Software Quality Analysis by Code Clones in Industrial Legacy Software

Akito Monden[1] Daikai Nakae[1] Toshihiro Kamiya[2]
Shin-ichi Sato[1,3] Ken-ichi Matsumoto[1]

[1]Nara Institute of Science and Technology, 8916-5, Takayama, Ikoma, Nara, 630-0101, Japan, {akito-m, daikai-n, siniti-s, matumoto}@is.aist-nara.ac.jp
[2]PRESTO, Japan Science and Technology Corporation, 1-8, Honcho 4-Chome, Kawaguchi, Saitama, 332-0012, Japan, kamiya@is.aist-nara.ac.jp
[3]NTT DATA Corporation, 3-3, Toyosu 3-Chome, Koto-ku, Tokyo, 135-6033, Japan, satousnb@nttdata.co.jp

Abstract

Existing researches suggest that the code clone (duplicated code) is one of the factors that degrades the design and the structure of software and lowers the software quality such as readability and changeability. However, the influence of code clones on software quality has not been quantitatively clarified yet.

In this paper, we have tried to quantitatively clarify the relation between code clones and the software reliability and maintainability of twenty years old software. As a result, we found that modules having code clones (clone-included modules) are more reliable than modules having no code clone (non-clone modules) on average. Nevertheless, the modules having very large code clones (more than 200 SLOC) are less reliable than non-clone modules. We also found that clone-included modules are less maintainable (having greater revision number on average) than non-clone modules; and, modules having larger code clone are less maintainable than modules having smaller code clone.

1. Introduction

Coping with large software is a great challenge for companies who maintain them. Large software becomes more and more complicated and unchangeable than it used be through a long-continued process of maintenance [21][24][26]. It is partially because changing the software itself makes the software more difficult to be changed; and, it is partially because turnover rate of maintainers is much higher than the large system maintained by them [22]. Thus, when software gets really aged and it finally became a legacy system, it is then extremely difficult for maintainers to keep up the maintainability and the reliability of the system.

In this paper we look into twenty years old software focusing on the code clone, which is a duplicated code section in source files of software. Previous works suggest that considerable parts (5-50%) of large software are code clones [2][7][17]. One of the major reasons why clones occur is code reuse by copying a pre-existing program fragment [5][7][23]; and, such code reuse may easily take place when one adds functionalities to an existing system during maintenance. Previous work suggests that the code clone is one of the factors that degrades the design and the structure of software and lowers the software quality such as readability and changeability [7]. If one revises a copy of duplicated code sections, he/she must update all the other copies, and this may raise the maintenance cost. Moreover, if he/she overlooks one of the copies, a fault will remain in the copy, and this may lower the reliability of the system. However, the influence of code clones on software quality has not been quantitatively clarified yet.

The purpose of this paper is to quantitatively clarify the relation between the code clone and the software quality. Especially, we focus on the reliability and maintainability of a large legacy system. If a certain kind of clone particularly had a bad effect on maintenance, we can feedback it to the field so that maintainers may remove such a code clone or try not to produce such a code clone.

There are some related works concerning the code clone. Various kinds of clone detection techniques have been proposed [1-5][7][12-17][19]. For example, Baker proposed an efficient technique that can detect clones in huge C software in a realistic time [1]. A systematic technique for reducing code clones of object-oriented programs is also proposed [10]. While these researches focus on detecting and removing code clones, this research focuses on analyzing detected code clones.

The remainder of this paper first describes the goal and approach of our study (Section 2). Next describes a technique we used for detecting code clones (Section 3). Then we describe an experiment for analyzing the relation between code clones and the reliability and maintainability of an industrial legacy system (Section 4). Afterwards, we describe the result of the experiment (Section 5); and in the end, conclusions and future topics will be shown (Section 6).

2. Goals and Approach

2.1. Goals of this study

The main goals of this study are follows:

(1) Clarify the relation between code clones and the reliability.

(2) Clarify the relation between code clones and the maintainability.

