An Approach to Experimental Evaluation of Software Understandability

K. Shima[1], Y. Takemura[2], and K. Matsumoto[1]

[1]Nara Institute of Science and Technology
8916-5 Takayama, Ikoma
Nara 630-0101, JAPAN
+81-743-72-5312
shima@computer.org
matumoto@is.aist-nara.ac.jp

[2]Osaka University of Arts Junior College
2-14-19 Yata, Higashi-sumiyosi-ku,
Osaka 546-0023, Japan
takemura@mxw.mesh.ne.jp

Abstract

Software understandability is one of important characteristics of software quality because it can influence cost or reliability at software evolution in reuse or maintenance. However, it is difficult to evaluate software understandability in practice because understanding is an internal process of humans. This paper proposes "software overhaul" as a method for externalizing the process of understanding and presents a probability model to use process data of overhaul to estimate software understandability. An example describes an overhaul tool and an application of it.

Keywords: measurement, experimentation, human factors, reuse, maintenance, and evolution.

1. Introduction

Software reuse is promoted by object orientated technology or component-ware technology [1]. However, when developers try to reuse a software system developed by other developers, the difficulty of understanding the system limits reuse [3]. Even if the developers of the original system were in the same organization at first, they may be transferred, or may change their jobs or retire. It is not rare that changes to reused software systems will be needed for enhancing functions, correcting faults, or adapting them to new circumstances. If the developers of the original system were absent, the developers reusing it need to understand it. If it is difficult to understand, changes to it may cause serious faults and a chain reaction of changes. Such changes may cost more time than remaking the software system.

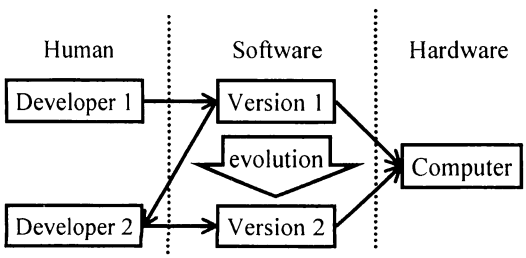

Fig. 1. Communications between human, software, and hardware in software evolution

Boehm defined software understandability as a characteristic of software quality which means ease of understanding software systems [2]. In his model, understandability is placed as a factor of software maintenance. Although developers of the original software system usually maintain it, they may be transferred, or change their jobs or retire. Software maintenance staffs need to understand and change it for enhancing functions, correcting faults, or adapting it to new circumstances. Changes to software systems are called software evolution in the research field of software maintenance. Changes to reused software systems can be considered as evolution of reused software systems. Therefore, software understandability can be placed as a factor of software evolution in reuse or maintenance. In an experiment of code inspection, 60% of issues which professional reviewers reported were soft maintenance issues related to understandability [6]. It means that professional reviewers regarded understandability as important.

However, it is not easy to measure software understandability because understanding is an internal process of humans. Fig. 1 shows communications among humans, software, and hardware in software evolution. Developer 1 writes version 1 of a software system. Developer 2 evolves the software system from version 1 into version 2. Software can be considered as the media of communications from the developers to the computer. Developer 2 reads version 1 in order to write version 2. Therefore, software can be considered as the media of communications from developer 1 to developer 2.

0-7695-1796-X/02 $17.00 © 2002 IEEE

Understanding a software system can be considered to be "reading necessary information via the software system to evolve it". Developer 2 may feel that he/she understood version 1 when he/she finished reading it. However, developer 2 might misunderstand version 1 and thus introduce faults into version 2. Therefore, we usually consider that developer 2 understands version 1 when he/she could correctly write version 2. In order to measure understandability, we need to observe an external process (like writing the version 2) which externalize understanding.

This paper proposes "software overhaul" as a method for externalizing process of understanding software systems. Overhaul itself does not change software systems. However, data from the overhaul process can be used to measure software understandability. This paper presents a probability model to use the process data to estimate understandability. An example describes an overhaul tool for source code and an application of it.

