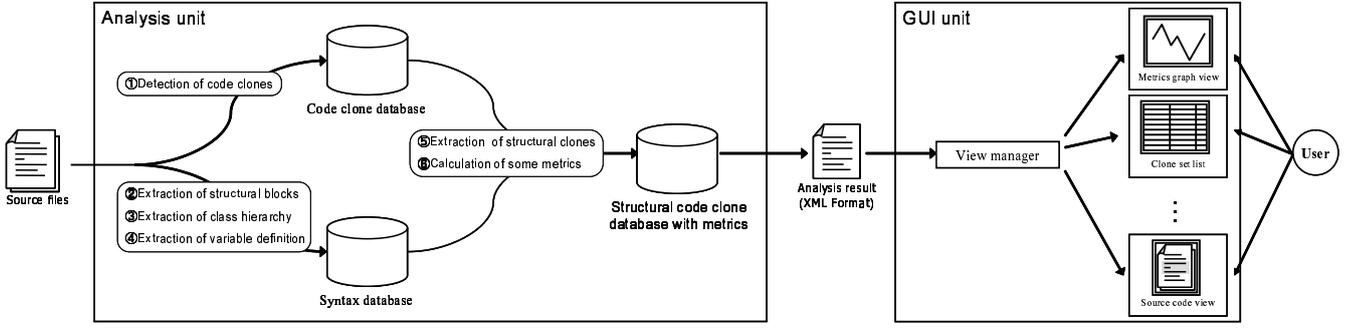


Figure 6: The analysis flow of Cancer

tance between class  $C_k$  and class  $C_h$  in the class hierarchy.

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

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.

### 4.3 Refactoring Support Tool: Cancer

Based on the proposed method, we have implemented a refactoring support tool **Cancer** with Java language. Figure 6 shows the analysis flow of Cancer. Cancer consists of two units, Analysis unit and GUI unit. Analysis unit performs the following analyses.

1. Detection of code clones,
2. Extraction of structural blocks,
3. Extraction of class hierarchy,
4. Extraction of variable definition,
5. Extraction of structural clones,
6. Calculation of some metrics.

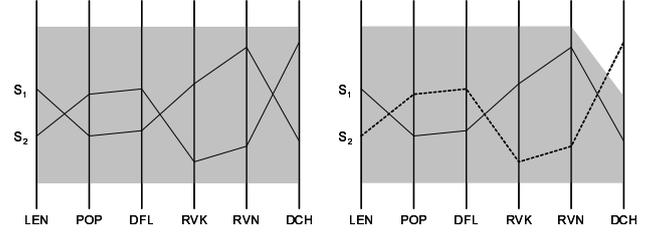
For detection of code clones, Cancer internally calls CCFinder[10].

For the analyses which need syntax or semantic analyses, we used JavaCC[8], which is an open source code generator written in Java language. The analysis result, that is structural clone information with metrics, is passed to GUI unit as XML format. Figures 7(a) and 7(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*.

### 4.4 Function of Each Component

Here, we explain each component on Cancer.



