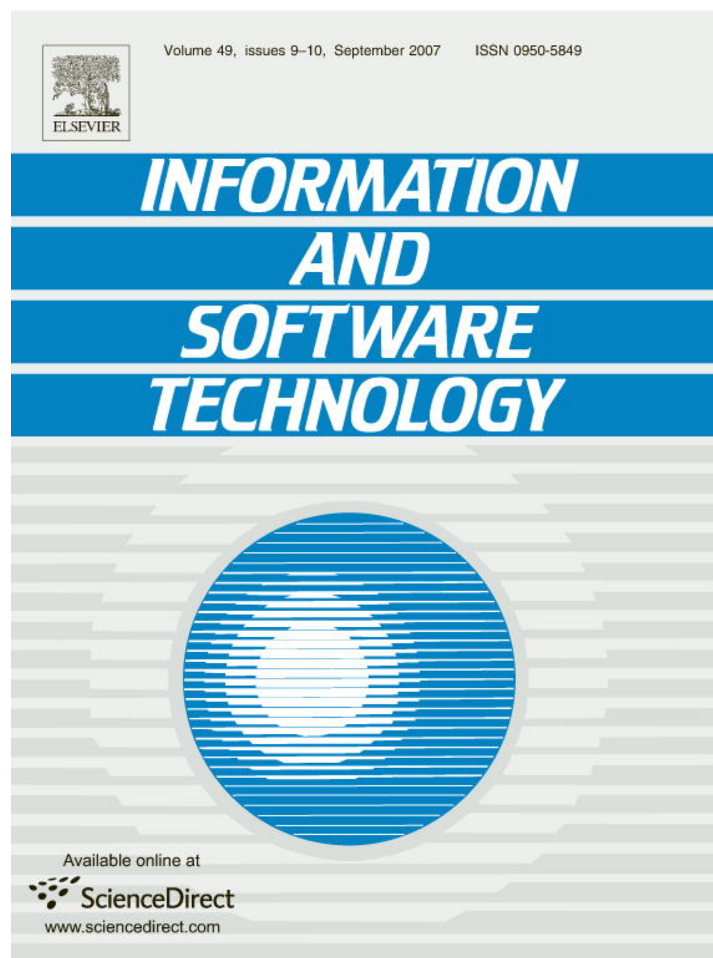
Provided for non-commercial research and educational use only.
Not for reproduction or distribution or commercial use.



This article was originally published in a journal published by Elsevier, and the attached copy is provided by Elsevier for the author's benefit and for the benefit of the author's institution, for non-commercial research and educational use including without limitation use in instruction at your institution, sending it to specific colleagues that you know, and providing a copy to your institution's administrator.

All other uses, reproduction and distribution, including without limitation commercial reprints, selling or licensing copies or access, or posting on open internet sites, your personal or institution's website or repository, are prohibited. For exceptions, permission may be sought for such use through Elsevier's permissions site at:

http://www.elsevier.com/locate/permissionusematerial

Available online at www.sciencedirect.com
ScienceDirect
ELSEVIER
Information and Software Technology 49 (2007) 985–998
INFORMATION AND SOFTWARE TECHNOLOGY
www.elsevier.com/locate/infsof

Method and implementation for investigating code clones in a software system

Yoshiki Higo [a,*], Toshihiro Kamiya [b], Shinji Kusumoto [a], Katsuro Inoue [a]

[a] Department of Software Science, Graduate School of Information Science and Technology, Osaka University, 1-3 Machikaneyama-cho, Toyonaka, Osaka 560-8531, Japan
[b] National Institute of Advanced Industrial Science and Technology, Information Technology Research Institute, Akihabara Dai Bldg. 1-18-13 Sotokanda, Chiyoda-ku, Tokyo, 101-0021, Japan

Received 11 March 2006; received in revised form 10 October 2006; accepted 12 October 2006
Available online 22 November 2006

Abstract

Maintaining software systems is becoming more difficult as the size and complexity of software increase. One factor that complicates software maintenance is the presence of code clones. A code clone is a code fragment that has identical or similar code fragments to it in the source code. Code clones are introduced for various reasons such as reusing code by 'copy and paste'. If modifying a code clone with many similar code fragments, we must consider whether to modify each of them. Especially for large-scale software, such a process is very complicated and expensive. In this paper, we propose methods of visualizing and featuring code clones to support their understanding in large-scale software. The methods have been implemented as a tool called Gemini, which has applied to an open source software system. Application results show the usefulness and capability of our system.
© 2006 Elsevier B.V. All rights reserved.

Keywords: Software maintenance; Code clone; Software understanding; Software metrics

Author's personal copy

1. Introduction

Software maintenance is a continuous process of changing a software system after the system has been delivered to the field. As the size and complexity of software increase, maintenance tasks become more difficult and burdensome. For example, Yip and Lam reported that 66% of the cost of the software life cycle is spent on maintenance phase [23].

A code clone is defined as a code fragment that has identical or similar fragments to it in the source code. The presence of code clone is one factor that complicates software maintenance. Fowler postulates in [9] that the most frequent cause of refactoring is the presence of code clones. They are introduced in the source program for such reasons as reusing code by 'copy-and-paste'. When modifying a code clone with many other similar code fragments that reside in the system, we must determine whether to modify each of them. For large-scale software, in particular, such a process is very complicated and costly. Therefore, a powerful code clone detection tool is essential to effectively maintain software.

So far, much research has been done on the automatic detection of code clones [3,4,6,8,13,14]. We have also developed a code clone detection tool, CCFinder [11], designed to detect code clones effectively in a large-scale software system used in the industrial world, and it is still being improved day by day. CCFinder has been applied to dozens of software systems from which we have found many code clones. Today CCFinder is used in more than 100 software organizations all over the world, and we have received significant feedback from them. Since CCFinder often identifies an enormous amount of code clones, it is necessary to provide some guidelines for selecting only the important code clones from the raw output of the tool.

* Corresponding author. Tel.: +816 6850 6571; fax: +816 6850 6574.
E-mail addresses: y-higo@ist.osaka-u.ac.jp (Y. Higo), t-kamiya@aist.go.jp (T. Kamiya), kusumoto@ist.osaka-u.ac.jp (S. Kusumoto), inoue@ist.osaka-u.ac.jp (K. Inoue).

0950-5849/$ - see front matter © 2006 Elsevier B.V. All rights reserved.
doi:10.1016/j.infsof.2006.10.005

Author's personal copy

In this paper, we propose visualization and characterization methods for code clones. By using these methods, users can see how code clones are distributed in the system at a glance or obtain the code clones that have the features they are interested in. Also, based on our previous experience, we determined that there are many uninteresting code clones in the results of code clone detection. An uninteresting code clone is a code clone deemed insignificant in the context of software maintenance. We propose a filtering method that skips such code clones and reduces investigation effort. A filtering method makes the visualization and characterization methods more effective. A system named Gemini has been implemented based on the proposed methods. Case study results demonstrate the usefulness and applicability of Gemini.

In Section 2, we describe a code clone and subsequently introduce CCFinder, the code clone detection tool. Section 3 describes the proposed methods and the tool, Gemini. Section 4 illustrates the evaluation of Gemini. Section 5 introduces several other methods related to code clone visualization and code clone detection. Finally, we conclude our discussion with future works in Section 6.

2. Code Clone

2.1. Definition

A **code clone**, in general, means a code fragment that has identical or similar code fragments to it in the source code. However, there is no single or generic definition for a code clone. So far, several methods of code clone detection have been proposed, and each has its own definition of code clone. Some of these methods are described in Section 5. In the remaining parts of this paper, we apply the code clone detection feature of CCFinder as the definition of code clone.

2.2. CCFinder

In CCFinder [11], a clone relation is defined as an equivalence relation (reflexive, transitive, and symmetric relations) on fragments. A **fragment**, which is a part of a source file, can be represented using *ID*, $Line_{start}$, $Column_{start}$, $Line_{end}$, and $Column_{end}$. For fragment *f*, *ID*(*f*) is the numeral ID of the source file where *f* resided. CCFinder assigns a unique ID to all target source files. $Line_{start}(f)$ ($Line_{end}(f)$) is the line number of the start (end) of *f*, and $Column_{start}(f)$ ($Column_{end}(f)$) is the column number of the start (end) of *f*. In this definition, some fragments may partially overlap. A clone relation exists between two fragments if and only if the token sequences included in them are identical[1]. For a given clone relation, a pair of fragments is called a **clone pair** if the clone relation holds between them. An equivalence set of a clone relation is called a **clone set**. That is, a clone set is a maximal set of fragments where a clone relation exits between any pair of them.

[1] The sequences are the transformed ones as described below.

