Software Quality Analysis by Code Clones in Industrial Legacy Software

Akito Monden[1] Daikai Nakae[2] Toshihiro Kamiya[3] Shin-ichi Sato[4] Ken-ichi Matsumoto[1]

[1] Graduate School of Information Science, Nara Institute of Science and Technology,
8916-5 Takayama, Ikoma, Nara 630-0101, Japan
[2] FUJITSU KCN
[3] PRESTO JST
[4] NTT DATA Corporation
{akito-m, daikai-n, kamiya}@is.aist-nara.ac.jp, satousnb@nttdata.co.jp, matumoto@is.aist-nara.ac.jp

ABSTRACT

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

In this paper we 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 a code clone (clone-included modules) are more reliable than modules having no code clone (non-clone modules) in average. Nevertheless, the modules having very large code clones (more than 200 lines) are less reliable than non-clone modules. We also found that clone-included modules are less maintainable than non-clone modules; and, modules having larger code clone are less maintainable than modules having smaller code clone.

Keywords: Software maintenance, software aging, code decay, reengineering, refactoring

1. INTRODUCTION

Coping with large software is a big challenge for companies who maintain them. Large software becomes more and more complicated and unchangeable through a long-continued process of maintenance [19][22][24]. 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 [20]. 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 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][6][15]. One of the major reasons why clones occur is code reuse by copying a pre-existing program fragment [5][6][21]; 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 structure of software and lowers the software quality such as readability and maintainability [6]. 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 whole 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 [1-6][11-15][17]. For example, Baker proposed an efficient technique that can detect clones in huge C software in a realistic time [1]. Systematic technique for reducing code clones of object-oriented programs is also proposed [9]. While these researches focus on detecting and removing code clones, this research focuses on analyzing detected code clones.

The remainder of the paper first describes the goal and approach of this 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 ex experiment (Section 5); and in the end, conclusions and future topics will be shown (Section 6).

2. GOAL AND APPROACH

2.1 Goal 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.

On the other hand, it is not easy to estimate the maintainability of a system. The maintainability is essentially related to the maintenance cost; however, the maintenance cost depends on maintenance situations such as 5W1H (when, where, who, what, why, how) [23].

Hence, we need to indirectly estimate the maintainability of a system. One possible way is using software (product) metrics. Many software metrics have been proposed to measure the complexity of software such as Cyclomatic number, Halsted's metrics, Fan-in and Fan-out, etc [8][10][18]. However, these metrics only describe a certain aspect of a system; and, more than anything else, these metrics do not consider the influence of the code clone.

In this paper, as a simple and practical solution, we used the revision number for estimating the maintainability. 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 to be maintained than the system having lower revision number. It is pointed out in past researches that various properties of a system degrade as we continually revise the system [7][16].

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 clones into following two types (Figure 1.)

(1) In-module clone

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

(2) Inter-module clone

We call a code fragment "inter-module clone" if one of the equivalent fragments exists in the different module.

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

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

(1) Non-clone module

A module containing no clones.

(2) Clone-included module

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

(2a) Closed module

A module containing in-module clones only.

(2b) Related module

A module containing inter-module clones only.

(2c) Composite module

A module containing both in-module and inter-module clones.

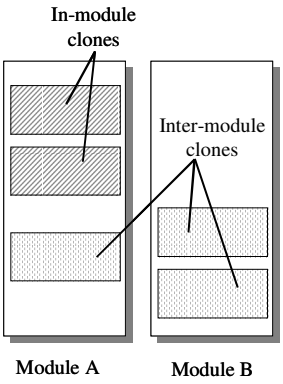


Figure 1. Two types of code clones

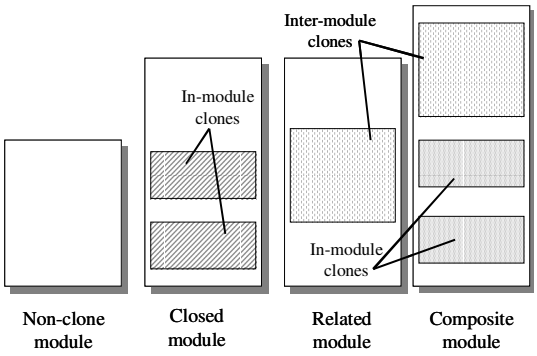


Figure 2. Module classification

3. DETECTION OF CODE CLONES

Recently, various kinds of clone detection techniques and tools have been proposed [1-6][11-15][17]. 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.