It is possible to estimate the reliability of a system by measuring the number of faults found in recent years. We can say a system that had fewer faults was more reliable than a system that had more faults in recent years. So, if we can measure the number of faults of an existing system, we will be able to analyze the relation between code clones and the reliability of that system.

On the other hand, it is not easy to measure the maintainability of a system. Essentially, maintainability is related to the maintenance cost (person-hours). We can say a system of poor maintainability requires more cost in doing maintenance works than that of higher maintainability. However, under the uncontrolled environment, observing how many person-hours are required to perform maintenance works does not mean measuring the maintainability. The problem is that it measures not only maintainability, but also the person doing the maintenance, as well as the environment in which the person is working, the tools the person is using, and the amount of the work itself [25].

Another way to estimate the maintainability is using software (product) metrics. Many software metrics have been proposed to measure the complexity of software such as McCabe's Cyclomatic number, Halsted's metrics, and Chidamber & Kemerers' metrics, etc [6][9][11][20]. Each of these conventional metrics may measure a certain aspect of maintainability; however, they are not useful in our analysis because code clones are essentially independent from these metrics. For example, we assume a case we have pointed out that a certain module had low maintainability because the module had a large cyclomatic number (per SLOC). In this case, whether this module had a code clone or not, the influence of code clones to the maintainability cannot be addressed. In our study, we must somehow estimate the maintainability without using complexity metrics.

In this paper, as a simple and practical solution, we use the revision number for estimating the maintainability of software modules. Generally, as we repeatedly revise a system, such as adding and changing functionalities, the system becomes more complicated and more difficult to be maintained than it was before. In other word, it can be considered that a system having higher revision number is more difficult in maintenance than that having lower revision number on average. It is pointed out in past researches that various properties of a system, such as modularity of functions, degrade as we continually revise the system [6][18]. Below we describe our considerations in measurement:

(1) The increase of revision number could be due to various maintenance activities such as adding and changing functionality, enhancing and adapting the code, and fixing faults, etc. Although not all the activities cause the degradation of maintainability, significant part of revisions in our twenty-years old system had been taken up with adding and changing functionality, which are inevitable parts of an evolving system, and, these changes had certainly degraded the maintainability of the system. In addition, some modules in our system were redesigned as a new module through reengineering activities, however, in this case the revision number of the modules were set to zero.

(2) We do *not* believe that a module having greater revision number is *always* less maintainable than that having smaller revision number. Nevertheless, we believe that an *average* maintainability is higher in modules having smaller revision number than that having greater revision number.

(3) We do not believe that it is significantly more difficult to maintain a module set whose average revision number is 11 than that is 10. Nevertheless, we believe that it becomes significant if the average revision number becomes extremely higher (e.g. 50).

2.2. Module based analysis

In many industrial software systems, the module (file) is a basic unit of software, and, software metrics such as the number of faults and the revision number are measured in each module. Thus, this paper conducted a module-based analysis to clarify the relation between software quality and code clones.

We classified clone pairs into following two types (Figure 1.)

(1) In-module clone pair

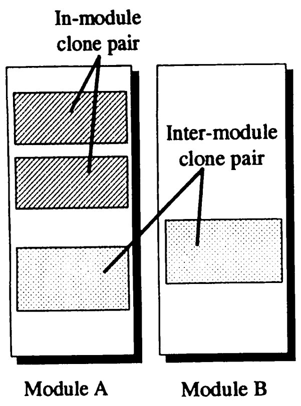


Figure 1. Two types of code clone pairs

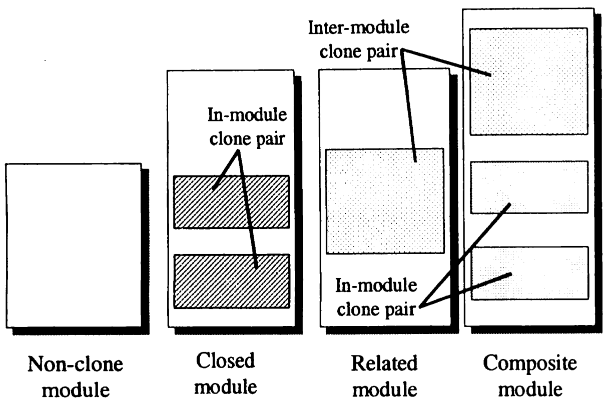


Figure 2. Module classification

We call a code fragment pair "in-module clone pair" if both fragments in the pair exist in the same module.

