Gemini: Maintenance Support Environment Based on Code Clone Analysis

Yasushi Ueda†, Toshihiro Kamiya‡, Shinji Kusumoto† and Katsuro Inoue†

†Graduate School of Engineering Science,
Osaka University,
Toyonaka, Osaka 560-8531, Japan
Phone:+81-6-6850-6571, Fax:+81-6-6850-6574
{y-ueda, kusumoto, inoue}@ics.es.osaka-u.ac.jp

‡PRESTO, Japan Science and Technology Corp.
Current Address:
Graduate School of Engineering Science,
Osaka University,
Toyonaka, Osaka 560-8531, Japan
Phone:+81-6-6850-6571, Fax:+81-6-6850-6574
kamiya@ics.es.osaka-u.ac.jp

Abstract

Maintaining software systems is getting more complex and difficult task, as the scale becomes larger. It is generally said that code clone is one of the factors that make software maintenance difficult. A code clone is a code portion in source files that is identical or similar to another. If some faults are found in a code clone, it is necessary to correct the faults in its all code clones. However, for large scale software, it is very difficult to correct them completely. In this paper, we develop a maintenance support environment, called Gemini, which visualizes the code clone information from code clone detection tool, CCFinder. Using Gemini, we can specify a set of distinctive code clone through the GUI (scatter plot and metrics graph about code clones), and refer the fragments of source code corresponding to the clone on the plot or graph.

Keywords: software maintenance, code clone, software metrics

1 Introduction

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].

The maintenance phase is the most expensive in software life cycle. Many research studies have reported that large software companies spent a lot of cost to maintaining the existing systems. For example, Yip and Lam reported that 66% of the cost in software life cycle is spent on maintenance phase [15].

Code clone is one of the factors that make software maintenance more difficult. A code clone is a code portion in source files that is identical or similar to another. Clones are introduced because of various reasons such as reusing code by 'copy-and-paste', mental macro, or intentionally repeating a code portion for performance enhancement, etc[4].

Modification and enhancement of legacy system would introduce code clones. Code clones make the source files very hard to modify consistently. For example, assume that a software system has several clone subsystems created by duplication with slight modification. When a fault is found in one subsystem, the engineer has to carefully modify all other subsystems. In order to detect the code clones effectively, various clone detection methods have been proposed[1][2][3][4][5][10][12].

We have proposed and developed a code clone detection tool, **CCFinder**[11], that detects code clones from single program or multiples. It has been developed to support maintenance for large scale software and evaluation of the programs in an educational environment. But, since the output of CCFinder shows only the information of the location of similar code fragment pairs, we cannot understand the output intuitively, especially for large scale software. So, in order to maintain large scale software by using the code clone information efficiently, it is necessary to develop a total maintenance support environment which includes the mechanism to immediately grasp the correspondence between the code clones and the actual code fragment and easily modify the code clones.

In this paper, we propose a maintenance support environment, **Gemini**, which provides the user with the useful functions to analyze the code clones and modify them. In Gemini, CCFinder is one of the components of Gemini and used to detect code clones. Gemini primarily provides two diagrams: scatter plot and metrics graph. The scatter plot graphically shows the locations of code clones among source codes. The metrics graph shows metric value of each clone and has a feature to identify the distinctive code clones. So, using Gemini, we can specify the code clones that should be taken notice in the maintenance phase. Furthermore, Gemini can display source code corresponding to

```
#version: ccfinder 3.1
#langspec: JAVA
#option: -b 30,1
#option: -k +
#option: -r abcdfikmnprsv
#option: -c wfg
#begin{file description}
0.0 52  C:\Gemini.java
0.1 94  C:\GeneralManager.java
0.2 237 C:\MDI.java
1.0 7   C:\CCFEventListener.java
1.1 116 C:\CCFinderManager.java
1.2 695 C:\CCFinderOptionFrame.java
             :
             :
#end{file description}
#begin{clone}
0.1 53,9   63,13   1.10 542,9  553,13 35
0.1 53,9   63,13   1.10 624,9  633,13 35
0.2 124,9  152,31  0.2  154,9  216,51 42
0.2 124,9  152,31  1.10 194,9  225,30 42
0.2 126,9  152,31  1.10 185,9  204,34 37
0.2 153,14 211,9   1.10 207,9  242,5  31
0.2 153,14 216,51  1.10 193,9  225,30 44
0.2 172,9  216,51  1.10 185,9  204,34 37
             :
             :
#end{clone}
```

Figure 1. Example of output from CCFinder

the code fragments, so that refactoring of the codes can be carried out with high maintainability. In this study, we apply Gemini to an actual program development to show the usefulness of it.

In Section 2, we briefly introduce a code clone detection tool, CCFinder. Section 3 explains approach, features, design and implementation of Gemini. Section 4 applies Gemini to programming exercise in university and then analyzes the results. Finally, Section 5 concludes this paper.

2 Preliminaries

2.1 Definitions on code clone

A clone relation is defined as an equivalence relation (i.e., reflexive, transitive, and symmetric relation) on code portions. A clone relation holds between two code portions 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 portions is called clone pair if the clone relation holds between the portions. An equivalence class of clone relation is called clone class. That is, a clone class is a maximal set of code portions in which a clone relation holds between any pair of code portions. A code portion in a clone class of a program is called a code clone or simply a clone.

2.2 CCFinder

CCFinder detects code clones from programs and outputs the locations of the clone pairs on the programs.

Clone detection of CCFinder is a process in which the input is source files and the output is clone pairs. The process consists of 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 portions 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.

Details of CCFinder have been shown in [11].