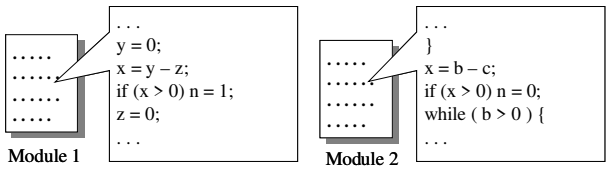
In this paper we used a token-based code clone detection technique proposed by Kamiya et al. [11] because their technique have industrial strength, and is applicable to a million-line size system within affordable computation time and memory usage. This technique is also easily applicable to legacy software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

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

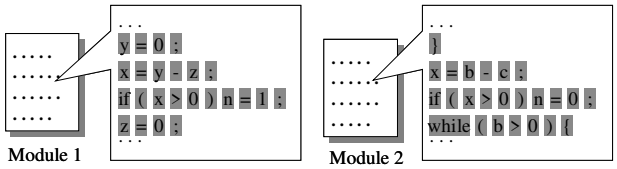
(1) Lexical analysis
All the source files are divided into tokens based on lexical rules of the programming language (Figure 3(a) and 3(b)). White spaces and comments are ignored in this analysis.

(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.

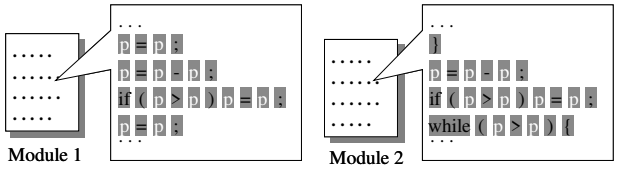
(3) Match detection and formatting
From all the sub-strings on the transformed token sequence, equivalent pairs are detected as clone pairs (Figure 3(d)). Then, each location of clone pair is converted into line number on the original source files.



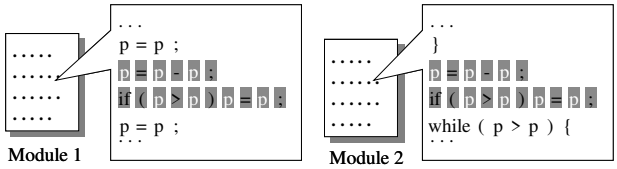
(a) Original source code



(b) Lexical analysis



(c) Transformation



(d) Match detection and formatting

Figure 3. Clone detection procedure

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 an old programming language, which is an expansion of COBOL.

4.2 What we measured

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

(1) LOC (Lines of code)
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 modules and clones. 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.

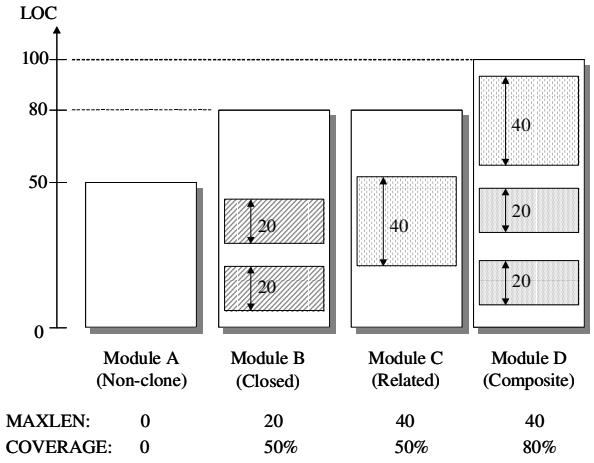


Figure 4. Example of modules and clones

5. RESULT OF EXPERIMENT

5.1 Module type and clone-code metrics

Figure 5 shows the classification of modules. About 50% of the modules are clone-included module. This result follows the previous research that Cobol payroll system had 59% code duplication [6]. 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.