(2) Inter-module clone pair

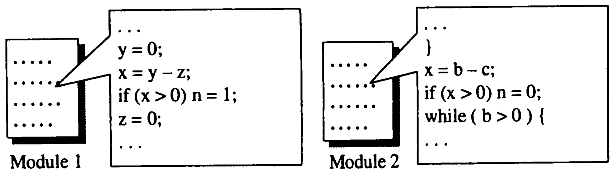
We call a code fragment pair "inter-module clone pair" if each fragment in the pair exists in the different module.

These two types of clone pairs may have different influence on software quality. Inter-module clones may implicitly increase the functional coupling between modules, while in-module clones do not affect the strength of coupling between modules.

Based on above classification, we also classified modules into following four types (Figure 2.)

(1) Non-clone module

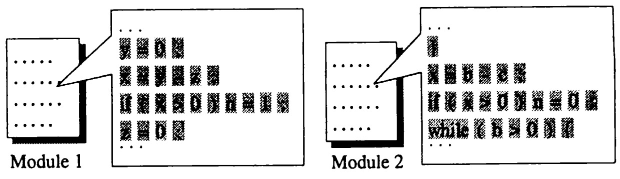
A module containing no clones.



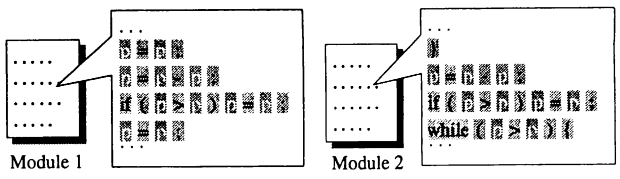
```
...
y = 0;
x = y – z;
if (x > 0) n = 1;
z = 0;
...
```

```
...
}
x = b – c;
if (x > 0) n = 0;
while ( b > 0 ) {
...
```

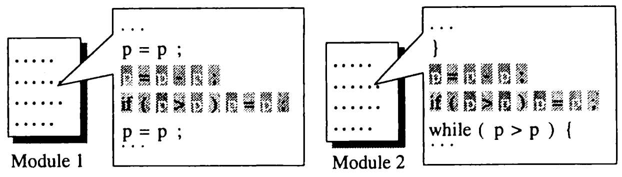
(a) Original source code



(b) Lexical analysis



(c) Transformation



```
...
p = p ;
p = p ;
...
```

```
...
}
while ( p > p ) {
...
```

(d) Match detection and formatting

Figure 3. Clone detection procedure

(2) Clone-included module

A module containing at least one code clone pair. This type of module is classified into following three modules.

(2a) Closed module

A module containing in-module clone pairs only.

(2b) Related module

A module containing inter-module clone pairs only.

(2c) Composite module

A module containing both in-module and inter-module clone pairs.

3. Detection of Code Clones

Recently, various kinds of clone detection techniques and tools have been proposed [1-5][7][12-17][19]. Since clones usually occur when a code fragment is copied and partly modified, it is insufficient to detect a code fragment that is exactly identical to another fragment. Thus, existing tools also detect a fragment that is nearly identical to others.

In this paper we used a token-based code clone detection technique proposed by Kamiya et al. [11][12] because their technique have industrial strength, and is applicable to a million-line size system within affordable computation time and memory usage. This technique is also easily applicable to legacy software written in old programming language such as COBOL and PL/I.

Below we briefly describe the overview of clone detecting process we used.

(1) Lexical analysis
All the source files are divided into tokens based on lexical rules of the programming language (Figure 3(a) and 3(b)). White spaces and comments are ignored in this analysis.

