An Experiment of Evaluating Software Understandability

Shinji UCHIDA
Electrical and Information Engineering, Kinki University Technical College,
2800 arima, kumano, Mie 519-4395, Japan

and

Kazuyuki SHIMA
Department of Computer Science, Hiroshima City University
3-4-1 Ozuka-Higashi, Asa-Minami-Ku, Hiroshima, 731-3194, Japan

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. But it is difficult to evaluate software understandability because understanding is an internal process of humans. So, we propose “software overhaul” as a method for externalizing process of understanding software systems and propose a probability model for evaluating software understandability based on it. This paper presented the experiment of evaluating software understandability using a probabilistic model.

Keywords: Software Understandability, Software Maintenance,

1. INTRODUCTION

Software reuse is promoted by object orientated technology or component-ware technology [1]. However, the difficulty of understanding the system limits reuse when developers try to reuse a software system developed by other developers [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.
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.
We propose “software overhaul” as a method for externalizing process of understanding software systems [9]. Overhaul itself does not change software systems. However, data from the overhaul process can be used to measure software understandability. This paper presents an experiment of evaluating software understandability using a probability model. We provide 20 modules (10 faulty modules and 10 non-faulty modules) in the same software for overhaul. The result of analysis using our model, we clarify that faulty modules are worse understandability than non-faulty modules.

2. SOFTWARE OVERHAUL

Software overhaul consists of deconstruction and reconstruction like overhaul of hardware 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

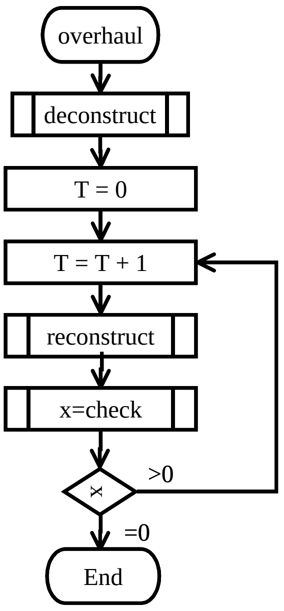


Fig.1 The procedure of overhaul

original software system so that workers need not to select or make components. This constraint reduces the time needed for reconstruction. 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. Fig. 1 shows a procedure of software overhaul.

3. PROBABILISTIC 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 reconstructing when the worker l overhauled the software system n (l=1~L, n=1~N).

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 reconstructing under H_R .
${}_MP'_k = {}_MC_k \times P''_k$: 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.
$P''_M = {}_MP'_M$: 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 $T > 0$.
${}_MP'_0 = P''_0 = {}_0P'_0 = 1$.
${}_MP'_k = {}_MC_k \times P''_k$.
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,

$f_M(T) = \frac{1}{M!}\sum_{k=0}^{M} {}_MP'_k f_k(T-1)$ when $M > 0$ and $T > 0$.

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

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

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

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

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 reconstructing.
t : the random variable of reconstructing.

$F_M(T) = P(t \le T \mid H_R) = \sum_{t=0}^{T} f_M(t)$: the probability that the worker correctly rearranges M components within T reconstructing.
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. Therefore, probably $\overline{H}_R$. If $F_M(T) > \alpha$, H_R is accepted. However, it is not significant. That is, it does not mean that 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 reconstructing, he/she can usually overhaul the same software system within T reconstructing 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 maybe $\overline{H}_R$.

$T_{\max}(M) = \max\{{}^{\forall}T \mid F_M(T) \le \alpha\}$: 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 =0.05, 0.01, 0.005, or 0.001. The horizontal axis shows the number of components. The vertical axis shows the number of reconstructing. If $T_{\max}(M) = 0$, the results of overhaul can never be significant because the number of reconstructing is one at least. If $T_{\max}(M) = 1$, the comparison of results is meaningless because the number of reconstructing 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 attempt, 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 fix components in the same place with the original in each attempt. Fig. 3 shows the

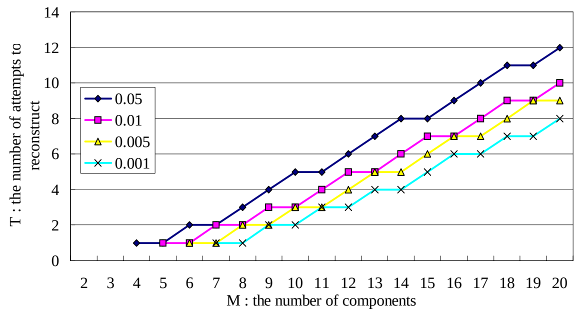


Fig2. The maximum number of attempts to reconstruct with significance

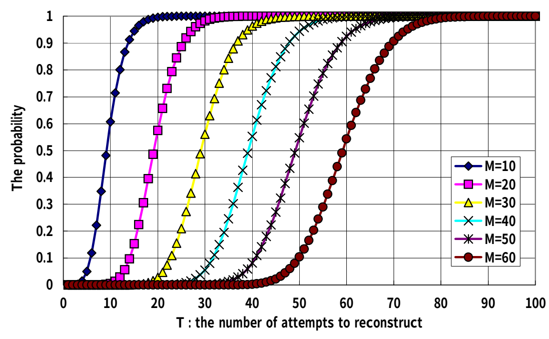


Fig. 3 The number of attempts to reconstruct vs the probability

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.