2.3 Problem to be solved for maintenance

CCFinder has no GUI but it only generates character-based output. Figure 1 shows an example of the output results from CCFinder. In Figure 1, main information for code clones are described between `#begin{clone}` and `#end{clone}`. Here, for example, the code from the 53rd line to the 63rd line in the file (0.1) and the code from the 542nd line to the 553rd line in the (1.10) are detected as a code pair[1]. It is quite difficult for the person who analyzes the source code to investigate a code clone only from this information and source code, and to perform analysis of the source code and reconstruction of it.

3 Source code analysis environment: Gemini

3.1 Design policy

Various clone detection tools have been implemented. Among them, DUPLOC [6] has useful GUI mechanism. DUPLOC extracts clone pairs from source files which are implemented in various programming languages, and also offers a visual support for code clone analysis. The user can click the scatter plot(See Section 3.2.2) to edit code sections of the clone.

We also adopt the scatter plot as one of the interface mechanism in Gemini. It is very effective mechanism to analyze code clones since the state of distribution of code

[1](x.y) is used to specify the each of the input source files.

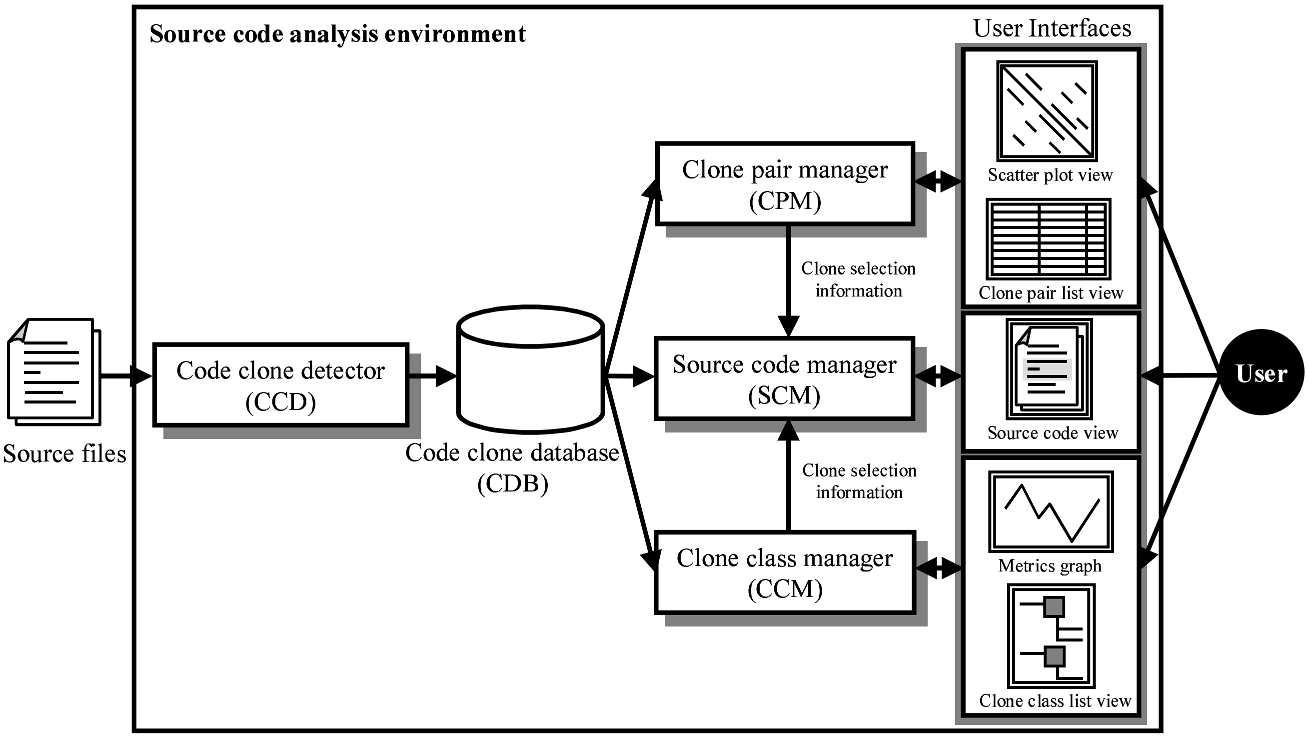


Figure 2. Architecture

clone can be grasped at a glance. Also, it can be applied to provide the user with the interactive operations for the code clones and the source files where the clones exist.

However, as for large scale software in which there are many code clones, it is very difficult to decide which plot (that is code clone) in the huge scatter plot should be kept our eye on. Since we are interested in applying Gemini to large scale software, we have to devise the way how to decrease the difficulty in searching code clones which should be taken notice. Our devised approach to operate the scatter plot will be shown in Section 3.2.2.

Also, it is important to extract the distinctive code clones from the target programs. For the purpose, we use several metrics for code clone(See Section 3.2.3). To easily select the distinctive code clone, we adopt the parallel coordinate plot [16] as one of the interface mechanism in Gemini. The details of the graph will be shown in Section 3.2.3.

3.2 Architecture

Gemini invokes CCFinder internally and analyzes the outputs from CCFfinder. The architecture of Gemini is shown in Figure 2.

Gemini mainly consists of five components: (1) Code clone detector, (2) Clone pair manager, (3) Clone class manager, (4) Source code manager and (5)User Interface. First, when a user specifies target source files, Gemini executes the Code clone detector (CCD), which includes CCFinder, with setting the several options. Then, the results (code clone information) of CCD are accumulated into the Code clone database(CDB). Based on the data in CDB, the user analyzes the code clones through several graphical user interfaces controlled by Clone pair manager(CPM), Clone class manager(CCM), and Source code manager (SCM).