2. Software Overhaul

"Software overhaul" consists of deconstruction and reconstruction like overhaul of hardware systems e.g. engines, clocks, etc. Deconstruction is to take a software system apart to components. Reconstruction is to reproduce the software system by putting the components together again. Reconstruction simulates the construction which is to produce the original software system by selecting or making the necessary components and putting them together. In reconstruction, workers are given the same components of the original software system so that workers need not to select or make components. This constraint reduces the time needed for reconstruction.

```
/* overhaul the original software system and return the number of attempts
needed for correct reconstruct
  M : the number of components
  original : components of the original software system */
int overhaul ( int M , Component [ ] original ) {
  /* T : the number of attempts to reconstruct */
  int T = 0 ;
  /* deconstruct the original software system and get components. */
  Component [ ] components = deconstruct ( M , original ) ;
  do {
    ++ T ;
    /* reconstruct the software system from components */
    reconstruct ( M , components ) ;
  } while ( check ( M , components , original ) ) ;
  return T ;
}
/* deconstruct the software system and return removed components */
Component [ ] deconstruct ( int M , Component [ ] software ) {
  /* make a clone of the software system */
  Components [ ] components = clone ( M , software ) ;
  /* shuffle the components in order to hide the original arrangement. */
  shuffle ( M , components ) ;
  return components ;
}
/* reconstruct the software system from components */
void reconstruct ( int M , Component [ ] components ) {
  The worker rearranges the components.
}
/* check the reconstructed software system with the original, removed correct
components from the reconstructed software system, and return the number of
differences */
int check ( int M , Component [ ] reconstructed , Component [ ] original ) {
  int k , differences = 0 ;
  for ( k = 0 ; k < M ; ++ k ) {
    if ( reconstructed [ k ] != NULL ) {
      if ( reconstructed [ k ] = = original [ k ] )
        reconstructed [ k ] = NULL ;
      else ++ differences ;
    }
  }
  return differences ;
}
```

Table 1. A procedure of software overhaul

Workers use a tool to "overhaul". The tool deconstructs the original software system and checks the software system reconstructed by workers. When the tool checks the reconstructed software system, it fixes components in the same place with the original so that the workers use only remaining components at the next reconstruction. Therefore, workers can overhaul by trial and error. Table 1 shows a procedure of software overhaul written in a language like C.

Software overhaul is a new method for externalizing process of understanding software systems. Dunsmore and Roper reviewed many papers and listed maintenance task, recall (memorization), subjective rating, label/group code, fill-in-blank (cloze), code overage/optimization, and call graph as techniques used to measure comprehension [4]. Most of them (except subjective rating) can be considered as methods for externalizing process of understanding software systems. Maintenance task are easy to be believed because the task represents what programmers do. However, this method requires costs of preparation, execution, and analysis. In the other methods, researchers suppose some knowledge which subjects who understood the software system should have or can infer. Subjects execute a task which requires such knowledge in order to achieve it. Software overhaul is a method of the latter group. Experiments which show the link between various maintenance tasks and software overhaul are needed. However, this paper focuses on the theoretical part. Another paper [10] presents experiments which show the link between debug task and software overhaul.

3. Model

When a worker needed to reconstruct one software system many times until he/she correctly reconstructs it, it can be considered the software system is difficult for him/her to understand. Needless to say, understanding depends on not only understandability of the software system, but also comprehension of the worker. If many workers overhauled many software systems, the average number of attempts needed for correct reconstruct can be a metric of understandability or comprehension. The average number of attempts needed for correct reconstruction that one worker reconstructed many software systems means comprehension of the worker. The average number of attempts needed for correct reconstruction that many workers reconstructed one software system means understandability of the software system. However, if the amount of data is small, such average number does not carry high confidence as an estimator. This section presents probabilistic models to estimate comprehension and understandability. The followings are given.

L : the number of workers

N : the number of software systems

M_n : the number of components of the software system n (n=1~N).

${}_lT_n$: the number of attempts to reconstruct when the worker l overhauled the software system n (l=1~L, n=1~N).