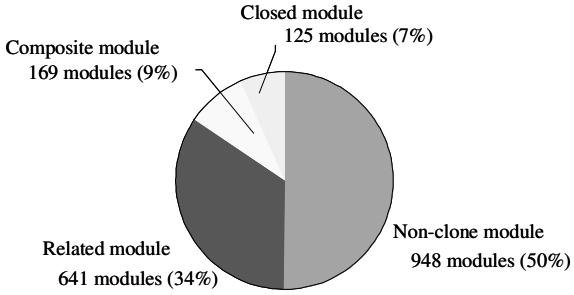


Figure 5. Classification of modules

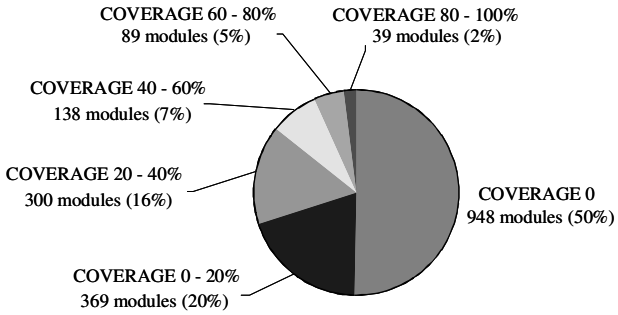


Figure 6. COVERAGE and the number of modules

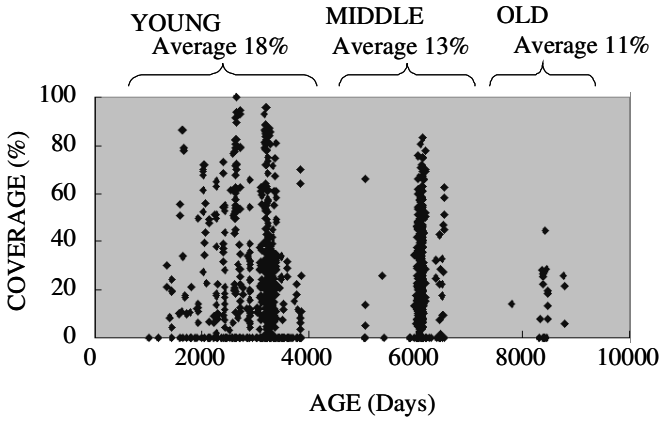


Figure 7. COVERAGE and AGE

5.2 Reliability analysis

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 the 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 in average. One possible interpretation for this result is that copying the code from trusted part can lessen the fault injection compared with writing the code from scratch.

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 in 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.

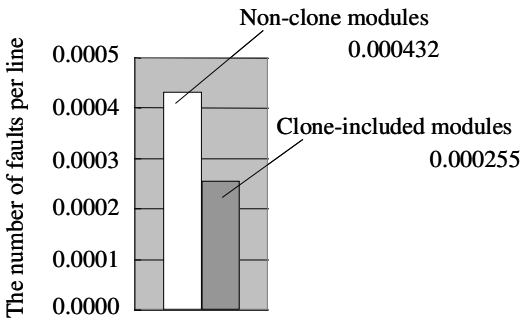


Figure 8. Relation between reliability and clones

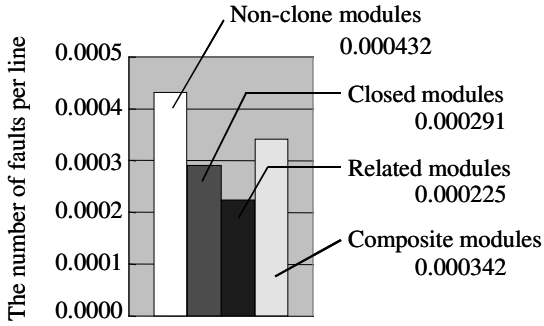


Figure 9. Reliability of different types of modules

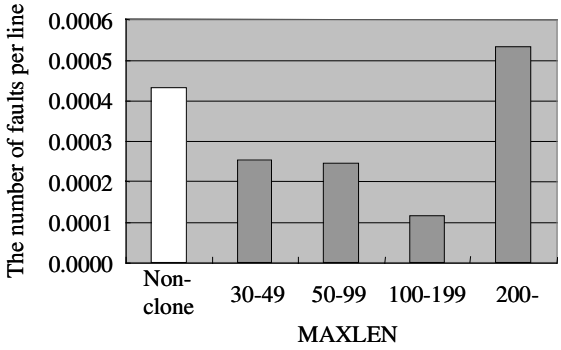


Figure 10. Relation between faults and MAXLEN

5.3 Maintainability analysis

In order to evaluate the maintainability of modules, we used the revision number as a maintainability measure. As we stated in Section 2.1, we consider a module having higher revision number is more difficult to be maintained than the module having lower revision number. 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 in 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.