(2) Transformation
Each token related to types, variables and constants is replaced with a special token (Figure 3(c)). This replacement makes code-portions with different variable names to become clone pairs.

(3) Match detection and formatting
From all the sub-strings on the transformed token sequence, equivalent pairs are detected as clone pairs (Figure 3(d)). Then, each location of clone pair is converted into line number on the original source files.

4. Experiment

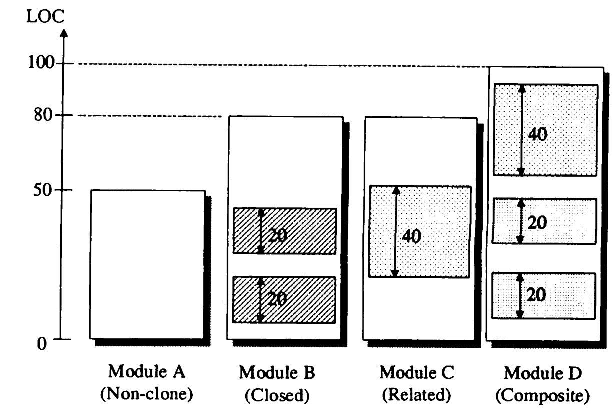
4.1. Target system

The target system is legacy software developed about 20 years ago in NTT DATA Corporation. This software was developed to manage transactions for a public institute and has been continuously maintained till today. It consists of about one million lines in 2000 modules (files) written in a COBOL-like language, which is an expansion of COBOL.

4.2. What we have measured

In order to remove accidental duplication of code fragments, we detected clone pairs having at least 30 same lines. In the detection, we ignored self-overlapping clone pairs. Below describes what we have measured in this experiment.

(1) LOC (Lines of code)



	Module A	Module B	Module C	Module D
MAXLEN:	0	20	40	40
COVERAGE:	0	50%	50%	80%

Figure 4. Example of code clone metrics

Lines of code of each module.

(2) AGE (Module age)
The number of days from the date each module is initially developed to the present.

(3) REV (Revision number)
The number of revisions made upon each module till present. The revision includes any kind of modifications done to each module such as fixing faults and adding and changing functionalities.

(4) Faults (The number of faults)
The number faults found from each module in recent years (past six years in this experiment).

(5) MAXLEN (Length of maximum clone)
The length (LOC) of the largest code clone included in each module.

(6) COVERAGE (Coverage of clone)
The percentage of lines that include any portion of clone in each module.

Figure 4 shows an example of code clone metrics. MAXLEN of module B is 20 because module B contains two clones and both of them are of 20 LOC. Similarly, MAXLEN of module D is 40 because the largest clone included in module D is of 40 LOC.

COVERAGE of module B is 80% (= 40 / 80 * 100) because total clone size is 40 LOC (= 20 + 20) and the module size is 80LOC. Similarly, COVERAGE of module D is 80% (= 80 / 100 * 100) because total clone size is 80 LOC (= 40 + 20 + 20) and the module size is 100 LOC.