3.1. Random Reconstruction

Some workers may randomly reconstruct just by trial and error when they can not understand the software system because the workers are not good at comprehending or the software system is not well-understandable. Let us define:

H_R : the hypothesis that the worker randomly rearranges all components of the software system in reconstructing.

$f_M(T)$: the probability that the worker correctly rearranges M components at the T attempts under .

${}_M P'_k = {}_M C_k \times P''_k$: the number of permutations of the M components in which k components are different from the original permutation and the other (M-k) components are the same with the original permutation.

$P''_M = {}_M P'_M$: the number of permutations in which all M components are different from the original permutation.

The following equations can be derived.

$f_0(0) = 1$.

$f_M(0) = 0$ when $M > 0$.

$f_0(T) = 0$ when $T > 0$.

${}_M P'_0 = P''_0 = {}_0 P'_0 = 1$.

${}_M P'_k = {}_M C_k \times P''_k$.

When the worker rearranged M components and k of M components are different from the original software system, he/she rearranges k components at the next attempt to reconstruct. Therefore,

$$f_M(T) = \frac{1}{M!} \sum_{k=0}^{M} {}_M P'_k f_k(T-1)$$

when $M > 0$ and $T > 0$.

${}_M P'_k$ and P''_M can be calculated as follows:

$$\sum_{k=0}^{M} {}_M P'_k = M!.$$

$$ {}_M P'_M = M! - \sum_{k=0}^{M-1} {}_M P'_k \text{ when } M > 0 . $$

$$ P''_M = M! - \sum_{k=0}^{M-1} {}_M C_k \times P''_k \text{ when } M > 0 . $$

3.2. Significance Test of Understanding

In order to confirm that the worker did not randomly reconstruct the software system, $f_M(T)$ can be used to statistically test H_R as follow:

T : the observed number of attempts to reconstruct.

t : the random variable of attempts to reconstruct.

$F_M(T) = P(t \le T \mid H_R) = \sum_{t=0}^{T} f_M(t)$: the probability that the worker correctly rearranges M components within T attempts.

α : the significance level such as 0.05, 0.01, 0.005, or 0.001.

For example, when $F_M(T) \le \alpha$, H_R is significantly rejected and it means probably $\overline{H}_R$. When $F_M(T) > \alpha$, H_R is accepted. However, it is not significant. That is, it does not mean that H_R is proved. This relationship is described as follows:

$$ P(H_R \cap t \le T) = P(H_R \mid t \le T) P(t \le T) $$
$$ = P(t \le T \mid H_R) P(H_R) $$

$$ P(H_R \mid t \le T) = \frac{F_M(T) P(H_R)}{P(t \le T)} . $$

If a worker could overhaul a software system within T attempts, he/she can usually overhaul the same software system within T attempts at the next time because he/she can remember the original software system. Therefore, it can be considered $P(t \le T) = 1$. When the worker overhaul the software system many times, $P(H_R)$ will decrease because he/she remembers the original software system. However, it is difficult to estimate $P(H_R)$ at the first overhaul. Therefore, we use $P(H_R) \le 1$ to derive the following inequality.

$$ P(H_R \mid t \le T) \le F_M(T) . $$

$$ P(\overline{H}_R \mid t \le T) = 1 - P(H_R \mid t \le T) \ge 1 - F_M(T) . $$

Therefore, if $F_M(T)$ is small, the probability of $\overline{H}_R$ is large. It means that the worker could understand the software system at least a little. However, even if $F_M(T)$ is large, maybe H_R or $\overline{H}_R$.