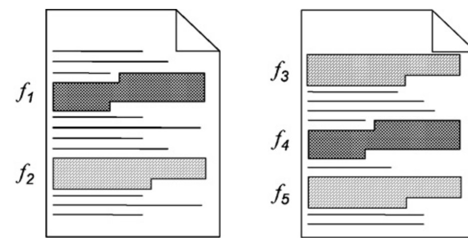


Fig. 1. Clone pair and clone set.

Fig. 1 illustrates an example of a clone relation. As shown here, five fragments have clone relations with other fragments. Fragment f_1 has a clone relation with fragment f_4, and fragments f_2, f_3, and f_5 have clone relations with each other. In this case,four clone pairs, (f_1, f_4), (f_2, f_3), (f_2, f_5), (f_3, f_5), and two clone sets, $\{f_1, f_4\}$, $\{f_2, f_3, f_5\}$ exist.

CCFinder detects code clones from source programs and reveals the locations of clone pairs in source programs. The minimum code clone length to be detected is set by users in advance. The clone detection of CCFinder is a process in which source files are input and clone pairs are output. The process consists of the following steps. (Fig. 2 shows how the input source code is treated in each step):

(1) *Lexical analysis:* Each line of every source file is divided into tokens corresponding to the lexical rules of the programming language. The tokens of all source files are concatenated into a single token sequence (Fig. 2(b)).
(2) *Transformation:* The token sequence is transformed, for example, tokens are added, removed, or changed based on the transformation rules that aim at the regularization of identifiers and the identification of structures (Fig. 2(c)).
(3) *Match detection:* From all the substrings on the transformed token sequence, equivalent pairs are detected as clone pairs (Fig. 2(d)).
(4) *Formatting:* Each location of detected clone pairs is converted into line and column numbers on the original source files (Fig. 2(e)).

3. Visualization and characterization methods of code clone for comprehension

3.1. Motivation

Since we have delivered CCFinder to more than 100 software organizations, its usefulness in actual software maintenance can be evaluated. We have received significant feedback from these organizations, including that CCFinder extracts too many code clones. Hence it is necessary to provide guidelines to select important code clones from the tool's output. Moreover, it is quite difficult for users to

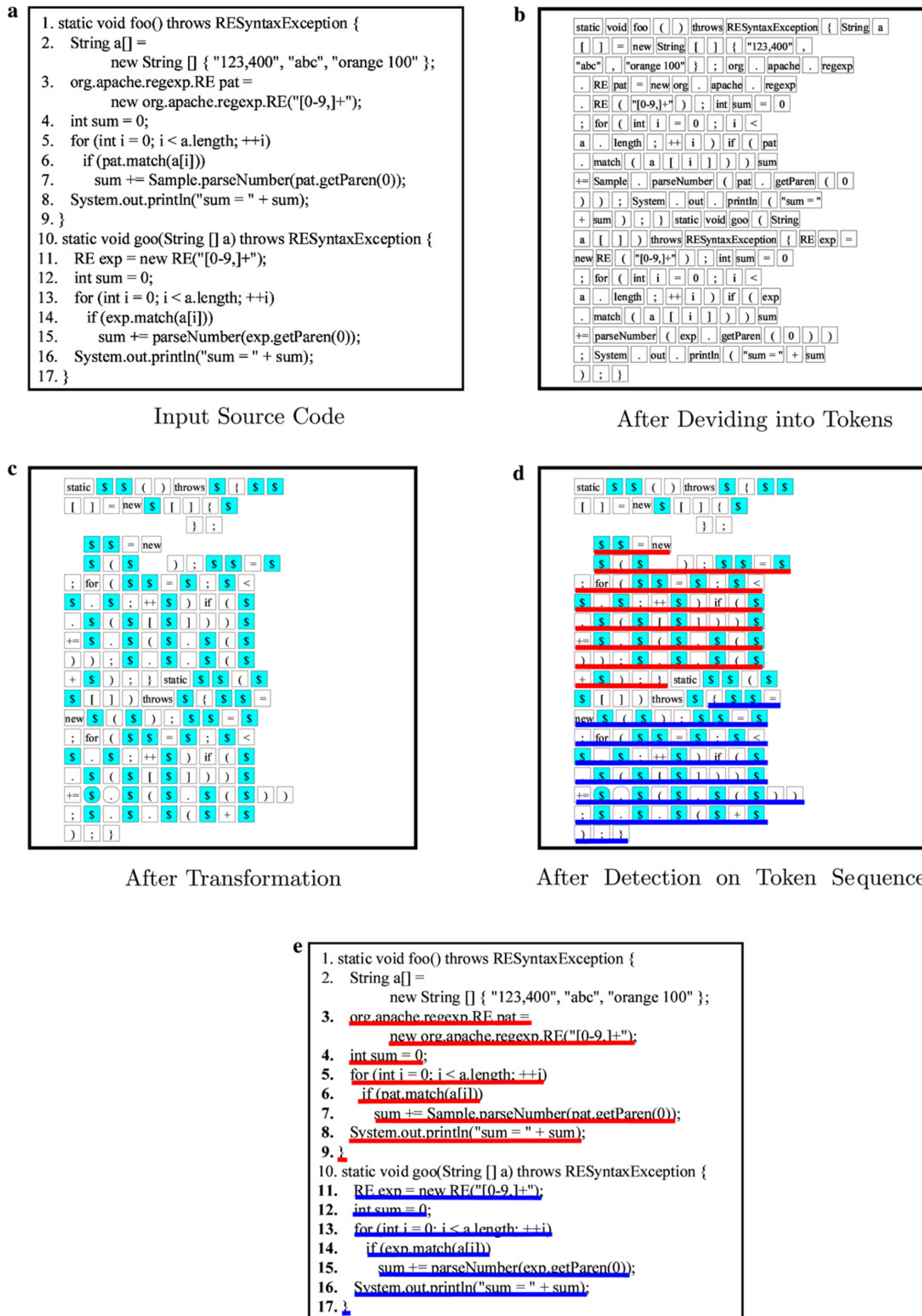


Fig. 2. Detection process of CCFinder.

investigate code clones only using the output of CCFinder. Fig. 3 shows an example of the output. In the figure, each line between `#begin{clone}` and `#end{clone}` identifies a clone pair. For example, the fragment starting at line 73 to 86 in the source file (0.2) and the fragment starting at line 124 to 137 in the source file (1.2) contribute to a clone pair. Depending on the size and nature of the target program, the amount of code clones detected by CCFinder sometimes becomes quite huge. For example, in JDK 1.5, there are approximately 2,500,000 clone pairs (12,000 clone

```
#version: ccfinder 7.2.4
#format: pairwise
#langspec: C
#option: -b 30
#option: -e char
#option: -k 30
#option: -r abdfikmnpstuv
#option: -c wfg
#begin{file description}
0.0    249    697     C:¥experiment¥linux-2.6.5¥fs¥autofs¥dirhash.c
0.1    52     93      C:¥experiment¥linux-2.6.5¥fs¥autofs¥init.c
0.2    249    622     C:¥experiment¥linux-2.6.5¥fs¥autofs¥inode.c
0.3    557    1591    C:¥experiment¥linux-2.6.5¥fs¥autofs¥root.c
0.4    30     58      C:¥experiment¥linux-2.6.5¥fs¥autofs¥symlink.c
0.5    205    488     C:¥experiment¥linux-2.6.5¥fs¥autofs¥waitq.c
1.0    272    656     C:¥experiment¥linux-2.6.5¥fs¥autofs4¥expire.c
1.1    42     63      C:¥experiment¥linux-2.6.5¥fs¥autofs4¥init.c
1.2    315    774     C:¥experiment¥linux-2.6.5¥fs¥autofs4¥inode
                :
                :
#end{file description}
#begin{clone}
0.2    73,2,116     86,18,165    47    1.2      124,2,241     137,18,290    47
0.2    152,2,385    165,6,420    34    1.2      221,2,547     237,6,582     34
0.2    171,2,432    201,2,521    81    1.2      248,2,598     282,2,687     81
0.2    80,3,140     88,5,173     32    2.13     193,3,397     201,6,430     32
0.2    75,2,118     88,5,173     52    4.6      702,2,1555    715,6,1610    52
0.2    75,2,118     85,15,161    41    13.10    317,2,659     327,19,702    41
0.2    75,2,118     85,15,161    41    14.11    609,2,1273    619,19,1316   41
0.2    75,2,118     83,47,153    33    15.4     353,2,780     361,44,815    33
                :
                :
#end{clone}
```

Fig. 3. Example of output from CCFinder.

sets) whose fragment length is more than 30 tokens. It is true that CCFinder enables users to obtain code clones quickly from such large-scale software, but it is unrealistic to check all of the detected code clones by hand to determine useful information. A simple browsing tool, which displays the source code of clone pairs one by one, is not helpful for large-scale software.