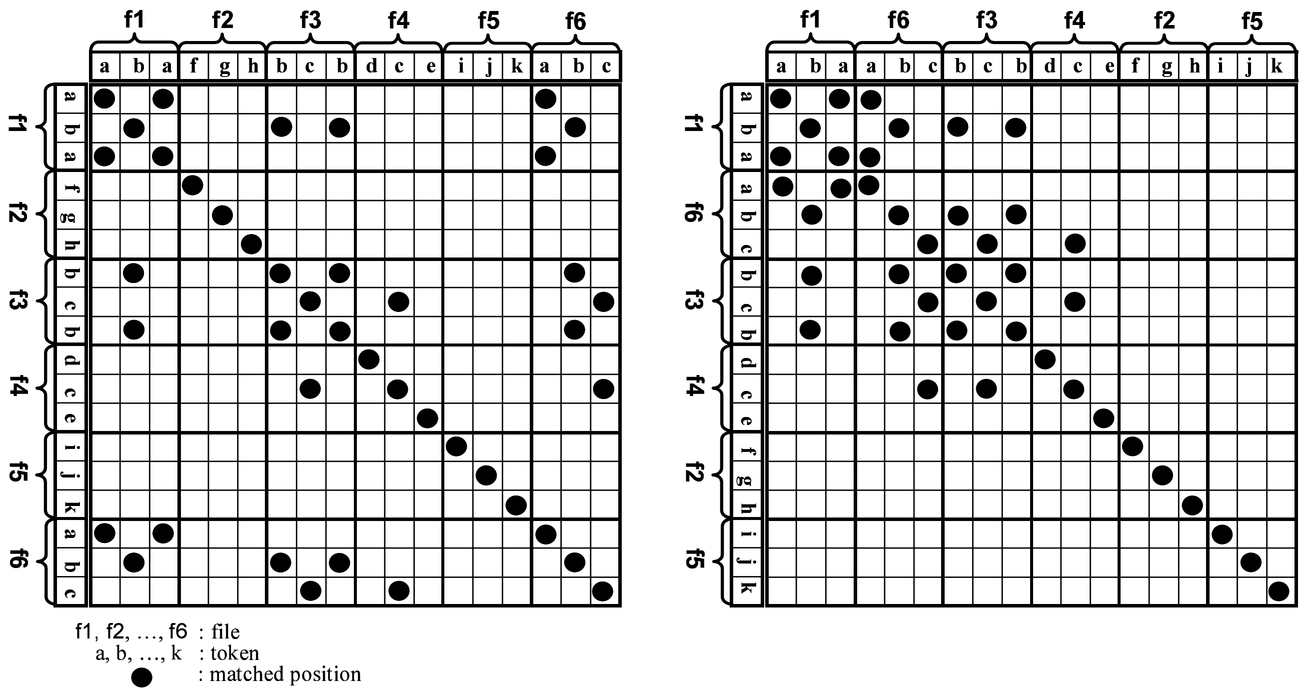
The details of the each component in Figure 2 are explained in the following subsections.

3.2.1 Code clone detector (CCD)

CCD manages the operations for CCFinder. The programming language (C/C++, Java, COBOL, Plain Text) of target files, the minimum length of code clone which CCFinder detects and other options are specified by the user through the User Interface.

3.2.2 Clone pair manager (CPM)

Complying with a request from the user, CPM gets the required code clone data from Code clone database (CDB) and shows it through the scatter plot viewer and clone pairs list viewer. Through the viewers, the user can specify a set of clone pairs to make them in *selected state*. Figure 3



(a) Without sorting

(b) Sorting based on RSA and RST

Figure 4. A simple example of scatter plot

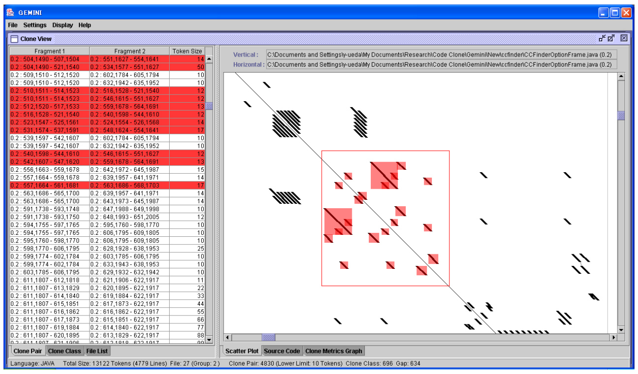


Figure 3. GUI Snap shot of scatter plot viewer (right side) and clone pairs list viewer (left side)

shows examples of scatter plot. If the user selects the area of the set of clone pairs, the area is surrounded by the quadrilateral shown in Figure 3. Then, the corresponding code fragments on the actual source code are also emphasized through source code viewer.

Here, we explain the details of scatter plot viewer.

Scatter plot

Figure 4(a) shows simple examples of simple scatter plot. Both the vertical and horizontal axes represent lines of source files. The files are sorted in alphabetical order of the file paths, so that files in the same directory are also located near on the axis. A clone pair is shown as a diagonal line segment.

In Figure 4(a) in order to simplify the plot, each file includes only three tokens and each token is located on each separate line. For example, the file `f1` includes three tokens (*a, b* and *a*) and `f2` includes three tokens (*f, g* and *h*). A black dot means that the corresponding tokens on the horizontal and the vertical axis are the same. Naturally, a diagonal line from the upper left to the lower right is drawn since such dot means comparison of token with itself. The dots are symmetrical with a diagonal line.