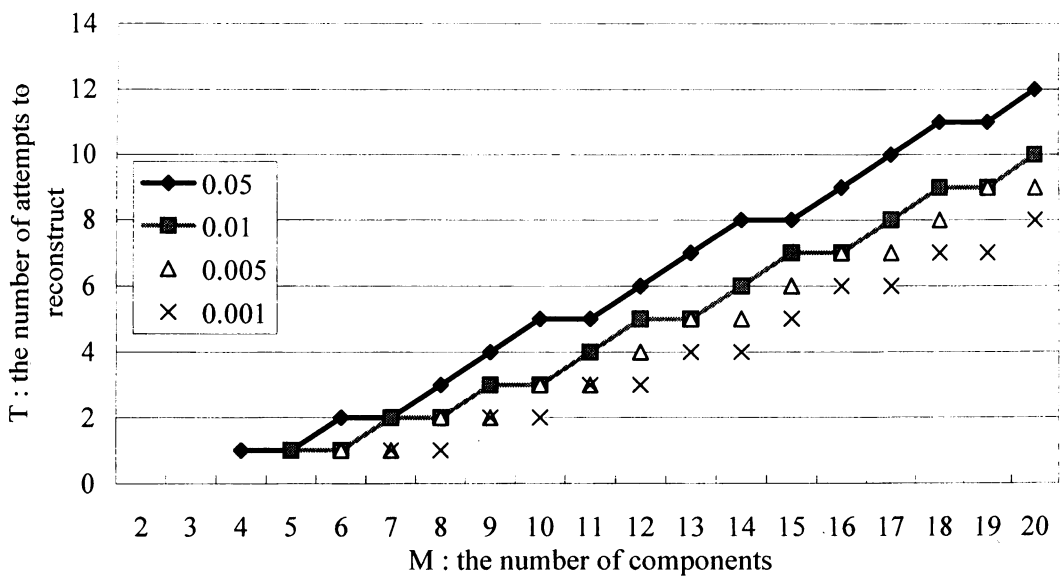

Fig. 2 The maximum number of attempts to reconstruct with significance

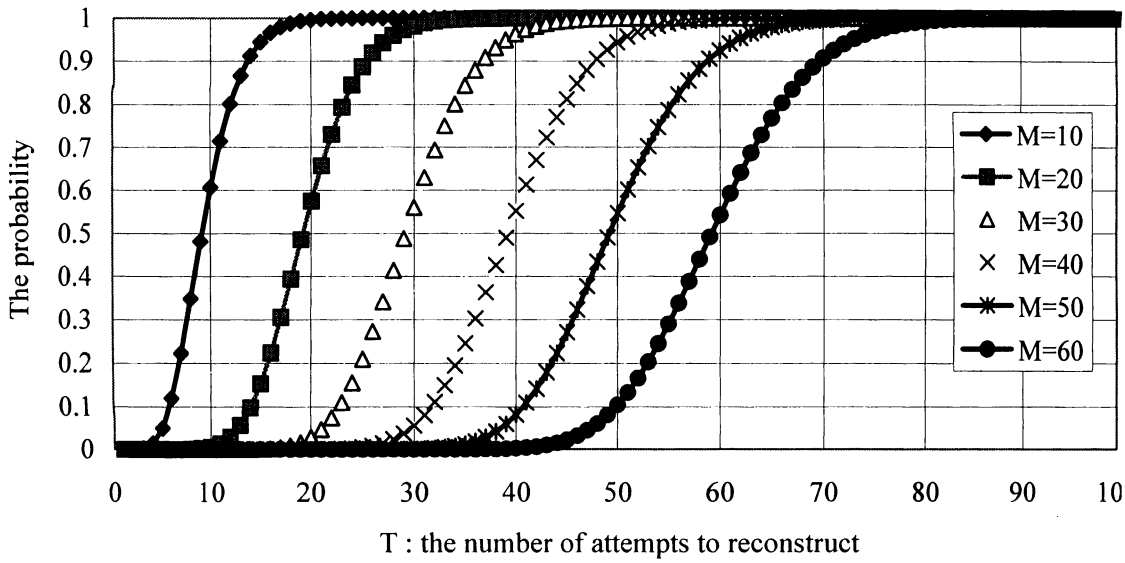

Fig. 3 The number of attempts to reconstruct vs the probability

$T_{\max}(M) = \max\{^{\forall}T \mid F_M(T) \le \alpha\}$: The maximum number of attempts to reconstruct of which $F_M(T)$ is less than the significance level α.