3.2. Proposal techniques

From many our previous experiences with CCFinder, we observed that CCFinder detects many uninteresting code clones. As such, a filtering method is proposed to counteract this problem. To realize an effective code clone analysis for large software products, we need some kind of bird's eye view of code clones to grasp the amount of code clones in the system, especially in the initial phase of analysis. Also, we provide an appropriate characterization of entities (code clones, target source files, and functionalities) that enables users to select arbitrary entities based on their features.

In this section, we explain our methods with an example to give a complete picture of them. Here, we assume that we detect code clones from four source files (F_1, F_2, F_3, F_4) located in two directories (D_1, D_2). Each source file consists of the following five tokens (the meaning of superscript * will be described in Section 3.2.1).

F_1: $a\ b\ c\ a\ b$,
F_2: $c\ c^*\ c^*\ a\ b$,
F_3: $d\ e\ f\ a\ b$,
F_4: $c\ c^*\ d\ e\ f$

Also, we use label $C(F_i,j,k)$ to represent a fragment. Fragment $C(F_i,j,k)$ starts at the jth token and ends at the kth token in source file F_i (j must be less than k).

Here, we assume that at least two tokens are needed to be identified as a code clone. With this assumption, the following three clone sets are detected from the source files.

$$S_1 : \{C(F_1,1,2), C(F_1,4,5), C(F_2,4,5), C(F_3,4,5)\}$$
$$S_2 : \{C(F_2,1,2), C(F_2,2,3), C(F_4,1,2)\}$$
$$S_3 : \{C(F_3,1,3), C(F_4,3,5)\}$$

3.2.1. Filtering out uninteresting code clones

Many uninteresting code clones are included in the CCFinder detection results. In this paper, an "uninteresting code clone" is a code clone whose existence information is useless when using code clone information in software development or maintenance.

We have identified two types of uninteresting code clones. The first is a language-dependent code clone. When a specific programming language is used, a programmer has to repeatedly write some code fragments that cannot be merged into code fragments due to language limitations. A language-dependent code clone consists of such code fragments. The second is an application-dependent code clone. Some application frameworks sometimes require idiomatic code fragments to the application code to interface with the frameworks. For example, a code fragment of database connection is a typical application-dependent code clone. Application-dependent code clone is a code clone that consists of such code fragments. Language-dependent code clones are detected from all software systems written in the same programming language, but not application-dependent code clones, which differ greatly among systems. Therefore, it is much more difficult to filter out application-dependent code clones than language-dependent ones. The presence of uninteresting code clones does not negatively influence software development or maintenance. They are stereotyped code and are very stable.

As the first step to filter out uninteresting code clones, we propose a method to filter out language-dependent code clones. For example, consecutive variable declarations, consecutive method invocations, and case entries of switch statements, which become code clones due to the structure of the programming language, are typical language-dependent code clones.

Fig. 4 is an example of a language-dependent code clone. The highlighted parts are a clone pair between files A and B. Each code fragment of the clone pair is an implementation of consecutive method invocations. The variable and method names in the code fragments are different. As described in Section 2.2, CCFinder transforms user-defined names into the same special token. This transform detects the same logic code with different names; for example, after copy and paste some variable names are changed. Unfortunately, CCFinder also detects many language-dependent code clones such as Fig. 4.

We focused on a repetition structure within a code clone because a language-dependent code clone has repetitive implementations of the same logics. We propose to filter

Author's personal copy

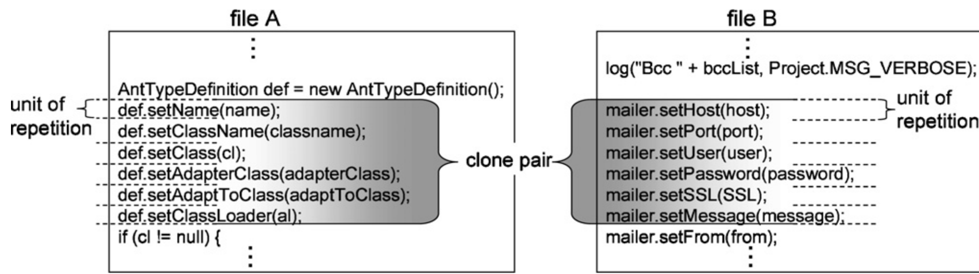


Fig. 4. Example of language-dependent code clone.

out such code clones with a metric called **RNR(S)** that represents the ratio of *non-repeated code sequence* in clone set S.

Here, we assume that clone set S includes n fragments, $f_1, f_2, \cdots, f_n$. $LOS_{whole}(f_i)$ represents the Length Of *whole* Sequence of fragment f_i, and $LOS_{repeated}(f_i)$ represents the Length Of *repeated* Sequence of fragment f_i, then,

$$RNR(S) = 1 - \frac{\sum_{i=1}^{n} LOS_{repeated}(f_i)}{\sum_{i=1}^{n} LOS_{whole}(f_i)}$$

We defined *repeated code sequence* as repetitions of its adjacent code sequence and *non-repeated code sequence* as the other parts. In the example, three tokens are considered *repeated code sequences*, and the superscript * indicates that its token is in a *repeated code sequence*.

In this case,

$$RNR(S_1) = 1 - \frac{0+0+0+0}{2+2+2+2} = \frac{8}{8} = 1.0$$

$$RNR(S_2) = 1 - \frac{1+2+1}{2+2+2} = \frac{2}{6} = 0.\dot{3}$$

$$RNR(S_3) = 1 - \frac{0+0}{3+3} = \frac{6}{6} = 1.0$$

This metric enables users to identify clone sets such as consecutive variable declarations or consecutive accessor declarations in Java or repeated `printf`, `scanf`, and `switch` statements clones in C language. From our experience, clone sets whose *RNR* values are less than '0.5' are deemed uninteresting.

3.2.2. Scatter Plot of code clone

We utilized and enhanced Scatter Plot [2,8,15] for a bird's eye view visualization method of code clones. Fig. 5 shows the Scatter Plot of the example explained previously. Both the vertical and horizontal axes represent tokens of source files that are sorted alphabetically by file path, so the source files in the same directory are close to each other. A clone pair is shown as a diagonal line segment, assuming that a cloned fragment has at least two tokens. Each dot on diagonal line segments means the corresponding tokens on the horizontal and vertical axes are identical. The dots are spread symmetrically with a diagonal line from the upper left corner to the bottom right.

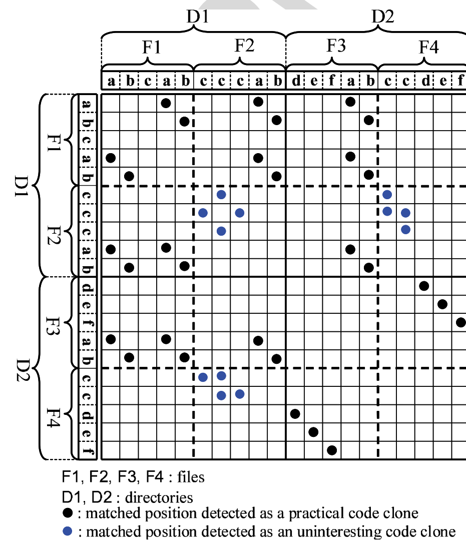


Fig. 5. Example of *Scatter Plot*.

Using Scatter Plot, the distribution state of code clones can be grasped at a glance. Also, our Scatter Plot shows filtering results with metric *RNR*. Each blue dot represents an element judged an uninteresting code clone. By using the filtering results, users can avoid spending too much time on uninteresting code clones, which means code clone analysis can be done more effectively by using Scatter Plot.

The directory (a package in the case of Java) separators are drawn as a solid lines, to distinguish them from file separators shown as dotted lines. Users can recognize the boundaries of directories and understand which directories (packages) contain many code clones and which directories shares many code clones with other directories.

3.2.3. Clone set metrics

Here, we elaborate how to quantitatively characterize code clones and how to visualize them. We defined the following metrics to characterize code clones.

- **LEN(S)**: $LEN(S)$ is the average length of sequences (size of fragments) in clone set S. In the example, the values of $LEN(S_1)$, $LEN(S_2)$, and $LEN(S_3)$ are 2, 2, and 3, respectively. Using metric LEN, the fragment size of clone set S_3 is greater than the ones of clone sets S_1 and S_2.