(a) Before selection

(b) After selection

Figure 8: Metric Graph

#### 4.4.1 Metric Graph View

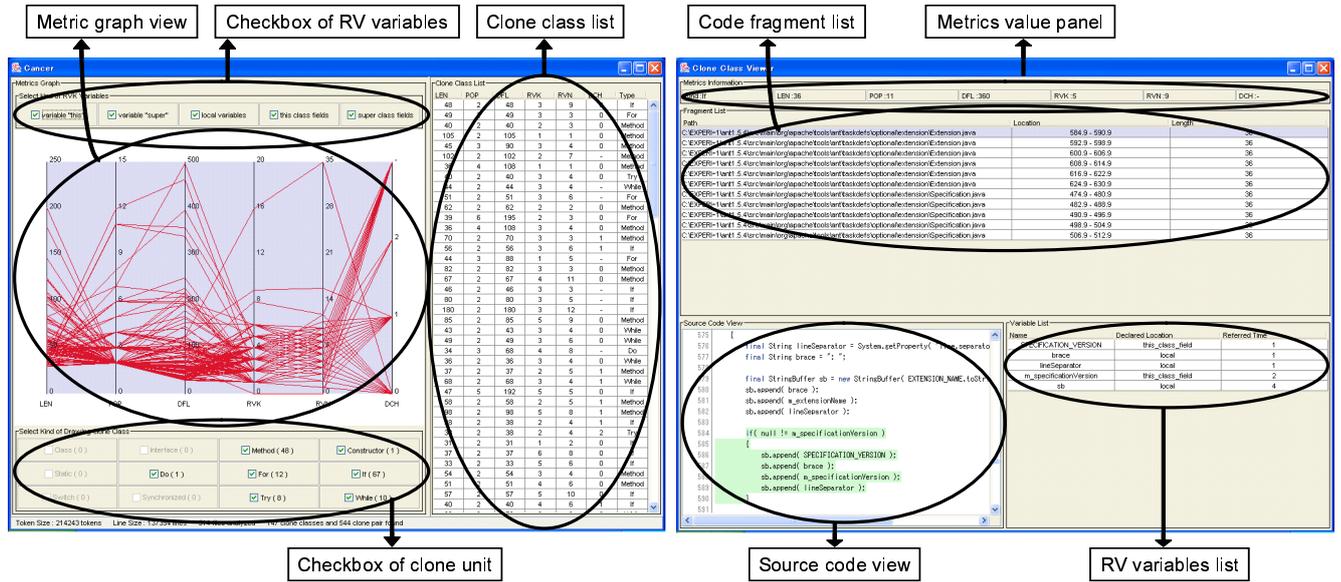
The *Metric Graph View* uses existing metrics,  $LEN(S)$ ,  $POP(S)$ , and  $DFL(S)$  [15] in addition to three metrics defined in Section 4.2. The followings are brief explanations of each metric.

**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 Graph View* using an example shown in Figure 8. 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(8(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(8(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



(a) Main Window

(b) Clone Set Viewer

Figure 7: Snapshots of Cancer

set by changing upper and lower limits of each metric. And, the result of selection is reflected on the *Clone Set List*.

#### 4.4.2 Checkbox of RV Variables

In the *Checkbox of RV Variables* in Figure 7(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.

#### 4.4.3 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 2.4. 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.

#### 4.4.4 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 7(b). It shows more detail information of the selected clone set.

#### 4.4.5 Metrics value panel

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

#### 4.4.6 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.

#### 4.4.7 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.

#### 4.4.8 RV variables list

The *RV Variables 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 kind of its variable and the count of used.

### 4.5 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,

**Table 1: Target software**

	Student's	Ant 1.6	Eclipse 2.1.3
Num. of Files	70	627	7920
LOC	7200	180,000	1,690,000
Detection time	2 seconds	20 seconds	5 minutes

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 than 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)The 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 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 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 ( $0 \leq DCH(S) \leq 1$ ), and the upper limit of  $RVK(S)$  as less than 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

Here, we describe several case studies to evaluate the usefulness of SuperGemini and Cancer.

### 5.1 SuperGemini

The objective of the case study for SuperGemini is confirm how the scalability of SuperGemini has improved as compared with Gemini. In this case study, we applied SuperGemini to small, middle and large scale software. For each application, we analyzed the detection time of code clone (by CCFinder), the initialization time (reading code clone information file and initializing each view) of SuperGemini and Gemini and the used memory size. Of course, several conditions for the code clones are the same both in SuperGemini and Gemini except the memory size assigned to JVM.

Table 1 shows the scale and clone detection time for each software, and Table 2 shows the result of each application.