Fig. 2 shows $T_{\max}(M)$ when α=0.05, 0.01, 0.005, or 0.001. The horizontal axis shows the number of components. The vertical axis shows the number of attempts to reconstruct. If $T_{\max}(M) = 0$, the results of overhaul can never be significant because the number of attempts to reconstruct is one at least. If $T_{\max}(M) = 1$, the comparison of results is meaningless because the number of attempts to reconstruct is always one when it is significant. Therefore, $T_{\max}(M)$ should two at least. It means that 6, 7, 8, or 9 components are required for the significance levels 0.05, 0.01, 0.005, or 0.001, respectively.

If workers needed to rearrange all components in every attempts, the probability that workers succeed to correctly reconstruct is 1/*M*! in every attempts, and then, it is too difficult for them to succeed to correctly reconstruct within practical attempts. However, even if the workers randomly rearrange the components, the number of components that they need to rearrange will decrease because the tool fixes components in the same place with the original in each attempt. Fig. 3 shows the feasibility of overhaul. The horizontal axis means *T*. The vertical axis means $F_M(T)$. $F_M(T)$ is more than 0.9 at *T*=14, 26, 37, 48, 59, or 70 when *M*=10, 20, 30, 40, 50, or 60, respectively. Although workers may be tired if they repeated to reconstruct in such number of times, they can succeed at final.

3.3. Significance Test for Multiple Overhauls

Needless to say, the question whether the worker can understand the software system depends on not only understandability of the software system, but also comprehension of the worker. Therefore, experimenters may assign one software system to multiple workers for accurate measurement. Suppose the result of only one worker rejected H_R and the results of others accepted H_R. The experimenter can logically think that H_R is rejected when the number of workers is small. However, if 20 workers randomly reconstructed the software system, one lucky worker may reject H_R with the significance level 0.05.

The Kolmogorov-Smirnov one-sample test is a test of goodness-of-fit [8]. It is concerned with the degree of agreement between the distribution of a set of sample values (observed scores) and some specified theoretical distribution. Therefore, it can be used to determine whether results in overhaul by multiple workers can reasonably be thought to have come from a population having the theoretical distribution under H_R. The tested hypothesis is that H_R for all workers. The Kolmogorov-

Smirnov test focuses on the maximum deviation D as follows:

$$D = \max |F_{M_n}({}_lT_n) - \frac{{}_lS_n}{L}| \text{ where } n = 1,2,...,N,$$

$l = 1,2,...,L$, and ${}_lS_n$ is the number of workers whose the number of attempts to reconstruct were equal to or less than ${}_lT_n$.

The sampling distribution of D under the hypothesis is known (for example, see [8]). If the observed D is more than the sampling value, the hypothesis is rejected. It means that some of workers can understand the software system.

4. Example

Software systems consist of computer programs and documents that describe planning, specifications, designs, testing, etc. Programs written in programming languages are called source code. This section describes "overhaul" of source code as an example of "software overhaul". In overhaul of source code, workers need to understand specifications or designs of the program written in documents or comments in source code in order to reconstruct source code. Therefore, overhaul of source code evaluates not only the source code, but also such documents.

Source code consists of characters. New line characters separate other characters into lines. Although new line characters mean white spaces in most of recent programming languages, most developers choose new line characters and white spaces to make source code easy to read. Developers usually separate characters into lines by statement that is a unit of executions. Therefore, we selected executable lines as components of source code.

4.1 An Overhaul Tool

We developed an overhaul tool for source code. This tool consists of a client and a server which are written in Java. The server is one of WWW servers so that workers can use WWW browsers to access it. At first, workers open a home page on the server. The home page contains the client as a Java applet so that the WWW browsers download and execute it. Therefore, it does not need to install the client into workers' computers in advance.