Gemini provides the user with the following functions to operate the scatter plot:

- Browsing the part of source code which correspond to the user selected clone pairs,
- Zooming the user specified area,
- **Sorting** the order of files on the coordinate axis, and
- Hiding the files in which no code clones are included.

Sorting is the most distinctive function of Gemini for decreasing the difficulty in the analysis of code clones in a

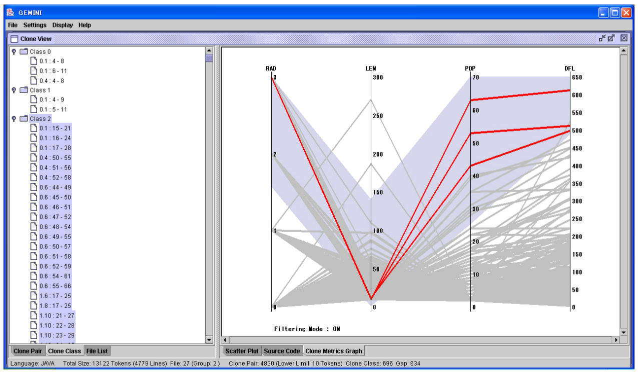


Figure 5. GUI Snap shot of metrics graph (right side) and clone class list viewer (left side)

large scale software. If the files are located on the axis of coordinate in naive order, such as alphabetical order with file name, the distribution of code clones is occasionally spread widely without conspicuous deviation as shown in Figure 4(a). Then, it is difficult to judge which portion should be paid attention to and much cost is needed for the analysis. So, sorting aims to decrease the cost by causing code clones not to distribute all over a scatter plot as much as possible. The more code clones exist among two source files, the nearer the files are to be located in each axis.

In the sorting, we take a policy to concentrate a distribution of code clones in upper left corner as much as possible. The sorting is executed as follows:

Step1: Select a head file which are located on the upper left on the plot, from the target files. Make H the head file.

Step2: For the remaining files, select the most similar file to H and put it next to H.

Step3: Repeat step2 successively while any file remains, treating H as the most similar file in previous step2.

Here, we use the following two criterions to sort the files on scatter plot. For n files, $RSA(i)$ denotes the *ratio of similarity of file i to all other $n-1$ files* and $RST(i,j)$ denotes the *ratio of similarity between two files i and j*.

RSA is used to select the head file. $RSA(f)$ is the ratio of covered code range of file f by clones of all files except f and defined as the following expression:

$$RSA(f) \quad = \frac{1}{LOC(f)} \sum_{c \in CF(f)} LOC(c)$$

$$(LOC(C) : \text{the number of lines of code } C\ )$$

Here, $CF(f)$ is a set of code fragments which are included in file f and have clone relation with some code fragments in other files, and c is a element of $CF(f)$. In this summation, overlapped code portions are counted only once.

$RST(f_1, f_2)$ is the ratio of covered code range of file f_1 by clones of file f_2, and defined as the following expression:

$$RST(f_1, f_2) \quad = \frac{1}{LOC(f_1)} \sum_{c \in CF(f_1, f_2)} LOC(c)$$

$CF(f_1, f_2)$ is a set of code fragments which are included in file f_1 and have clone relation with some code fragments in file f_2, and c is a element of $CF(f_1, f_2)$. Again in this summation, overlapped code portions are counted only once.

For example, in Figure 4(b), let the position of file `f1` be the head of axis since `f1` has the highest RSA value Then, `f6` is arranged next to `f1`, because RST(`f1`, `f6`) has the highest values and so `f6` is most similar to `f1` among the five files (`f2`, `f3`, `f4`, `f5`, `f6`) whose position are not yet decided. Next, in the same way, `f3` will be arranged next to `f6` as the most similar file to `f6` among the remaining files(`f2`, `f3`, `f4`, `f5`). If these processes are repeated for the remaining files, files which are similar to each other will be located close together as shown in Figure 4(b).

3.2.3 Clone class manager(CCM)

Clone class manager is aimed to perform an analysis from several points of view. CCM classifies the code fragments which are extracted by code clone detector into clone classes, and several metrics are calculated. CCM shows the clone class information through metrics graph viewer and clone class list viewer.

Here, we use the following metrics for clone class[11].

RAD(C) (Radius of clone class):
For a given clone class C, let F be a set of files which include each code fragment of C. Define $RAD(C)$ as the maximum length of path from each file F to the lowest common ancestor directory of all files in F. If a value of $RAD(C)$ is high, since code clones in C scatter widely over the file system, it is more difficult to modify the faults which is contained in C.

LEN(C) (Length):
$LEN(C)$ for clone class C is the maximum length of token sequence for each one in C.

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

DFL(C) (Deflation by clone class):

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

$$\begin{aligned} DFL(C) &= (\text{Old number of tokens related to } C\ ) \\ &\quad -(\text{New number of tokens related to } C\ ) \\ &= (LEN(C) \times POP(C)) \\ &\quad -(LEN(C) + k \times POP(C)) \\ &= (LEN(C) - k) \times (POP(C) - 1) - k \end{aligned}$$

Metrics graph

In this graph, four kinds of metrics, $RAD(C)$, $LEN(C)$, $POP(C)$ and $DFL(C)$, are displayed for each clone class C. In order to efficiently show the values of metrics and to consider the extensibility to easily add another metrics to Gemini, we adopt the parallel coordinate plot[16]. Since a focus on the plot can be changed by operation of a user, it is suitable for interactive analysis. It is generally said that the plot is effective to extract valuable information from a lot of data. Gemini draws one one polygonal line per each clone class.