5. Result of experiment

5.1. Module type and code clone metrics

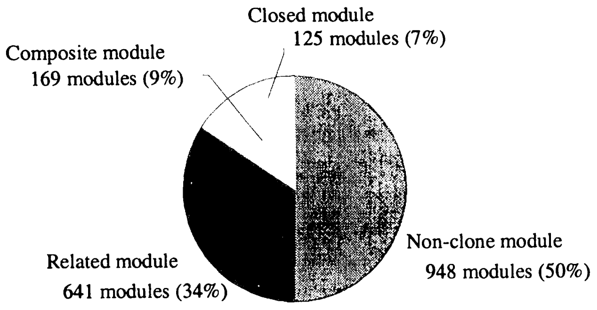


Figure 5. Classification of modules

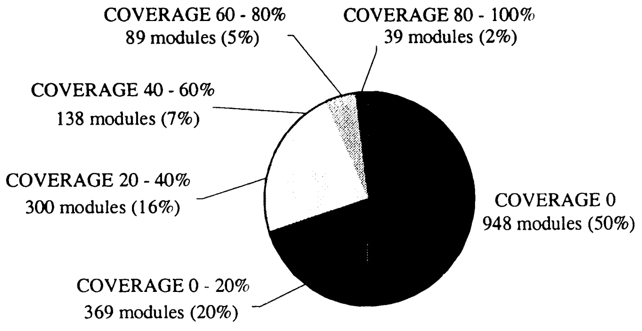


Figure 6. COVERAGE and the number of modules

Figure 5 shows the classification of modules. About 50% of modules are clone-included module. This result follows the previous research that Cobol payroll system had 59% code duplication [7]. As shown in Figure 5, most of clones are inter-module clone. Closed modules hold only 7% while related modules account for 34% of the whole.

Figure 6 shows the relation between COVERAGE and the number of modules. The non-clone modules and the low coverage modules (0-40%) together account for 86% of the whole. On the other hand, 2% of the modules are of 80% coverage. Some of them were almost identical copies of each other.

Figure 7 shows the COVERAGE and AGE of each module. We see in the figure three clusters of modules. For convenience, we call them YOUNG, MIDDLE, and OLD. The average COVERAGE for these module clusters is 18%, 13%, and 11% respectively. More clones are detected from modules that are younger than from older modules.

5.2. Reliability analysis

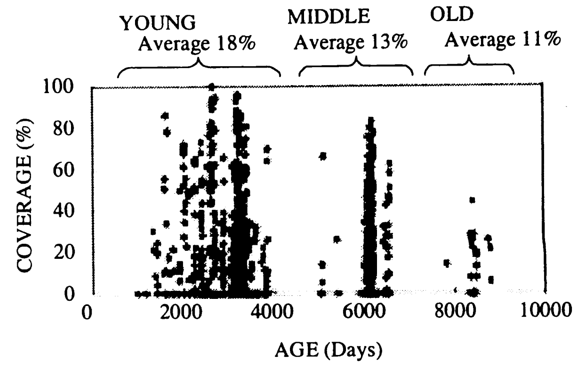


Figure 7. COVERAGE and AGE

In order to evaluate the reliability of modules, we used the number of faults per line as a reliability measure. Figure 8 shows a comparison of reliability between non-clone modules and clone-included modules. Obviously, clone-included modules are more reliable than non-clone modules. Clone-included modules are 1.7 times as reliable as non-clone modules on average. One possible interpretation for this result is that copying code from trusted part can lessen the fault injection compared with writing the code from scratch. Another possible interpretation is that code fragments created by copy-and-past programming do not have new types of functionality, so that there may be little chance of introducing unknown types of faults in the fragments.

Figure 9 shows the reliability of each type of modules. Closed modules, related modules, and composite modules are all more reliable than non-clone modules on average.

Figure 10 shows the relation between the number of faults per line and MAXLEN. As shown in the figure, we classified clone-included modules into four groups based on modules' MAXLEN. As the MAXLEN of modules gets larger (30 MAXLEN up to 199 MAXLEN), the reliability of modules becomes higher; however, the modules having more than 200 lines of code-clones suddenly show the worst reliability. This result suggests that producing too large clones degrades software reliability.

Although the result showed evidence that clone-included modules were more reliable than non-clone modules in past six years, the relation between the code clone and the reliability were not clarified yet. Further analysis is needed in the future.