Workers can access the server any time and any where even when experimenters are absent. Therefore, this tool has a simple login session in order to distinguish workers. The client asks workers to register their personal data or to enter their name and ID. The personal data are the name, birth year, job, etc. which are needed to know the characteristics of workers. When workers have registered, the client sends the personal data to the server. The server assigns an ID and replies it to the client. The client shows

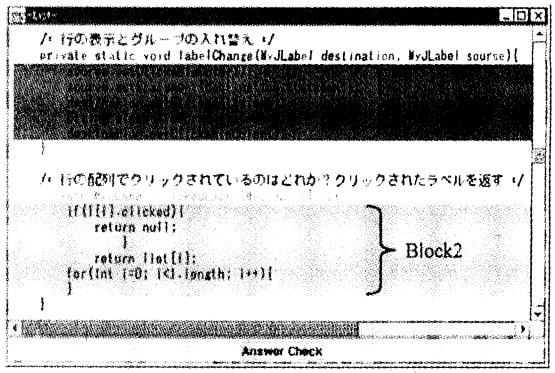

Fig. 4 An overhaul tool for source code

the ID. When workers entered their name and ID, the client sends them to the server. The server checks them with the registered data and replies the result. If the name and ID are the same with the registered data, the client starts overhaul session. If not, the client ask workers their name and ID again.

At the first of overhaul session, the client asks the server to send a file of source code. The server randomly selects a file from files of the software system and sends it to the client. The client shuffles lines of the file and shows them. Fig. 4 shows a window of the client. This window shows two blocks (Block1 and Block2). Each block consists of executable lines in one function. Gray lines (of which the background is gray) are shuffled in each block. When workers click two of gray lines, the two lines are exchanged. When workers click the 'Answer Check' button at the bottom of the window, the client checks gray lines that workers rearranged with the original lines. The client makes gray lines that are the same with the original lines white. Gray lines that are different from the original lines remain. The client sends the number of times when workers clicked the 'Answer Check' button to the server.

4.2 Experiment

We conducted an experiment to apply our tool to a program developed in an industry. The program provides a language-oriented user interface which allows the user to describe the configuration of an array of antennas using a high level language. The program was developed for the European Space Agency (ESA) in the C language within Microsoft Visual C++ 1.5 environment. The program consists of almost 10,000 lines of code (6,100 executables) and is organized in three subsystems of parser, computation, and formatting. 21 files (11 files without faults and 10 files with faults which detected in testing) are randomly sampled from all 150 files. They consist of 1801 lines of code (549 executables, 90 lines per file). Subjects are five engineering students (one graduate and four undergraduates). They studied syntax of C++ programming language such as control statements,