- **POP(S)**: $POP(S)$ is the number of S fragments. A high $POP(S)$ value means that S fragments appear in many places in the system. In the example, the values of $POP(S_1)$, $POP(S_2)$, and $POP(S_3)$ are 4, 3, and 2, respectively. Using metric POP, the number of occurrences of clone sets S_1 is larger than S_2 and S_3.
- **NIF(S)**: $NIF(S)$ is the number of source files that include any S fragments. A high $NIF(S)$ value may indicate a badly designed software system, the absence of abstraction for S fragments, or spread code fragments of a crosscutting concern. In the example, the values of $NIF(S_1)$, $NIF(S_2)$, and $NIF(S_3)$ are 3, 2, and 2, respectively. Using metric NIF, clone set S_1 involves more source files than S_2 and S_3.
- **RNR(S)**: $RNR(S)$ is described in Section 3.2.1. As mentioned there, using metric RNR, we can see whether each clone set is practical or uninteresting. From our experience, '0.5' is deemed appropriate as a threshold value of RNR. In the example, clone set S_2 is judged uninteresting.

Using these simple metrics, we can see which clone sets are discriminative in various aspects.

We propose a visualization/selection method using a **Metric Graph** for characterized code clones. Here, we explain Metric Graph using Fig. 6. In Metric Graph, each metric has a parallel coordinate axis. Users can specify the upper and lower limits of each metric. The hatching part shows the range bounded by their upper and lower limits. A polygonal line is drawn per clone set. In this figure, three lines of clone sets S_1, S_2, and S_3 are drawn. In the left graph (Fig. 6(a)), all metric values of all clone sets are in the hatching part. As such, all clone sets are in the *selected* state. In the right graph (Fig. 6(b)), the values of $LEN(S_1)$ and $LEN(S_2)$ are smaller than the under limit of LEN, which places S_1 and S_2 in an *unselected* state. This means that we can get "long" code clones by changing the lower limit of LEN. Thus Metric Graph enables users to choose arbitrary clone sets based on metric values.

3.2.4. File metrics

We also defined the following metrics to characterize source files. All metrics only use clone sets whose RNR have thresholds of th or greater for calculation. Here, we use '0.5' as the threshold. These metrics are used for the filtering of source files, which will be described below in Section 3.3.

- **NOC$_{th}$(F)**: $NOC_{th}(F)$ is the number of fragments of any clone sets in source file F whose RNR values are equal to or greater than threshold th. In the example, the values of $NOC_{0.5}(F_1)$ and $NOC_{0.5}(F_3)$ are 2, and those of $NOC_{0.5}(F_2)$ and $NOC_{0.5}(F_4)$ are 1. Here, by using metric NOC, source files F_1 and F_3 have more duplicated fragments than source files F_2 and F_4.
- **ROC$_{th}$(F)**: $ROC_{th}(F)$ is the ratio of the duplication of F. In the example, the values of $NOC_{0.5}(F_1)$, $NOC_{0.5}(F_2)$, $NOC_{0.5}(F_3)$, and $NOC_{0.5}(F_4)$ are 0.8, 0.4, 1.0 and 0.6, respectively. Here, by using metric ROC, source file F_3 is completely duplicated.
- **NOF$_{th}$(F)**: $NOF_{th}(F)$ is the number of source files that share any code clones with F. In the example, the values of $NOF_{0.5}(F_1)$, $NOF_{0.5}(F_2)$, $NOF_{0.5}(F_3)$, and $NOF_{0.5}(F_4)$ are 2, 2, 3, and 1 respectively. Here, by using metric NOF, source file F_3 shares code clones with all the other source files.

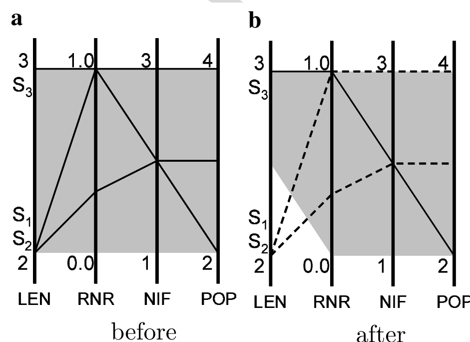


Fig. 6. Filtering clone sets using the Metric Graph.

3.3. Code clone visualization tool: Gemini

We implemented a code clone visualization tool named **Gemini** based on the proposed visualization and characterization strategies. Gemini supports all programming languages that CCFinder can handle and provides the following views as implementations of the code clone visualization mechanism and code clone/source file selection mechanisms.

- Scatter Plot
- Metric Graph
- File List

As described in Section 3.2.2, by using Scatter Plot, users can understand the distribution state of code clones at a glance, which is very useful, especially in the early stage of analysis.

Metric Graph is designed to enable users to quantitatively select clone sets. In Scatter Plot, the degree to which clone sets are distinguished depends on their positions. For example, suppose clone set $S_{example}$ has 100 fragments. If all fragments of $S_{example}$ are in the same source file, they will be distinguished because the positions of each line segment are close to each other. But if they are in different source files, their line segments are scattered, and so they cannot be distinguished. On the other hand, in Metric Graph, the '100 fragments' feature is represented as metric $POP(S_{example})$, and users can select $S_{example}$ regardless of their positions.

File List is used to select source files. File List exhibits all source files of the system with quantitative information, the file metrics described in Section 3.2.4 and two size metrics.

thors personal copy

File List

FILE ID	LOC(f)	TOC(f)	NOC(f)	ROC(f)(%)	NOF(f)	File Name
FILE 0.0	137	121	2(2)	82(82)	1(1)	ZipShort.java
FILE 0.1	203	408	2(2)	39(39)	1(1)	ExtraFieldUtils.java
FILE 0.2	113	5	0(0)	0(0)	0(0)	UnixStat.java
FILE 0.3	135	124	0(0)	0(0)	0(0)	UnrecognizedExtraField.java
FILE 0.4	496	861	10(12)	23(23)	14(17)	ZipEntry.java
FILE 0.5	119	6	0(0)	0(0)	0(0)	ZipExtraField.java
FILE 0.6	557	1018	3(7)	6(12)	1(1)	ZipFile.java
FILE 0.7	141	160	3(3)	83(83)	2(2)	ZipLong.java
FILE 0.8	871	1686	8(17)	24(29)	2(2)	ZipOutputStream.java
FILE 0.9	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
FILE 0.10	442	668	4(4)	22(22)	1(1)	TarBuffer.java
FILE 0.11	191	5	0(0)	0(0)	0(0)	TarConstants.java
FILE 0.12	682	993	2(5)	3(13)	4(6)	TarEntry.java
FILE 0.13	417	628	1(1)	6(6)	1(1)	TarInputStream.java
FILE 0.14	348	462	1(1)	8(8)	1(1)	TarOutputStream.java
FILE 0.15	233	317	0(0)	0(0)	0(0)	TarUtils.java
FILE 0.16	81	22	0(0)	0(0)	0(0)	ErrorInQuitException.java
FILE 0.17	562	969	7(9)	15(19)	15(15)	MailMessage.java
FILE 0.18	131	157	0(0)	0(0)	0(0)	SmtpResponseReader.java
FILE 0.19	136	5	0(0)	0(0)	0(0)	BZip2Constants.java
FILE 0.20	873	2069	24(25)	38(39)	2(2)	CBZip2InputStream.java
FILE 0.21	1670	3940	29(47)	21(26)	2(2)	CBZip2OutputStream.java
FILE 0.22	167	81	0(0)	0(0)	0(0)	CRC.java
FILE 0.23	277	295	2(8)	15(39)	1(1)	AnsiColorLogger.java
FILE 0.24	339	716	3(4)	23(26)	1(1)	CommonsLoggingListener.java

Z:¥apache-ant-1.6.0¥src¥main¥org¥apache¥tools¥zip¥ZipLong.java

Fig. 7. Snapshot of the File List.

Users can sort all source files based on any metrics. Fig. 7 shows a snapshot of File List. In the figure, the two size metrics, $LOC(F)$ and $TOC(F)$, represent the number of lines and tokens in source file F respectively. For metrics NOC, ROC, and NOF, two values are shown. The values outside parentheses are the metrics on threshold th, and those in parentheses are the metrics on threshold 0, which means that all code clones are counted. File List is very useful when users want to select source files based on quantitative information. We did not apply a quantitative file selection mechanism using Metric Graph for File List because it is more useful to use not only quantitative metrics but also their paths or names. As described in Section 3.2.3, Metric Graph is suitable for selections based on numeric values, but not for strings such as file paths or names. This explains why we use File List not Metric Graph. Also, File List has a function that sorts source files in ascending or descending order of their metrics or in alphabetical order of their file names.

4. Case study

4.1. Target and configurations

We chose Ant [1](version 1.6.0) as the target. Ant 1.6.0 includes 627 source files, and its size is approximately 180,000 LOC. In this case study, we set 30 tokens as the minimum token length of a code clone (intuitively, 30 tokens correspond to about 5 LOC). The value '30' comes from our previous studies of CCFinder [11].