5.3. Maintainability analysis

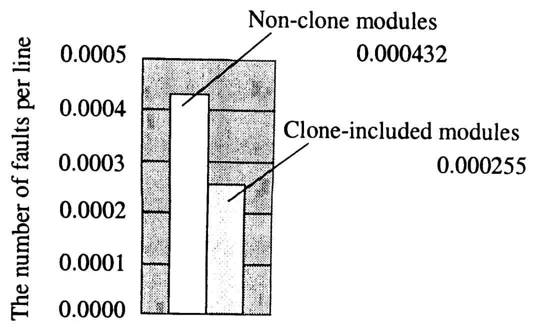


Figure 8. Relation between reliability and clones

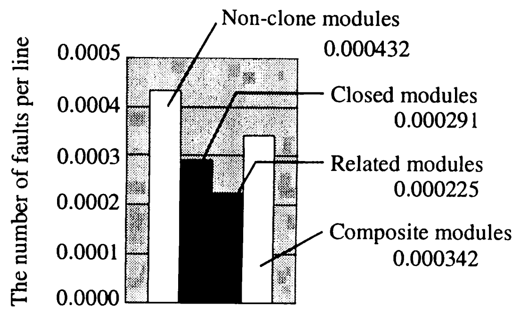


Figure 9. Reliability of different types of modules

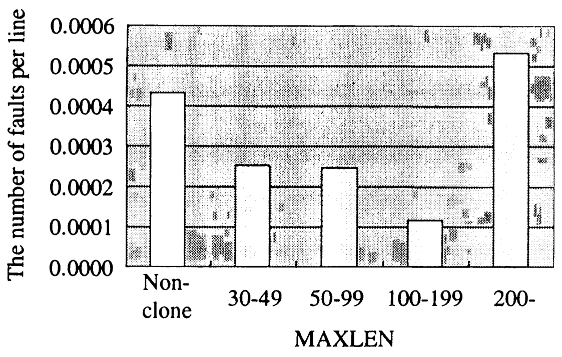


Figure 10. Relation between faults and MAXLEN

In order to evaluate the maintainability of modules, we have used the revision number as a maintainability measure. As we stated in Section 2.1, we consider a module set having higher revision number on average is more difficult to be maintained than that having lower revision number on average. Figure 11 shows a comparison of the maintainability between non-clone modules and clone-included modules. Obviously, clone-included modules are less maintainable than non-clone modules.

Figure 12 shows the maintainability of each type of modules. Closed modules, related modules, and composite modules are all less maintainable than non-clone modules on average.

Figure 13 shows the relation between the revision number and MAXLEN. As the MAXLEN of modules gets larger, the revision number of modules becomes higher. That is to say, a module having larger code clone is less maintainable than the module having smaller code clone. This result suggests that producing large clones raises maintenance cost.

Although the result showed evidence that clone-included modules have higher revision number than non-clone modules on average, the relation between the code clone and the maintainability (revision number) were not clarified yet. One possible interpretation is that code clones caused the revisions, and, another interpretation is that revisions produced the code clones. One thing that follows latter interpretation is that maintainers of this system said that when they add and/or change the functionality, they often intentionally produce new clone pairs by using copy-and-paste programming in order to keep up the reliability of the system. However, further quantitative analysis is needed in the future.

6. Summary

In this paper we tried to quantitatively clarify the relation between code clones and the software reliability and maintainability of twenty years old software. Below describes the overview of the result.

- Clone-included modules are 1.7 times as reliable as non-clone modules on average.
- Closed modules, related modules, and composite modules are all more reliable than non-clone modules on average.
- Nevertheless, the modules having very large code clones (more than 200 lines) are less reliable than non-clone modules.
- Clone-included modules are less maintainable maintainable (having greater revision number) than non-clone modules on average.
- Closed modules, related modules, and composite modules are all less maintainable than non-clone modules on average.
- The modules having larger code clone are less maintainable than modules having smaller code clone.