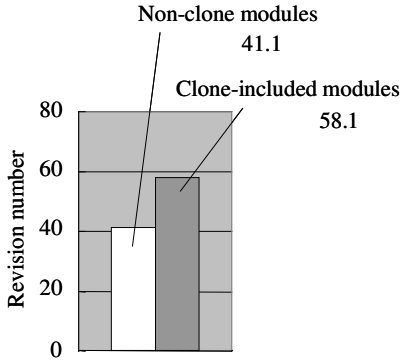


Figure 11. Relation between maintainability and clones

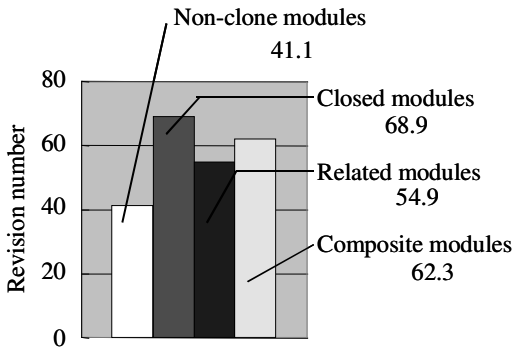


Figure 12. Maintainability of different types of modules

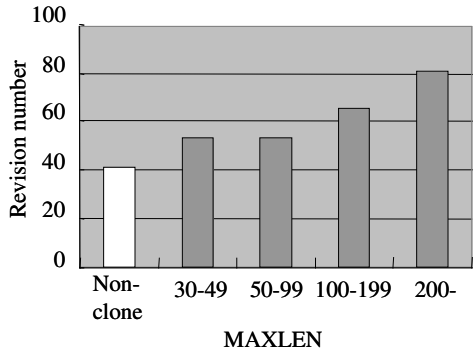


Figure 13. Relation between REV and MAXLEN

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 in average.
- Closed modules, related modules, and composite modules are all more reliable than non-clone modules in 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 than non-clone modules.
- Closed modules, related modules, and composite modules are all less maintainable than non-clone modules in average.
- The modules having larger code clone are less maintainable than modules having smaller code clone.

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.

In order to validate our result, we are planning to conduct the replicated experiment on other legacy systems.

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. Computing Science and Statistics: 24th Symposium on the Interface, 24, 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., 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. 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.
7. 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.
8. N. E. Fenton, “Software metrics: A rigorous approach,” Chapman & Hall, London, 1991.
9. M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts, “Refactoring: Improving the design of existing code,” Addison-Wesley, 1999.
10. M. H. Halstead, “Elements of software science,” Elsevier, New York, 1977.
11. 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.
12. 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.
13. 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.
14. 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.
15. 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.
16. M. M. Lehman and L. A. Belady, “Program evolution: Process of software change,” Academic Press, 1985.
17. 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.
18. T. J. McCabe, “A complexity measure,” IEEE Trans. on Software Engineering, Vol. 2, No.4, pp. 308-320, Dec. 1976.
19. A. Monden, S. Sato, K. Matsumoto, and K. Inoue, "Modeling and Analysis of Software Aging Process," International Conference on Product Focused Software Process Improvement (Profes2000), Lecture Notes in Computer Science, Vol. 1840, pp. 140-153, Springer-Verlag, June 2000.
20. 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.
21. 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, vol. 100, no. 570, SS2000-49, pp. 57-64 Jan. 2001 (in Japanese).
22. N. F. Schneidewind, and C. Ebert. “Preserve of redesign legacy systems?,” *IEEE Software*, Vol. 15, No.4, pp.14-17, July/Aug. 1998.
23. H. M. Sneed. Economics of software re-engineering, *Journal of Software Maintenance: Research and Practice*, Vol.3, No.3, pp.163-182, 1991.
24. H. M. Sneed. Planning the reengineering of legacy systems, IEEE Software, Vol.12, No.1, pp.24-34, Jan. 1995.