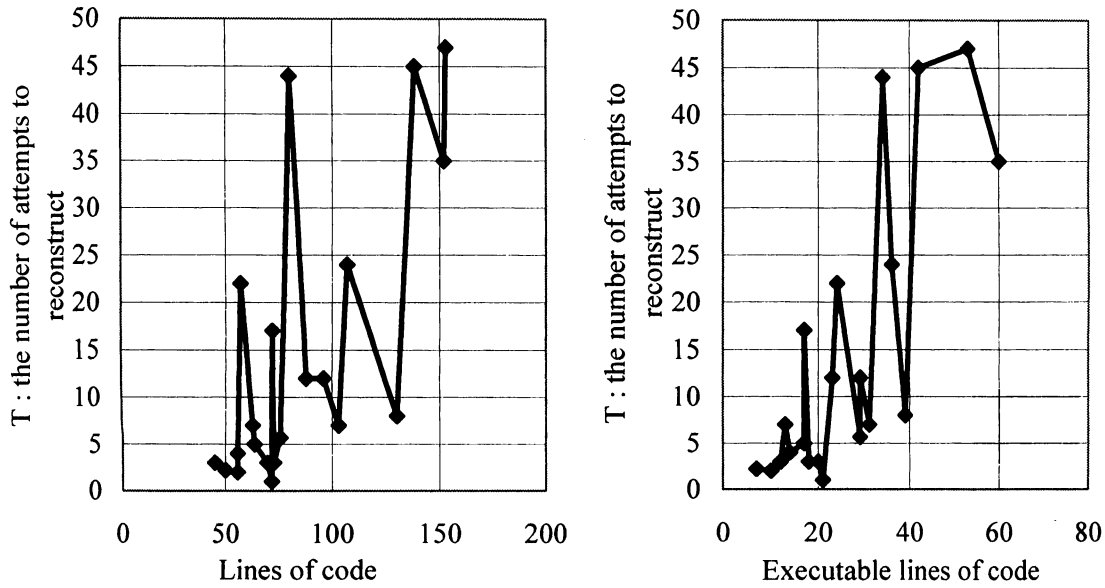

Fig. 6 The observed number of attempts to reconstruct

structure, pointer, etc. and they were doing their exercise in their course.

As a result of the experiment, it took subjects 88 min/KLOC. Therefore, it can be estimated that it will take them 15 man x hours to overhaul the whole program (10,000 lines). The average and standard deviation number of attempts to reconstruct was 11 and 14, respectively. Fig. 5 shows the observed number of attempts to reconstruct and the number of lines. The number of attempts to reconstruct depends on the number of executables rather than lines of code. The correlation coefficient between the number of attempts to reconstruct and executables is 0.77. The correlation coefficient between the number of attempts to reconstruct and lines of code is 0.69. This figure shows that it is extremely difficult to rearrange more than 40 executables.

Fig. 5 shows a comparison of executables and $F_M(T)$ between the faulty files and non-faulty files. Faulty files are uneven distribution although unfortunately they are not statistically significant because the number of sampling is not enough. The size of all files is from 7 to 60 and the size of faulty files is from 12 to 42 although the number of faulty files and non-faulty files are similar (10 vs 11). It may imply that developers carefully reviewed large files

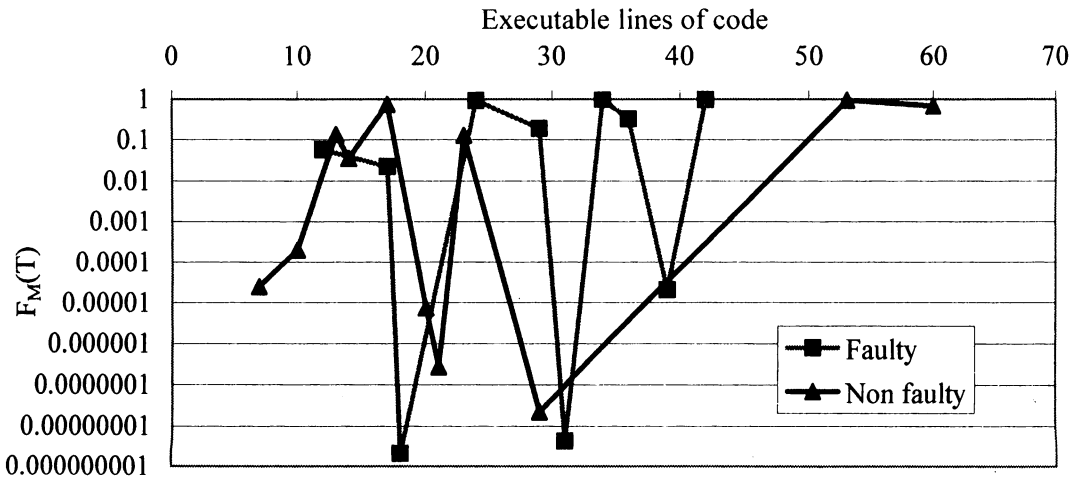

Fig. 5 Size and understandability between faulty file and non-faulty files

(like over 50 executables) and small files do not contain faults from the first. On the other hand, Faults frequently appear in files whose $F_M(T)$ is low or high except. One of six file has faults when $F_M(T)$ is the mid range from 0.00000001 to 0.01. It may imply that developers easily missed faults when the files are difficult to understand, however the developers sometimes missed faults when they feel the files are very easy to understand.

5. Conclusion

This paper presented software overhaul as an approach to externalize the process of understanding software systems, the probabilistic models to evaluate understandability, an overhaul tool for source code, and an experiment as an example of overhaul.