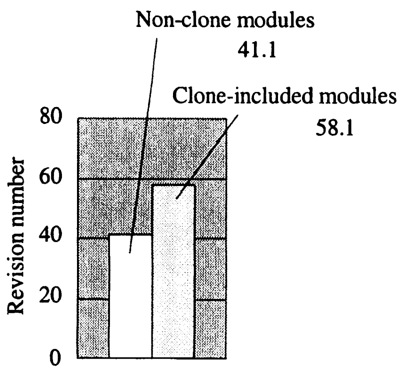


Figure 11. Relation between maintainability and clones

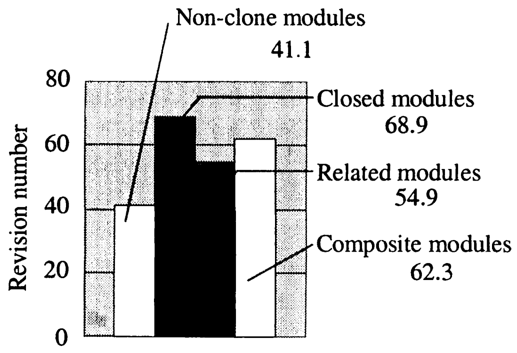


Figure 12. Maintainability of different types of modules

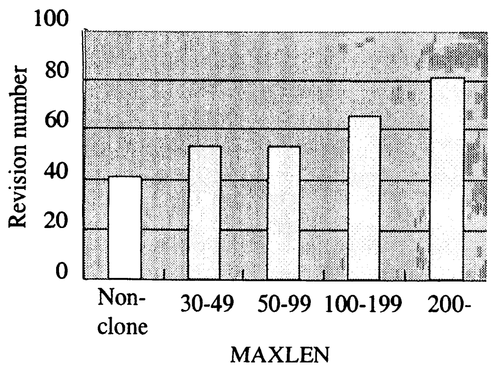


Figure 13. Relation between REV and MAXLEN

We also had some findings related to the nature of code clones:

- Closed modules held only 7% while related modules account for 34% of the whole modules.
- The non-clone modules and the low coverage modules (0-40%) together accounted for 86% of the whole. On the other hand, 2% of the modules were of 80% coverage.
- More clones are detected from modules that are younger than from older modules.

Although we have quantitatively pointed out that there is a relation between code clones and the software reliability and maintainability, the relation itself is not clarified yet. In order to make valid interpretations to our observations, we are planning to conduct further quantitative analyses.

Acknowledgement

This study was supported by the Industrial Technology Research Grant Program from the New Energy and Industrial Technology Development Organization (NEDO) of Japan.

References

[1] B.S. Baker, "A program for identifying duplicated code," *Proc. 24th Symposium on the Interface: Computing Science and Statistics*, pp. 49-57 Mar. 1992.

[2] B.S. Baker, "On finding duplication and near-duplication in large software system," *Proc. Second IEEE Working Conf. on Reverse Eng. (WCRE'95)*, pp. 86-95 Jul. 1995.

[3] M. Balazinska, E. Merlo, M. Dagenais, B. Lagüe, and K.A. Kontogiannis, "Measuring clone based reengineering opportunities," *Proc. 6th IEEE Int'l Symposium on Software Metrics (METRICS '99)*, pp. 292-303, Boca Raton, Florida, Nov. 1999.

[4] M. Balazinska, E. Merlo, M. Dagenais, B. Lagüe, and K.A. Kontogiannis, "Partial redesign of Java software systems based on clone analysis," *Proc. 6th IEEE Working Conf., on Reverse Eng. (WCRE '99)*, pp. 326-336, Atlanta, Georgia, Oct. 1999.

[5] I.D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier, "Clone detection using abstract syntax trees," *Proc. IEEE Int'l Conf. on Software Maintenance (ICSM'98)*, pp. 368-377, Bethesda, Maryland, Nov. 1998.

[6] S. R. Chidamber and C.F. Kemerer, "A metrics suite for object oriented design," *IEEE Trans. on Software Eng.*, Vol. 20, No. 6, pp. 476-493, 1994.

[7] S. Ducasse, M. Rieger, and S. Demeyer. "A language independent approach for detecting duplicated code," *Proc. IEEE Int'l Conf. on Software Maintenance (ICSM'99)*, pp. 109-118. Oxford, England. Aug. 1999.

[8] S. G. Eick, T. L. Graves, A. F. Karr, J. S. Marron, and A. Mockus, "Does code decay? Assessing the evidence from change management data," *IEEE Trans. on Software Engineering*, Vol. 27, No. 1, pp. 1-12, Jan. 2001.