Gemini provides the user with the following functions to operate the metrics graph:

- Browsing the part of source code which corresponds to the user selected clone classes,
- **Filtering** clone classes based on the value of each metric,
- Changing color of polygonal line according to the value of each metric

By using the filtering, the user can set the warning range about each metric. The clone classes whose metric values are in the warning ranges are highlighted in the graph, and are put in *selected state*.

For example, the number of clone classes existing in Figure 5 is about 6000 in total. Then, if the user sets the warning range (in this case, the range of the value of RAD, LEN, POP and DFL were set as low, all, high and high, respectively), the range is surrounded by the polygon shown in Figure 5 and only the polygonal lines for two clone classes become emphasized. Also the corresponding code fragments on the actual source code are emphasized through source code viewer.

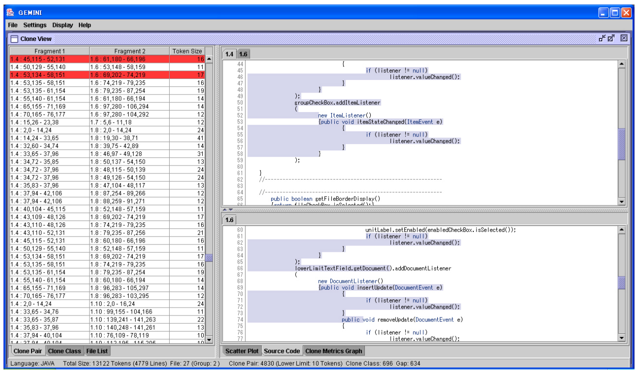


Figure 6. GUI Snap shot of source code viewer

3.2.4 Source code manager (SCM)

Source file manager gives information of location of the code clones in *selected state*, that is, specified by the user through CPM and CCM, and displays the selected code fragments through the source code viewer. Especially a code clone is highlighted and a clone pair is displayed by the pair as shown in Figure 6.

3.3 Implementation

Gemini has been implemented in Java (about 10,000 lines) and runs on the environment where JDK 1.3 VM can be executed. A example of GUI is shown in Figures 3, 5 and 6.

4 Application to the programming exercise

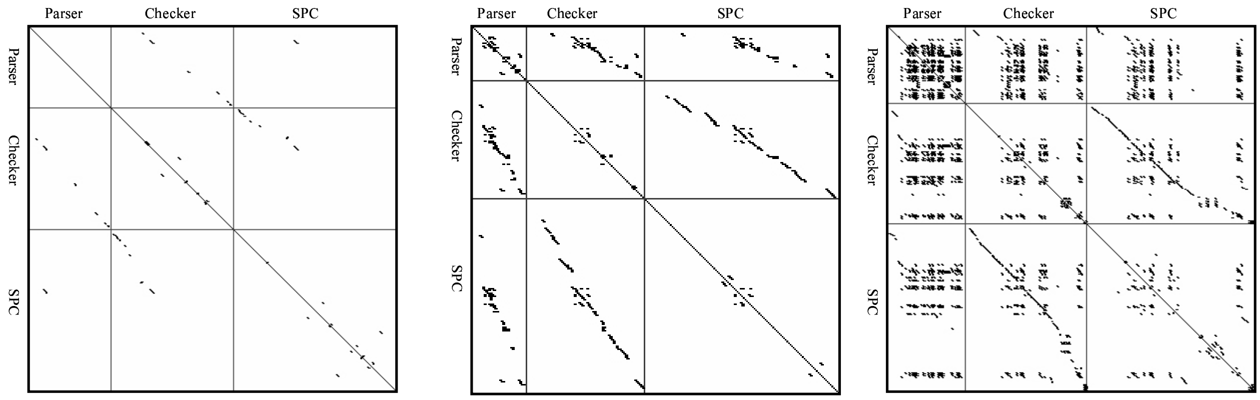
4.1 Overview

We have applied Gemini to source files of programs developed in a certain programming exercise of Osaka University. As for plagiarism of program, the paper[13] presents a detailed discussion and experiments. In our experiment, comparison of two or more versions of a developer's program is also discussed.

In the exercise, each student writes a compiler in C language, which translates a program written in the subset of Pascal language into the CASL(assembly language). Instruction documents are given to each student and the specification of a compiler is defined in the documents. At another lecture, they learn design and implementation of a compiler with a textbook, which contains sample source code of a simple compiler.

The exercise consists of three steps (sub-exercises):

Step1(Ex.1): Making a syntax checker($Parser$).



(a) Extended with low reuse (M_1)

(b) Extented with high reuse (M_2)

(c) Extended with not only high reuse but also high value of DFL (M_3)

Figure 7. Reuse of programs

Step2(Ex.2): Making a semantic checker($Checker$).

Step3(Ex.3): Making a compiler(SPC).

In addition, it was required that $Checker$ and SPC are developed by reusing the code of the previous programs. That is, $Checker$ is developed by reusing $Parser$ and SPC is developed by reusing $Checker$.

We collected source files of the programs ($Parser$, $Checker$ and SPC) from 69 students ($M_1..M_{69}$). Totally, the size of all the programs is about 360,000 lines. The minimum length of code clone was set to 30 tokens.

4.2 Analysis

In this experiment, by using Gemini, we analyzed the following issues:

(1)Reuse among three programs: In order to confirm whether each student meets the requirements of the exercise, we checked the values of $RST(Parser$, $Checker)$ and $RST(Checker$, $SPC)$ for each student. Then, using Gemini, we validate the results of RST to examines the actual code clones.