It took less than a minute to detect code clones with CCFinder. As the results of code clone detection, we found 2406 clone sets and 190,004 clone pairs. The results show that it is unrealistic to check all detected code clones due to the enormous amount, and it is also very important to select discriminative code clones or source files. In this study, we set 0.5 as the threshold of metric RNR. If $RNR(S)$ is less than 0.5, more than half of the tokens in clone set S are in repeated token sequences. The details of the filtering results are written in Section 4.2

4.2. Filtering with RNR

We browsed through the source code of all code clones judged uninteresting by using RNR. Table 1 shows the breakdown of clone sets whose RNR are less than 0.5. The number of such clone sets is 1073, and all of them are the consecution of simple implementations. As described in Section 2.2, CCFinder detects code clones after translating all user-defined names into the same special token, and so each code fragment included in the same clone set is an implementation of different contents, as in Fig. 4.

Many consecutive accessor declarations are coincidentally found as code clones; however, the user-defined names in them are different. We identify them as code clones because both the accessor and field names are transformed into the same token before detection.

Consecutive simple method declarations are coincidentally found as code clones just as in the case of consecutive accessor declarations. Fig. 8(a) shows one such code clones that implements simple instructions, but not accessors.

Consecutive method invocations are detected as code clones. Fig. 8(b) shows one such code clone. It is pointless to display it to users in code clone analysis because they can do nothing about it.

Consecutive if and if-else statements are detected as code clones. Fig. 8(c) shows one such code clone. These code clones implement verifications of variable states. Since these code clones are obviously harmless, it is pointless to show them in code clone analysis.

Consecutive case entries are coincidentally found as code clones, as in the case of consecutive accessor declarations. Fig. 8(d) shows one such code clone. The programmer usually implements simple instructions in case entries. Moreover, CCFinder replaces all user-defined names into the same special token. Thus consecutive case entries tend to be detected as code clones, but they are harmless.

Table 1
Breakdown of uninteresting code clones

Kinds of code clones	Number of clone sets
Consecutive accessor declarations	428
Consecutive simple method declarations	224
Consecutive method invocations	177
Consecutive if or if-else statements	160
Consecutive case entries	30
Consecutive variable declarations	29
Consecutive assign statements	19
Consecutive catch statements	4
Consecutive while statements	2
Total	1073

Authors personal copy

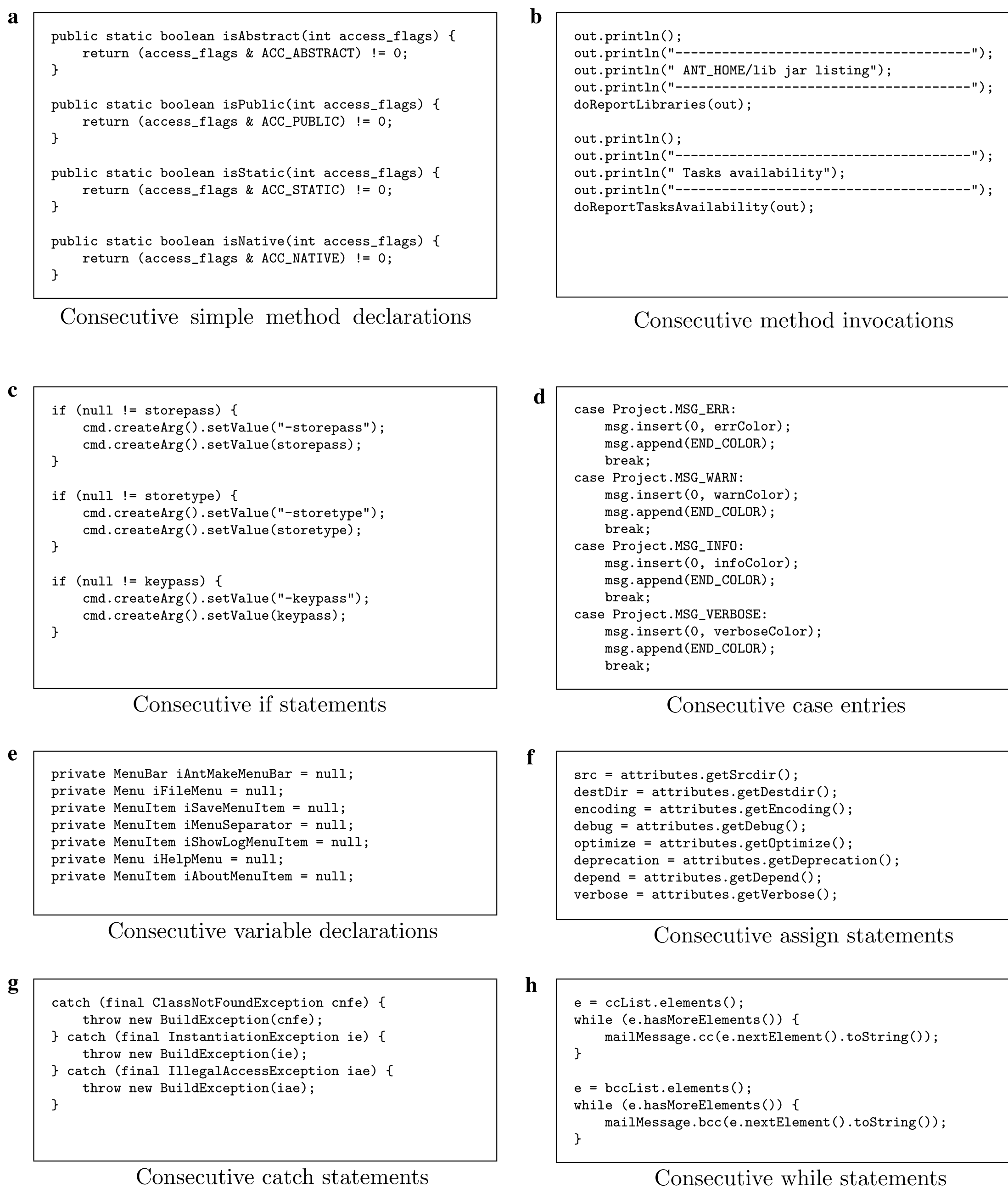


Fig. 8. Examples of uninteresting code clones.

Auth

Consecutive variable declarations and assign statements are coincidentally found as code clones, as in the case of consecutive accessor declarations. Figs. 8(e) and (f) show one such code clones. These coincidences reflect the detection algorithm of CCFinder, and they should not be detected as code clones.

Consecutive catch statements are detected as duplicated fragments. Fig. 8(g) shows one such code clone. Their existence reflects the specifications of Java and should not be detected as code clones.

Consecutive while statements are detected as duplicated fragments. Fig. 8(h) shows one such code clone. In this case, the logics of each while statement are very simple, and it is no problem to filter out them. But if their logics are complex, they should not be filtered out.

We were able to filter out 44% (1073 of 2403) of the clone sets by using *RNR*. All of the clone sets filtered out were either coincidental ones, inevitable duplications by Java specifications, or consecutive simple instructions.

4.3. Scatter Plot analysis

Fig. 9 shows snapshots of Ant's Scatter Plot. In Fig. 9, clone sets whose *RNR* are less than 0.5 are drawn in blue, and the others are drawn in black. Each vertical or horizontal line is the border between files or directories. In