[9] N. E. Fenton, "Software metrics: A rigorous approach," Chapman & Hall, London, 1991.

[10] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts, "Refactoring: Improving the design of existing code," Addison-Wesley, 1999.

[11] M. H. Halstead, "Elements of software science," Elsevier, New York, 1977.

[12] T. Kamiya, F. Ohata, K. Kondou, S. Kusumoto, and K. Inoue: "Maintenance support tools for Java programs: CCFinder and JAAT", *Proc. 23rd Int'l Conf. on Software Eng. (ICSE2001)*, pp. 837-838, Toronto, Canada, May. 2001.

[13] T. Kamiya, S. Kusumoto, and K. Inoue, "A token-based code clone detection technique and its evaluation," *Technical Report of IEICE (The Institute of Electronics, Information and Communication Engineers)*, Vol. 100, No. 570, pp. 41-48, Jan. 2001.

[14] J. H. Johnson, "Identifying redundancy in source code using fingerprints," *Proc. IBM Centre for Advanced Studies Conference (CAS CON'93)*, pp. 171-183, Toronto, Ontario. Oct. 1993.

[15] J. H. Johnson, "Substring matching for clone detection and change tracking," *Proc. IEEE Int'l Conf. on Software Maintenance (ICSM'94)*, pp. 120-126. Victoria, British Columbia, Canada. Sep. 1994.

[16] K.A. Kontogiannis, R. De Mori, E. Merlo, M. Galler, and M. Bernstein, "Pattern matching techniques for clone detection and concept detection," *J. Automated Software Eng.*, Kluwer Academic Publishers, vol. 3, pp. 770-108, 1996.

[17] B. Laguë, E.M. Merlo, J. Mayrand, and J. Hudepohl. "Assessing the benefits of incorporating function clone detection in a development process," *Proc. IEEE Int'l Conf. on Software Maintenance (ICSM'97)*, pp. 314-321, Bari, Italy. Oct. 1997.

[18] M. M. Lehman and L. A. Belady, "Program evolution: Process of software change," Academic Press, 1985.

[19] J. Mayland, C. Leblanc, and E. M. Merlo. "Experiment on the automatic detection of function clones in a software system using metrics", *Proc. IEEE Int'l Conf. on Software Maintenance (ICSM'96)*, pp. 244-253, Monterey, California, Nov. 1996.

[20] T. J. McCabe, "A complexity measure," *IEEE Trans. on Software Engineering*, Vol. 2, No.4, pp. 308-320, Dec. 1976.

[21] A. Monden, S. Sato, K. Matsumoto, and K. Inoue, "Modeling and Analysis of Software Aging Process," *Int'l Conf. on Product Focused Software Process Improvement (Profes2000), Lecture Notes in Computer Science*, Vol. 1840, pp. 140-153, Springer-Verlag, June 2000.

[22] A. Monden, S. Sato and K. Matsumoto, "Capturing industrial experiences of software maintenance using product metrics," *Proc. 5th World Multi-Conference on Systemics, Cybernetics and Informatics (SCI2001)*, Vol. 2, pp. 394 - 399, Florida, USA, July 2001.

[23] T. Nakae, T. Kamiya, A. Monden, H. Kato, S. Sato, and K. Inoue, "Quantitative analysis of cloned code on legacy software," *Technical Report of IEICE (The Institute of Electronics, Information and Communication Engineers)*, Vol. 100, No. 570, pp. 57-64 Jan. 2001 (in Japanese).

[24] N. F. Schneidewind, and C. Ebert, "Preserve of redesign legacy systems?," *IEEE Software*, Vol. 15, No.4, pp.14-17, July/Aug. 1998.

[25] H. M. Sneed, "Economics of software re-engineering," *J. of Software Maintenance: Research and Practice*, Vol.3, No.3, pp.163-182, 1991.

[26] H. M. Sneed, "Planning the reengineering of legacy systems," *IEEE Software*, Vol.12, No.1, pp.24-34, Jan. 1995.