(2)Similarity among all programs: In the exercise, illegal reuse sometimes happens. That is, some students copy others' programs and modify them to meet the deadline and so on. So, we use the values of RSA and examine the actual code which has high RSA.

(3)Usefulness of metrics graph: We evaluate the usefulness of the metrics graph whether it can identify the distinctive code clones that should be arranged into some modules.

Table 1. Values of RST

	Parser → Checker	Checker → SPC	ave.
M_1	0.117	0.086	**0.102**
M_2	0.535	0.563	**0.549**
M_3	0.674	0.729	**0.701**
M_4	0.156	0.449	0.303
M_5	0.118	0.363	0.241
M_6	0.119	0.426	0.272
M_7	0.273	0.282	0.278
M_8	0.039	0.538	0.288
M_9	0.236	0.211	0.224
M_{10}	0.071	0.709	0.390
		...	
M_{69}	0.112	0.598	0.355
ave.	**0.185**	**0.461**	0.320
max.	0.674	0.747	0.701
min.	0.037	0.086	0.102

4.2.1 Reuse of programs

The values of $RST(Parser, Checker)$ and $RST(Checker, SPC)$ are shown in Table 1.

The averages of $RST(Parser$, $Checker)$ and $RST(Checker$, $SPC)$ are 0.185 and 0.461, respectively. For $RST(Parser$, $Checker)$, the values seem to be low.

For some distinctive programs, we checked the details of them. Figure 7 shows the scatter plot of them. In the plots, $Parser$, $Checker$ and SPC are arranged on their axes in the order of exercise number.

For M_1, the average value of RST was lowest (0.102) and the data is shown in Figure 7(a). It means that M_1 did not reuse $Parser$ or $Checker$ so much. We examined the actual code of M_1 by using source code viewer of Gemini and then found the small code clones which could not found on the number of minimum length of code clone, 30 tokens. So, we recalculated the values by changing the minimum length of code clone to 15 for all the students. Then, the values of $RST(Parser, Checker)$ and $RST(Checker, SPC)$ become 0.515 and 0.266, respectively. (The averages of $RST(Parser, Checker)$ and $RST(Checker, SPC)$ become 0.441 and 0.651, respectively.) The value 30 of minimum length of code clone was decided based on the results of experiment in [11], when we applied CCFinder to a huge size of programs. In this experiment, each size of the program is about 1000-2000 LOC. The results show that it should be careful to set the value of minimum length of code clone according to the characteristics of the target program.

On the other hand, for M_2 and M_3, the average value of RST were relatively high (M_2: 0.549, M_3: 0.701) and the data are shown in Figure 7(b) and (c). It means that M_2 and M_3 did highly reuse in the exercise. However, the two scatter plots have quite different appearances. The difference can be explained by the value of DFL shown in Table 3. For M_2, $DFL(Parser)$ [2], $DFL(Checker)$ and $DFL(SPC)$ are 137, 157 and 157, respectively. These values are relatively low compared to the average values. On the other hand, for M_3, $DFL(Parser)$, $DFL(Checker)$ and $DFL(SPC)$ are 473, 239 and 419, respectively. These values are relatively high to the average values. That is, for M_3, there remains some possibility to improve the programs by collecting the code clones and arranging them into some modules(See Section 4.2.3).

Table 2. Values of RSA

	Parser	Checker	SPC	ave.
M_1	0.015	0.000	0.000	0.005
M_2	0.000	0.000	0.000	0.000
M_3	0.007	0.000	0.000	0.002
M_4	0.029	0.244	**0.159**	0.144
M_5	0.162	0.271	**0.148**	0.194
M_6	0.326	0.211	**0.137**	0.224
M_7	0.297	0.244	**0.160**	0.234
M_8	0.348	0.151	**0.142**	0.214
M_9	0.028	0.009	0.011	0.016
M_{10}	0.000	0.000	0.000	0.000
≈		...		≈
M_{69}	0.032	0.004	0.003	0.013
ave.	**0.089**	**0.032**	**0.019**	0.046
max.	0.407	0.271	0.160	0.234
min.	0.000	0.000	0.000	0.000

[2]The maximum value of $DFL(C)$ in program P is abbreviated to $DFL(P)$.