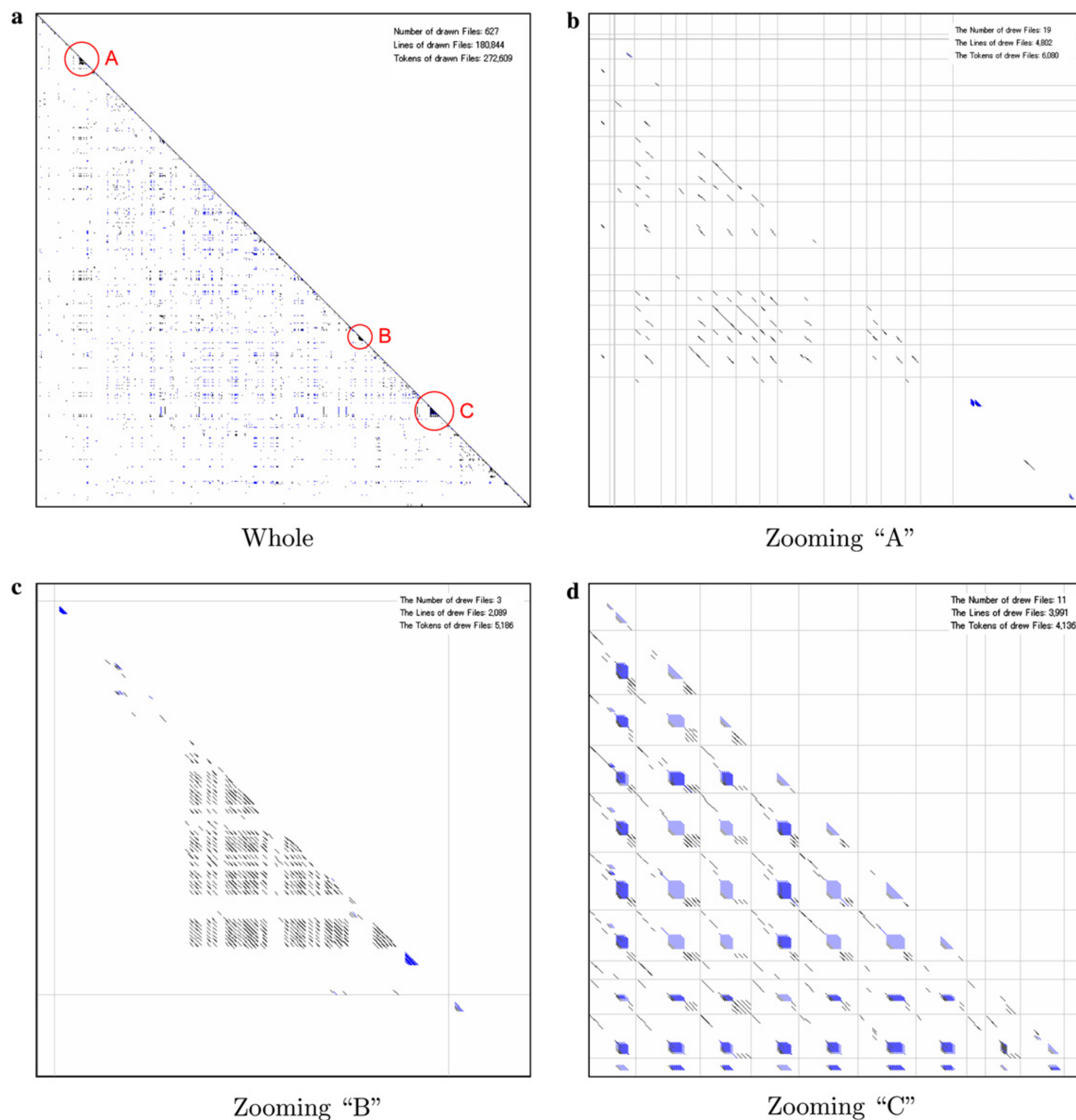


Fig. 9. Snapshots of Scatter Plot.

Fig. 9(a), all such lines are omitted because there are too many. We can see the distribution state of code clones over the system by using Scatter Plot at a glance. The distinct parts in Scatter Plot are the parts where there are many code clones in the system. Discovering whether the duplication of such parts in the system is the expected is one significant use of code clone information. We investigated what kinds of implementations were conducted in the distinct parts of the Scatter Plot. Fig. 9(a) is the entire Scatter Plot. The following describes “A,” “B,” and “C,” the three different parts marked in Fig. 9(a).

Fig. 9(b) represents part “A” in Fig. 9(a). Here are the source files under directory ant/filters/. These source files implement classes that return a `java.io.Reader` object under such various conditions as the following.

- ConcatFilter.java: Concatenate a file before and/or after the file.
- HeadFilter.java: Read the first *n*-lines of a stream.
- LineContains.java: Filter out all lines that don’t include all the user-specified strings.
- PrefixLines.java: Attach a prefix to every line.

These source files share the following functionalities.

1. Reads a character from a specified stream. If it has reached the end of stream, then some operations are performed.
2. Creates a new Reader object and returns it.

The details of each functionality in these source files were different, but the processing flows were duplicated.

Fig. 9(c) represents part “B” in Fig. 9(a). It shows that source file ant/taskdefs/optional/ide/VAJAntToolGUI.java contains many code clones. This source file implements a simple GUI for providing build information to Ant or

```
if (e.getSource() == VAJAntToolGUI.this.getBuildButton()) {
    executeTarget();
}
if (e.getSource() == VAJAntToolGUI.this.getStopButton()) {
    getBuildInfo().cancelBuild();
}
if (e.getSource() == VAJAntToolGUI.this.getReloadButton()) {
```

Fig. 10. Example of code clones in "B" (branching using source of events).

```
private Panel getAboutCommandPanel() {
    if (iAboutCommandPanel == null) {
        try {
            iAboutCommandPanel = new Panel();
            iAboutCommandPanel.setName("AboutCommandPanel");
            iAboutCommandPanel.setLayout(new java.awt.FlowLayout());
            getAboutCommandPanel().add(getAboutOkButton(),
                                       getAboutOkButton().getName());
        } catch (Throwable iExc) {
            handleException(iExc);
        }
    }
    return iAboutCommandPanel;
}
```

Fig. 11. Example of code clones in "B" (method's creating GUI widgets).

browsing build processes. Most of these code clones were classified into either of the following two types.

- If statements that determine process flow depending on the source of events. Fig. 10 shows one of them.
- Method declarations that create GUI widgets. Fig. 11 shows one of them.

These code clones are typical GUI processes.

Fig. 9(d) shows a closer view of part "C" in Fig. 9(a). It corresponds to the source files under directory ant/taskdef/optional/clearcase/. These source files implement several tasks working with ClearCase [7], a famous version control systems. Each command (for example, Checkin, Checkout, Update...) of ClearCase is implemented as a class. These source files were created by entire file copy, rather than copy and paste of particular parts of the text.

4.4. Metric Graph analysis

We investigated what kind of code clones is quantitatively discriminative by using Metric Graph. The following types of code clones are investigated. Before performing this analysis, we raised the lower limit of *RNR* to filter out clone sets whose *RNR* are less than 0.5.

- Clone sets whose *POP* are high.
- Clone sets whose *LEN* are high.
- Clone sets whose *NIF* are high.

4.4.1. Clone sets whose POP are high

Fig. 12 is one of the fragments comprising the clone set that has more fragments than any other. The clone set had

```
        } catch (Throwable iExc) {
            handleException(iExc);
        }
    }
    return iAboutCommandPanel;
}

/**
 * Return the AboutContactLabel property value.
 * @return java.awt.Label
 */
private Label getAboutContactLabel() {
    if (iAboutContactLabel == null) {
        try {
            iAboutContactLabel = new Label();
            iAboutContactLabel.setName("AboutContactLabel");
```

Fig. 12. One fragment comprising a clone set whose *POP* is highest.

31 fragments, and all were in source file VAJAntTool.java described in Section 4.3. Each fragment begins with the end of a method and ends with the beginning of its next method. This means the center parts of each method differ from each other.

4.4.2. Clone sets whose LEN are high

The two source files WebLogicDeployment.java and WebSphereDeployment.java, from directory ant/taskdefs/optional/ejb/, shared the longest code clones. The fragment size of the clone set was 282 tokens (77 lines). Both source files implement tasks working with WebLogic [21] and WebShpere [22], which are well-known application servers. Each source file has a method named isRebuildRequired, and both duplicated codes are in the methods. Some variable names differ between the methods, but other properties (indents, blank lines, comments) are completely identical, which indicates that those code clones were made by 'copy and paste'.

4.4.3. Clone sets whose NIF are high

The clone set that involved the most source files was implementations of consecutive accessor declarations, which appeared in 19 files (22 places). The accessor's names differed from each other, but CCFinder ignores differences of user-defined names[2] when detecting code clones. Since there are both setters and getters in the fragments of clone sets, the fragments are not simple consecutive code. The *RNR* value of the clone set was 85.

4.5. File List analysis

We investigated what kind of source files is discriminative by using File List. The following types of source files were investigated. In this analysis, we targeted only clone sets whose *RNR* are 0.5 or more.

- Files whose *ROC* are high.
- Files whose *NOC* are high.
- Files whose *NOF* are high.

[2] Naturally, it is possible to make CCFinder recognize differences of user-defined names.

Author's personal copy

Table 2
Duplicated ratios of files

Range of Duplicated Ratio($ROC_{0.5}$)	# Files	Percentage
0–10%	207	33%
11–20%	75	12%
21–30%	64	10%
31–40%	61	10%
41–50%	53	8%
51–60%	53	8%
61–70%	33	5%
71–80%	22	4%
81–90%	22	4%
91–100%	37	6%
Total	627	100%

4.5.1. Files whose ROC are high

Table 2 represents the duplicated ratio distribution of source files. As seen in this table, Ant has many source files with high duplicated ratios. Hence, we describe not only the highest duplicated ratio source file, but also the top 10 files. In the following items, the numbers in parentheses are the $ROC_{0.5}$ values.