The number of attempts to reconstruct that workers repeated to reconstruct a software system is defined as a process metric of understandability under the algorithm of software overhaul. In a research of a large scale software system, some process metrics were more useful to predict fault incidence than most product metrics [5]. One of the reasons can be human factors. For example, source code developed in companies included one comment line for each executable line. It means that developers feel comments are important. However, product metrics do not evaluate comments because it is difficult for computer to evaluate comments written in natural languages yet. Process metrics such as the number of changes to the software system can incorporate human factors because human behaviors influence them.

The probabilistic models presented in this paper translate the number of attempts to reconstruct into the probability of understanding that is comparable among different software systems. If we could enumerate all necessary knowledge items of the software system, we can test whether developers know it or not after they read the software system. However, this is usually impossible because we have no systematic method for finding such knowledge items or confirming that the enumerated knowledge items are enough. Therefore, probabilistic models are needed for estimation.

The overhaul tool we developed consists of a Java applet and a WWW server. Therefore, there are the following characteristics.

1. Experimenters do not need to install the tool on each computer of workers in advance.
2. Workers can overhaul anywhere not only the laboratory, but also their office or home.
3. Experimenters do not need to take a software system away from the organization of developers if they installed the server in the organization.

It is usual architecture itself as software systems. However, this paper shows a possibility of application of the architecture to computer aided empirical software engineering [9].

Acknowledgment

Our thanks to Prof. Bev Littlewood, Prof. Lorenzo Strigini, Dr. Diana Bosio and Mr. David Styles in City University for help to us with the experiment, to Dr. Andrey Povyakalo, Dr. Peter Popov, Dr. Eugenio Alberdi, Dr. Mourad Oussalah, and Dr. Mark Alexander in City University for valuable comments, and to the subjects for joining the experiment.

Parts of this work were funded through a grant from National Space Development Agency, Japan and a grant (No. 14780324) from Japanese Government.

References

[1] M. Aoyama, "Component-based software engineering: can it change the way of software development?" Proc. of the 20th International Conference on Software Engineering, vol. 2, pp. 24-27, 1998.

[2] B. W. Boehm, et. al, "Characteristics of Software Quality," North-Holland, 1978.

[3] G. Caldiera, and V. R. Basili, "The qualification of reusable software components," pp. 117-119 in [7].

[4] A. Dunsmore and M. Roper, "A Comparative Evaluation of Program Comprehension Measures", The Journal of Systems and Software, Volume 52, Issue 3, pp. 121-129, June 2000.

[5] T. L. Graves, A. F. Karr, J. S. Marron, and H. Siy, "Predicting fault incidence using software change history," IEEE Transactions on Software Engineering, vol. 26, no. 7, pp. 653-661, July 2000.

[6] A. A. Porter, H. P. Siy, C. A. Toman, and L. G. Votta, "An experiment to assess the cost-benefits of code inspections in large scale software development," IEEE Transactions on Software Engineering, vol. 23, no. 6, June 1997.

[7] W. Schafer, R. Prieto-diaz, and M. Matsumoto, ``Software Reusability,'' Ellis Horwood Limited, pp. 117, 1994.

[8] Sidney Siegel, N. John Castellan, Jr., Nonparametric Statistics for the Behavioral Sciences (second edition), McGRAW-HILL Inc., ISBN 0-07-057357-3, 1988.

[9] Koji Torii, Ken-ichi Matsumoto, Kumiyo Nakakoji, Yoshihiro Takada, Shingo Takada, and Kazuyuki Shima, "Ginger2: an environment for CAESE (computer-aided empirical software engineering)," IEEE Transactions on Software Engineering, vol. 25, no. 4, pp. 474-492, Aug. 1999.

[10] Shinji Uchida, Kazuyuki Shima, Makoto Sakai, Ken-ichi Matsumoto, "An experiment to evaluate software overhaul method," (in Japanese) Software Symposium 2002, pp. 116-121, July 16-19, 2002.