The small scale software is a student’s program of Osaka University. In this application, the used memory size of Gemini is a little bit lower than one of SuperGemini. We consider that the increased memory size of SuperGemini is caused by introducing new metrics. For the application of middle scale software, Ant 1.6.0[1], there is a remarkable difference between Gemini and SuperGemini. The initialization time of Gemini is 15 second, which is so impractical comparing with one of SuperGemini. Also, the used memory size of SuperGemini is 35% smaller than one of Gemini. This improvement is caused by introducing a new data structure of the *Scatter Plot View*. These difference becomes more remarkable in application to the large scale software, Eclipse 2.1.3[5]. In this case, in the execution of Gemini, we assigned 1GB memory to JVM (SuperGemini and Gemini are implemented in Java language). But, it was insufficient for Gemini and so Gemini couldn’t work. On the other hand, in the execution of SuperGemini, though we only assigned 100MB memory to JVM, SuperGemini worked smoothly. So, we can say that restructuring of the architecture makes marvelous improvement of the scalability.

## 5.2 Cancer

### 5.2.1 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
“Extract Method”	32
“Pull Up Method”	20

The conditions of “Extract Method” and “Pull Up Method” are the same as ones described in Section 4.5. In Section 5.2.2 and 5.2.3, 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[9], 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.

### 5.2.2 “Extract Method”

As described above, we extracted 32 clone sets using the “Extract Method” conditions described in Section 4.5. 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.

**Table 2: Result of case study**

	Gemini			SuperGemini		
	Student's	Ant 1.6.1	Eclipse 2.1.3	Student's	Ant 1.6.1	Eclipse 2.1.3
Initialization time	2 seconds	15 seconds	-	1 seconds	2 seconds	20 seconds
Memory usage	26MB	58MB	-	35MB	38MB	100MB

**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());
}
```

**Figure 9: Example of Extract Method in Group 1**

Three clone sets were classified to Group 1. Figure 9 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);
}
```

**Figure 10: Example of Extract Method in Group 2**

Eighteen clone sets were classified to Group 2. Figure 10 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);
    }
}
```

**Figure 11: Example of Extract Method in Group 3**

Seven clone sets were classified to Group 3. Figure 11 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.

Four clone sets were classified to Group 4. Figure 12 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.

### 5.2.3 “Pull Up Method”

Next, we describe the results of applying ‘Pull Up Method’. As described above, we extracted 20 clone sets using the

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

**Figure 12: Example of Extract Method in group 4**

“Pull Up Method” conditions described in Section 4.5. 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());
    }
}
```

**Figure 13: Example of Pull Up Method in group 2**

Ten clone sets were classified to Group 2. Figure 13 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 3. Figure 14 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”.

Eight clone sets were classified to Group 4. Figure 15 shows a source code of one of them. In this method, the method “checkOptions” was called. This method was defined in

```

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 14: Example of Pull Up Method in group 3

```

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 15: Example of Pull Up Method in group 4

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.

## 6. CONCLUSION

In this paper, we have proposed a new code clone analysis environments. We have implemented SuperGemini that is capable to analyze huge scale software. Also, we have provided some new viewers to SuperGemini and attained high usability compared with Gemini. Moreover, we have proposed a new refactoring method for code clones, 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.

In the present, SuperGemini and Cancer have been developed independently. So, some components are implemented on both of them (ex. the *Metrics Graph View*). It is necessary to integrate them and develop a total maintenance environment based on code clone analysis.

Also, we are going to deliver SuperGemini to software companies to confirm the impact of improving architecture, because this improvement is based on their requirement. Also, we are going to apply Cancer to several industrial software.

## 7. 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] JavaCC, <http://javacc.dev.java.net>, 2003.
- [9] JUnit, <http://www.junit.org>, 2003.
- [10] 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).
- [11] Cory Kapser and Michael W. Godfrey, *Toward a Taxonomy of Clones in Source Code: A Case Study*, Evolution of Large-scale Industrial Software Applications (ELISA), Amsterdam, The Netherlands, September 23, 2003.
- [12] 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.
- [13] 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.
- [14] Pigoski T. M., *Maintenance*, Encyclopedia of Software Engineering, 1, John Wiley & Sons, 1994.
- [15] 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.