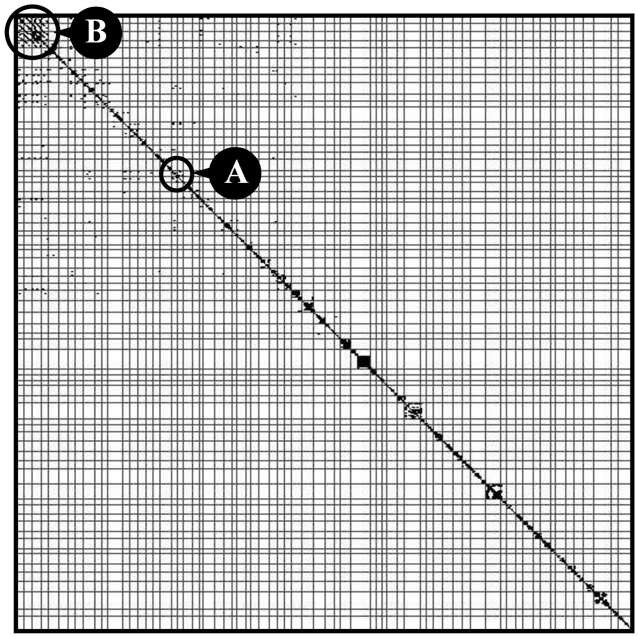


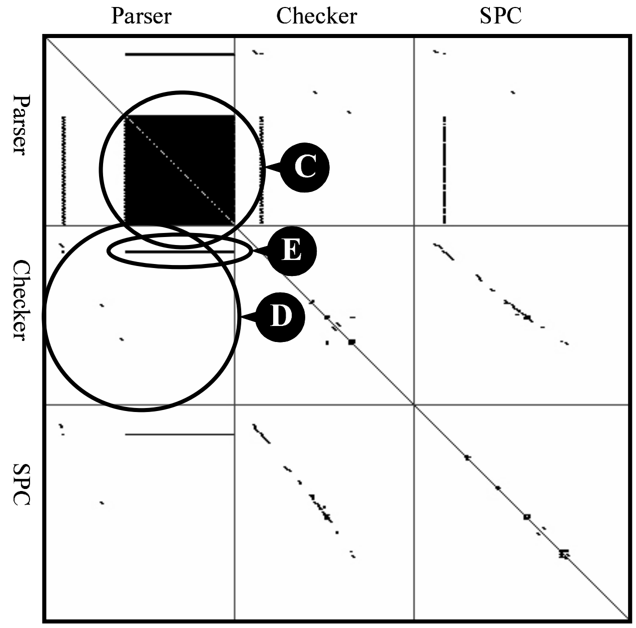
Figure 8. Sorted scatter plot of all students' SPC

4.2.2 Similarity among programs

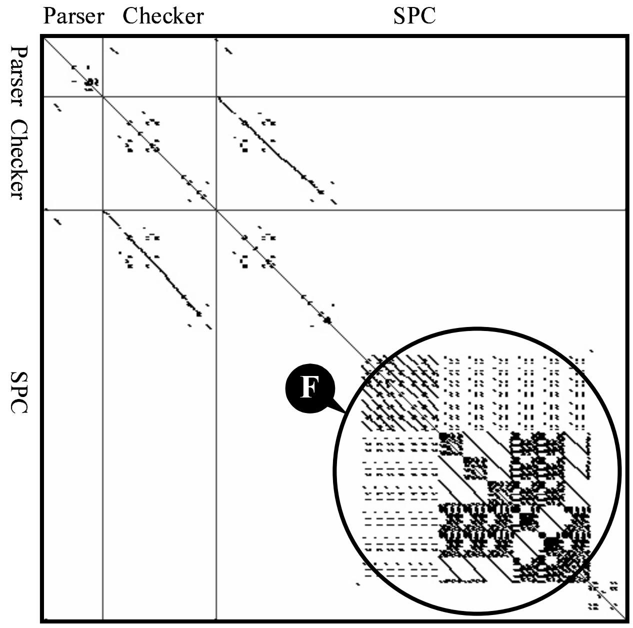
The values of RSA are shown in Table 2. The average value of RSA for $Parser$, $Checker$ and SPC are 0.089, 0.032 and 0.019, respectively. So, as the exercise is in progress, the similarity between students becomes lower. It was just as we had expected since the later exercise requires the originality of the student.

However, some students have high RSA values even for SPC. We checked their data using scatter plot. Figure 8 is the scatter plot of all the 69 students' SPC programs. The grillage in the figure shows a separator of individuals. Predictably, the distribution of code clones was spread widely all over the scatter plot in the beginning. So, we rearranged them using sorting function of Gemini. Then, the distribution concentrated into the upper left corner as Figure 8. Crowded code clones marked A in the Figure 8 are located in the area where the code clones between SPC programs of M_4 and M_5 (let M_4 and M_5 be called Group A) are shown. Crowded ones marked B are located in the area where the code clones among SPC programs of M_6, M_7 and M_8 (let M_6, M_7 and M_8 be called Group B) (they are arranged side-by-side on the axes) are shown.

Investigating the values of RSA for M_4, M_5, M_6, M_7 and M_8, their rank was top five of $RSA(SPC)$ and $RSA(Checker)$. The rank of each member of Group B was also included in top 10 of $RSA(Parser)$. Through the source code viewer, we examined the corresponding code



(a) M_9



(b) M_{10}

Figure 9. Scatter plots with high DFL

fragments in the source code. As for Group A, the most of similar code fragments among them are described in sample compiler in the textbook. However, as for Group B, some code fragments were similar even about name of variables or commentation. So, the possibility that some reference to others' programs were performed among Group B is high.

4.2.3 Usefulness of metrics graph

In this experiment, since the programs developed by each student are not so large and are located in each student's single directory, we did not use RAD. Also, since LEN and POP are in proportion to DFL according to the definition, we deal with the DFL here.

As for definition of DFL, the number of tokens for one caller statement was set as 5 tokens (a caller statement consists of "$Name\ of\ SubRoutine$", "(", "$Argument$", ")", ";"). Then, in this experiment, DFL is defined as follows:

$$DFL(C) \quad = (LEN(C) - 5) \times (POP(C) - 1) - 5$$

The maximum values of DFL in each program are shown in Table 3.

The average of $DFL(Parser)$, $DFL(Checker)$ and $DFL(SPC)$ are 196, 183 and 311, respectively. If we assume that one line has five tokens in average, 30-60 lines will be reduced by reconstruction.

Here, we examined the distinctive programs: $Parser$ by M_9 (DFL=3528) and SPC by M_{10} (DFL=3439). These values are prominent.

Firstly ,Figure 9(a) shows M_9's scatter plot. In this scatter plot, $Parser$, $Checker$ and SPC are arranged on their axes in the order of exercise number.