- FlatFileNameMapper.java (1.0): Returns the file name included in a specified java.lang.String.
- IdentityMapper.java (1.0): This source file is a duplication of FlatFileNameMapper.java. Only the class name is different.
- DirSet.java (1.0): Treats a set of directories. This source file is a complete duplication of FileSet.java.
- FileSet.java (1.0): Treats a set of source files. This source file is a complete duplication of DirSet.java.
- CCMkbl.java (0.98): Implements a task working with ClearCase. This source file is duplicated with several source files implementing other ClearCase's tasks.
- SOSCheckin.java (0.97): Implements a task working with SourceOffSite [16]. This source file is duplicated with several source files implementing other SourceOffSite's tasks.
- StringLineComments.java (0.97): This source file is one of the file filters described in Section 4.3 part "A." It shares code clones with other filters.
- FieldRefCPInfo.java (0.96): Stores information of a field (for example, field name, type, owner class,...). This source file is a duplication of InterfaceMethodRefCPInfo.java.
- InterfaceMethodRefCPInfo.java (0.96): Stores information of a method (for example, method name, signature, owner class, ...). This source file is a duplication of FieldRefCPInfo.java.
- MSVSSCREATE.java (0.96): Implements a task working with Visual SourceSafe [19]. This source file is duplicated with several source files implementing other Visual SourceSafe's tasks.

4.5.2. Files whose NOC are high

The source file with the highest $NOC_{0.5}$ value was VAJAntToolGUI.java described in Section 4.3 part "B". This source file had 378 code clones, which was overwhelmingly more than any others.

4.5.3. Files whose NOF are high

The source file with the highest $NOF_{0.5}$ value was ant/taskdefs/optional/jsp/JspC.java, and most of the code clones in the source file were implementations of consecutive accessor declarations. These fragments are identical to those described in Section 4.4.3. Not only this source file but most such source files have many code clones of consecutive accessor declarations.

4.6. Evaluation

4.6.1. Filtering with RNR

We examined how the RNR filtering worked well. We browsed through the source code of all detected code clones so as to calculate precision, recall and *f*-value of the filtering. 869 of 2406 were practical clone sets and 1537 were uninteresting ones. The definitions of the values are the followings.

$$recall(\%) = 100 \times \frac{\#\ real\ uninteresting\ clone\ sets\ filtered\ out\ by\ RNR}{\#\ clone\ sets\ filtered\ out\ by\ RNR}$$

$$precision(\%) = 100 \times \frac{\#\ clone\ sets\ filtered\ out\ by\ RNR}{\#\ all\ real\ uninteresting\ clone\ sets}$$

$$f - value = \frac{2 \times recall \times precision}{recall + precision}$$

Fig. 13 illustrates transitions of recall, precision, and *f*-value when the RNR threshold is between 0 and 1.0. As mentioned above, in this case study, we used 0.5 as the threshold. Under this condition, recall was 100(%), which means that no practical clone set was accidentally filtered out at all. Also, precision was 65(%), which indicates that about one third real uninteresting clone sets were not filtered out. Using 0.5 as the threshold raised precision from 36(%) to 65(%). Therefore, we can conclude that most part of uninteresting clone clones were filtered out with no false positive by using 0.5 as the threshold.

It might be useful to set the threshold as to make *f*-value the greatest. In this case study, *f*-value reached its greatest when the threshold was 0.7. Under this condition, recall was 95(%) and precision was 82(%). In other words, one twentieth clone sets filtered out were practical ones and four fifths of real uninteresting clone sets were filtered out. However, we consider that accidentally filtering out practical code clones should be avoided because filtered clone sets might play an important role in software development and maintenance. Hence, it deems to be better to use 0.5 as the threshold than 0.7.

4.6.2. Using Gemini in other contexts

In this section, we will discuss the external validity of the case study. The discussion points are the followings.

Author's personal copy

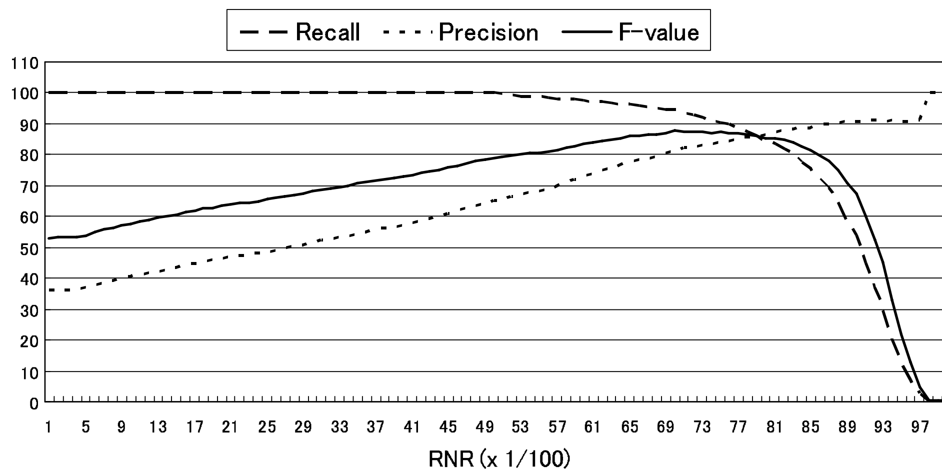


Fig. 13. Transition of recall, precision, and *f*-value.

Table 3
Size of target software systems and results of executions

Target Name	Size		CCFinder		Gemini	
	# Files	LOC	Run time	Mem usage	Init time	Mem usage
Ant	627	180,844	55 s	30 MBytes	4 s	46 MBytes
JDK1.5	6555	1,883,928	594 s	194 MBytes	15 s	137 MBytes

- Performance and scalability of Gemini.
- General versatility of the code clone analysis method described in this case study.
- Required users' skills to perform code clone analysis.

First discussion point is the performance and scalability of Gemini. We applied CCFinder and Gemini to a large-scale software system, JDK 1.5 besides Ant for investigating these properties. We used a PC-based workstation[3] to perform the tools. Table 3 illustrates the sizes of the target software systems and the results of executions. Note that total time of CCFinder's running and Gemini's initialization is only 10 min despites the huge size of JDK 1.5. Additionally, the memory usage of both CCFinder and Gemini is quite reasonable. Therefore, we can conclude that the performance and scalability of these tools are enough to be used in real software development and maintenance. Users can efficiently perform code clone analysis of a large-scale software system with an ordinary PC.

Second, we will discuss the general versatility of the code clone analysis method described in this case study. We have already analyzed many other open source and industrial software systems, and the analysis methods for them are almost the same as the one described in this case study. This analysis method can be applied to various software systems independently of their sizes, development patterns, and their domains. From many experiences of code clone analyses, we have learnt that '30' is an appropriate value of the minimum code clone size that CCFinder detects. But infrequently under this condition, especially in the case of large-scale software, too many code clones are detected, and we cannot efficiently analyze code clones. In such cases, users should change the minimum code clone size to '50' or '100' and re-run CCFinder for efficient analysis. Also, it became clear that industrial software tends to include more code clones than open source software. If users are going to detect and analyze code clones in a large-scale industrial software system, they should use '100' as the minimum code clone size in the first running of CCFinder.

Finally, we will discuss required users' skills to perform code clone analysis. In the code clone analysis with Gemini, they have to browse through that the source code of code clones and understand the implementations. Hence, they must be familiar with the programming language of the target software. And if the target software was developed by 2 or more people, the higher skill of reading source code is required. But they do not need to know the detail information of the target software; actually we do not have deep knowledge of Ant. If users had such information, they could perform deeper analysis. If users want to do the same kind of analysis as the one described in this case study, they do not need to have such information.

5. Related works and discussion

Kapser et al. implemented a visualization tool called CLICS to comprehend cloning [12]. CLICS shows the structures in the source files and the system architecture with

[3] CPU: PentiumIV 3.0 GHz, Memory Size: 2.0GBytes, OS: WindowsXP.

Author's personal copy

code clone information, which makes it possible for users to easily get the information of code clones they are interested in. Also, CLICS is facilitated with a query support feature to display code clones that satisfy the conditions of queries. CLICS does not implements Scatter Plot due to its limited scalability.

We believe that our enhanced Scatter Plot is scalable and useful for understanding the state of the code clones of a software system. In our case study, Scatter Plot worked smoothly on the source code of JDK 1.5, whose size was 1.8 million LOC, including 2,497,433 clone pairs in 12,522 clone sets, under $LEN \geqslant 30$ conditions. Our Scatter Plot displays the results from which the uninteresting code clones were already filtered out, which differentiates it from previous works [2,8,15]. This enhancement reflects our previous experiences, where we applied Scatter Plot to large software systems and found an enormous amount of such code clones. Kapser et al. share our opinion [12], and their tool, CLICS has implemented filtering features. Also, in Scatter Plot, the directory (package) separators are shown differently from the file ones, which makes it possible for users to discover the boundaries of directories and then finds out the directories (packages) that contain many code clones and directories that share many code clones with other directories. Users can withness the state of the practical code clones in the package hierarchy of a software system by using our Scatter Plot since uninteresting code clones are filtered out.

Kapser et al. also describe functionalities to support navigation and understanding cloning in a software system [12]. The functionalities are as below.

(1) *Facilities to evaluate overall cloning activity.*
(2) *Mechanisms to guide users toward clones that will be most effectively used in their task.*
(3) *Methods for filtering and refining the analysis of the clones.*

Here, we consider these functionalities in the context of our tool, Gemini. Since Scatter Plot provides a bird's eye view, it accomplishes the first functionality. We believe that Metric Graph and File List implement the second functionality because we can use them to easily get arbitrary clone sets based on their quantitative features. The third functionality is realized through the filtering metric *RNR* that enables users to filter out uninteresting code clones. With all the above characteristics, Gemini is definitely useful and appropriate as a code clone visualization tool.

Rieger et al. suggested and implemented diagrams to visualize code clone information [15]. Their diagrams are based on the principle of Polymetric View and provide abstracted code clone information on various granularities for users. They also argue that in the large-scale software, since too many code clones are detected, filtering functionalities are essential.

Johnson suggested a navigation method using HTML [10]. The hyperlink functionality of HTML enables users to jump freely between source files having clone relations with each other or fragments included in the same clone set (in his paper, Hash is used instead of clone set). We agree that hyperlink properties offer convenient navigation for users, but there is no functionality to see the state of the system's code clones.

As described in Section 2.1, several code clone detection methods have been proposed. A code clone detection tool Dup [3] uses a sequence of lines to represent source code and detects clones line-by-line. A language independent clone detection tool duploc [8] reads source files, makes a sequence of lines, removes white spaces and comments in lines, and detects matches by string-based Dynamic Pattern Matching (DPM). Baxter et al. [6] proposed a technique to extract clone pairs of statements, declarations, or their sequences from C source files. The tool parses source code to build an abstract syntax tree and compares its subtrees by characterization metrics. Komondoor et al. [13] proposed a method using program slicing in which a program dependence graph is constructed by analyzing target source code. Identical or similar parts are detected as code clones. A clone-detecting method proposed by Mayrand et al. uses a representation called Intermediate Representation Language (IRL) to characterize each function in the source code [14]. A clone detection tool, Similar Method Classifier (SMC) [4], uses a hybrid approach of characterization metrics and dynamic pattern matching (DPM).

Basit et al. suggested a method detecting structural clones [5]. A structural clone is a pattern of cloned code fragments, and it indicates the presence of design-level similarities. They also implemented a tool detecting such clones based on the "market basket analysis" technique of the Data Mining domain. Providing structural clone information is a great support of program comprehension, but how the information is expressed is a big challenge.

Walenstein et al. reported that the judgment of code clones varies among experts [20]. In one of their experiments, three experts disagreed whether the fragments are really code clone or not for more than 60% of automatically detected clones. Our metric system does not settle the controversy, but it helps users choose what kind of codes should be regarded as code clones.

6. Conclusion

We proposed visualization and characterization methods of code clones in a software system and implemented them as a tool, Gemini. Gemini provides valuable mechanisms that perform the following functions describe below.

- Filters out uninteresting code clones.
- Views the states of code clones over a system.
- Navigates code clones that have feature users are interested in.

As described in Section 4, our proposed methods worked well and we could determine various cloning activities in Ant.

Author's personal copy

Of course, further research issues remain. For example, sometimes gaps occur between the original code fragments and the modified ones since developers are usually modifying copied and pasted code fragments. Here, we call modified the code fragments, which include some gaps, *Gapped code clones* [18]. By treating the gapped code clones as well, our methods can become more effective.

As described in Section 3.2.1, the proposed filtering method only deals with language-dependent code clones. We are going to improve the method to filter out application-dependent code clones. Features of application-dependent code clones are so strikingly different among software systems that such metric-based filtering as *RNR* is probably unpractical. We believe that pattern-based filtering works well. For example, users input code patterns that they are not interested in. After that, Gemini filters out all code clones whose contents correspond to the code patterns. Using such a filtering approach, we can filter out any kind of code clones dependent on the domain or the framework of a system.

Acknowledgement

We thank Yasushi Ueda of the Japan Aerospace Exploration Agency for his contributions to a previous paper [17].

References

[1] Ant. http://ant.apache.org/.
[2] B.S. Baker, A program for identifying duplicated code, in: Proceedings of the 24th Symposium of Computing Science and Statistics, March 1992, pp. 49–57.
[3] B.S. Baker, On finding duplication and near-duplication in large software systems. in: Proceedings of the 2nd Working Conference on Reverse Engineering, July 1995, pp. 86–95.
[4] M. Balazinska, E. Merlo, M. Dagenais, B. Lagüe, K. Kontogiannis, Advanced clone-analysis to support object-oriented system refactoring, in: Proceedings of the 7th IEEE International Working Conference on Reverse Engineering, November 2000, pp. 98–107.
[5] H.A. Basit, S. Jarzabek, Detecting higher-level similarity patterns in programs, in: Proceedings of the ACM SIGSOFT Symposium on the Foundations of Software Engineering, September 2005, pp. 156–165.
[6] I. Baxter, A. Yahin, L. Moura, M. Anna, L. Bier, Clone detection using abstract syntax trees, in: Proceedings of the International Conference on Software Maintenance 98, March 1998, pp. 368–377.
[7] ClearCase. http://www-306.ibm.com/software/awdtools/clearcase/.
[8] S. Ducasse, M. Rieger, S. Demeyer, A language independent approach for detecting duplicated code, in: Proceedings of the International Conference on Software Maintenance 99, August 1999, pp. 109–118.
[9] M. Fowlor, Refactoring: Improving the Design of Existing Code, Addison Wesley, 1999.
[10] J.H. Johnson, Navigating the textual redundancy web in legacy source, in: Proceedings of the 1996 Conference of Centre for Advanced Studies on Collaborative Research, November 1996, pp. 7–16.
[11] T. Kamiya, S. Kusumoto, K. Inoue, Ccfinder: A multi-linguistic token-based code clone detection system for large scale source code, IEEE Transactions on Software Engineering 28 (7) (2002) 654–670.
[12] C. Kapser, M. Godfrey, Improved tool support for the investigation of duplication in software, in: Proceedings of the 21st International Conference on Software Maintenance, September 2005, pp. 305–314.
[13] R. Komondoor, S. Horwitz, Using slicing to identify duplication in source code, in: Proceedings of the 8th International Symposium on Static Analysis, July 2001, pp. 40–56.
[14] J. Mayrand, C. Leblanc, E. Merlo, Experiment on the automatic detection of function clones in a software system using metrics, in: Proceedings of the International Conference on Software Maintenance 96, November 1996, pp. 244–253.
[15] M. Rieger, S. Ducasse, M. Lanza, Insights into system-wide code duplication, in: Proceedings of the 11th Working Conference on Reverse Engineering, November 2004, pp. 100–109.
[16] SourceOffSize. http://www.sourcegear.com/sos/.
[17] Y. Ueda, T. Kamiya, S. Kusumoto, K. Inoue, Gemini: maintenance support environment based on code clone analysis, in: Proceedings of the 8th International Symposium on Software Matrics, June 2002, pp. 67–76.
[18] Y. Ueda, T. Kamiya, S. Kusumoto, K. Inoue, On detection of gapped code clones using gap locations, in: Proceedings of the 9th Asia-Pacific Software Engineering Conference, December 2002, pp. 327–336.
[19] VisualSourceSafe. http://msdn.microsoft.com/vstudio/previous/ssafe/.
[20] A. Walenstein, N. Jyoti, J. Li, Y. Yang, A. Lakhotia, Problems creating task-relevant clone detection reference data, in: Proceedings of the 10th Working Conference on Reverse Engineering, November 2003, pp. 285–294.
[21] WebLogic. http://www.beasys.com/products/weblogic/.
[22] WebSphere. http://www-306.ibm.com/software/websphere/.
[23] S.W.L. Yip, T. Lam, A software maintenance survey, in: Proceedings of the 1st Asia-Pacific Software Engineering Conference, December 1994, pp. 70–79.

Author's personal copy