For M_9, $DFL(Parser)$, $DFL(Checker)$ and $DFL(SPC)$ are 3538, 163 and 189, respectively. Although the $Parser$ has a very high DFL value, the DFL values of $Checker$ and SPC are almost the same with the average. It indicates that in making $Checker$, the reconstruction of each clone classes in $Parser$ to one routine in $Checker$ was conducted. Through metrics graph and source code viewer, we examined the code fragments that correspond to the clone class having very high DFL.

All the code fragments which are in a set of crowded code clones marked C in Figure 9(a) are included in the clone class having very high DFL. In turn, in the area marked C in which self-comparison of $Parser$ are performed, similar code fragments appeared once and again and sequentially. However, in the area marked D where the comparison of $Parser$ and $Checker$ are performed, code clones like the clones marked C don't exist, and instead of them, sharp line marked E exists. That is, the sharp line indicates that M_9 reconstructed the code fragments marked C in his $Parser$ into one routine in his $Checker$.

Next, Figure 9(b) shows M_{10}'s scatter plot. For M_4, $DFL(Parser)$, $DFL(Checker)$ and $DFL(SPC)$ are 100, 211 and 3439, respectively. Crowded code clones marked F in Figure 9(b) look like ones marked C. So, we confirmed whether the clone classes included in F can be

Table 3. The maximum values of DFL in each program

	Parser	Checker	SPC
M_1	0	99	113
M_2	137	157	157
M_3	473	239	419
M_4	79	131	131
M_5	145	199	199
M_6	75	97	199
M_7	75	75	391
M_8	75	119	233
M_9	**3538**	163	189
M_{10}	100	211	**3439**
	...		
M_{69}	223	211	258
max.	3538	603	3439
min.	0	47	51
ave.	196	183	311

arranged into some modules as M_9's $Parser$, through metrics graph and source code viewer. Then, we found that the difference is only the name of constant. That is, it is easy to arrange the fragment into one module by using some parameters.

Although the above programs are typical examples that can be refined by merging code fragments of clone, there are some code clones that are not appropriate for merging. For example, the rank of DFL values of M_3's $Parser$ and SPC belong to top 20 in Table 3. We examined whether some code fragments of M_3's can be arranged to one module. Then, we found that the differences are the name of subroutine and constant. However, since the semantics of the fragments are quite different, these fragments are not appropriate to arrange into one module. It means that we should pay attention to the meanings of the code clones in arranging into one module.

5 Conclusions

In this paper, we have developed a maintenance support environment, Gemini, which supports the maintenance activity by using code clone analysis result. Using Gemini, we can specify a set of distinctive code clone through the GUI (scatter plot and metrics graph about code clones), and refer the fragments of source code corresponding to the clone on the plot or graph. Then, in order to evaluate the usefulness of Gemini, we have applied it to the programs developed in an exercise in Osaka University. We could evaluate whether the students developed the program as to meet the requirements of the exercise or illegal reuse was happened by using Gemini. Also, we could examine the several distinctive code clones through the user interface of Gemini.

We are going to evaluate the applicability of Gemini to large scale software in actual software maintenance as future research work.

References

[1] B. S. Baker, "A Program for Identifying Duplicated Code", *Proceedings of the 24th Symposium on the Interface: Computer Science and Statistics*, ACM Press, pp.18-21, 1992.

[2] B. S. Baker, "On Finding Duplication and Near-Duplication in Large Software Systems", *Proceedings of IEEE Working Conf. on Reverse Engineering*, pp.86-95, July 1995.

[3] B. S. Baker, "Parameterized Duplication in Strings: Algorithms and an Application to Software Maintenance", *SIAM Journal on Computing*, vol.26, no.5, pp.1343-1362, 1997.

[4] I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier, "Clone Detection Using Abstract Syntax Trees", *Proceedings of ICSM'98 (International Conference on Software Maintenance)*, pp.368-377, Bethesda, Maryland, Nov. 1998.

[5] S. Ducasse, M. Rieger, and S. Demeyer, "A Language Independent Approach for Detecting Duplicated Code", *Proceedings of ICSM'99 (International Conference on Software Maintenance)*, pp.109-118, Aug. 1999. Proceedings of the IEEE International Conference on Software Maintenance

[6] Duploc, `http://www.iam.unibe.ch/~rieger/duploc/`, 1999.

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

[8] D. Gusfield, *Algorithms on Strings, Trees, And Sequences*, Cambridge University Press, 1997.

[9] J. Helfman, "Dotplot Patterns: a Literal Look at Pattern Languages", *Theory and Practice of Object Systems*, vol.2, no.1, pp.31-41, 1996.

[10] J. H. Johnson, "Identifying Redundancy in Source Code using Fingerprints", *Proceedings of CASCON'93*, pp.171-183, Toronto, Ontario, 1993.

[11] 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*, (to appear).

[12] J. Mayland, C. Leblanc, and E. M. Merlo "Experiment on the Automatic Detection of Function Clones in a Software System Using Metrics", *Proceedings of ICSM'96 (International Conference on Software Maintenance)*, pp. 244-253, Monterey, California, Nov. 1996.

[13] L. Prechelt, G. Malpohl, M. Philippsen, "Finding plagiarisms among a set of programs with JPlag", *submitted to Journal of Universal Computer Science*, Nov. 2001, taken from `http://wwwipd.ira.uka.de/~prechelt/Biblio/`

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

[15] S. W. L. Yip and T. Lam, "A software maintenance survey", *Proceedings of APSEC'94*, pp. 70-79, 1994.

[16] E. J. Wegman and Q. Luo, "High Dimensional Clustering Using Parallel Coordinates and the Grand Tour", *Proceedings of 28th Symposium Interface of Computing Science and Statistics*, 1996.