SIGNIFICANCE TEST FOR MULTIPLE OVERHAULS

Needless to say, whether the worker can understand the software system or not 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 not large. 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 [7]. 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:

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

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

The sampling distribution of D under the hypothesis is known (for example, see [7]). 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. EXPERIMENT

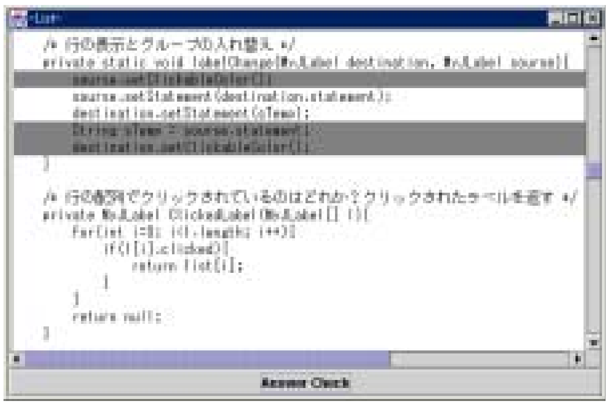
In order to evaluate software understandability using our model, we conducted an experiment. In the experiment, at first, the subjects were given the source code and the documents. Then they started to carry out the overhaul using overhaul tool.

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 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 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. Gray lines (of which background are gray) are shuffled. 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.



```
/* 行の表示とグループの入れ替え */
private static void labelChange(MyJLabel destination, MyJLabel source){
        source.setClickableColor();
        source.setStatement(destination.statement);
        destination.setStatement(sTemp);
        String sTemp = source.statement();
        destination.setClickableColor();
}

/* 行の配列でクリックされているのはどれか？クリックされたラベルを返す */
private MyJLabel ClickedLabel(MyJLabel[] l){
        for(int i=0; i<l.length; i++){
                if(l[i].clicked){
                        return list[i];
                }
        }
        return null;
}
```

Fig.4 An overhaul tool for source code

SUBJECTS
73 subjects participated in the experiment and were assigned to carry out the overhaul independently. All subjects are graduate school student. We separated the subjects into two groups. One group (46 subjects) overhauled the non-faulty module. Another one (27 subjects) overhauled the faulty module.

TARGET PROGRAM
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 10,000lines of code (6,100 executables) and is organized in three subsystems of parser, computation, and formatting. This program consists of 139 modules and includes 33 faults. We choose 20 modules in the program (10 faulty modules and 10 non-faulty modules).

5. RESULT

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 modules that have less than 30 executable lines of code are almost non-faulty module (42 non-faulty modules and 12 faulty modules). On the other hand, the modules that have more than 30 executable lines of code are faulty module (3 non-faulty modules and 14 faulty modules). This figure shows that the fault is included in the module that has longer executable lines of code.

Fig. 6 shows a comparison of executables and $F_M(T)$ between the faulty files and non-faulty files.
In less than 10 executable lines of code, all modules are non-faulty module. From 10 to 30 executable lines of code, the modules of which understandability is low are almost faulty modules. In more than 30 executable lines of code, almost modules are faulty module. This figure shows faulty modules are worse understandability than non-faulty modules.

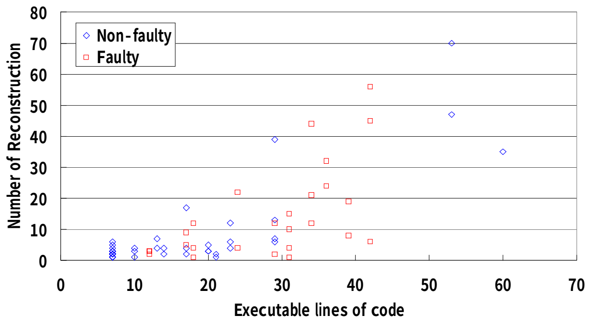


Fig. 5 The observed number of attempts to reconstruct

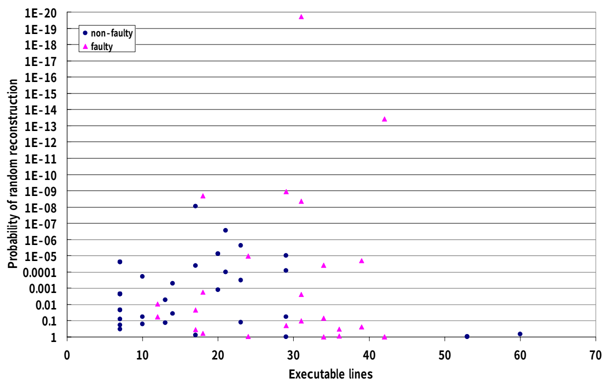


Fig. 6 Size and understandability between faulty modules and non-faulty modules

6. RERATED WORKS

There are some techniques used to measure software understanding, such as code review, Recall and Fill-in-the-blank.
Code review is static analysis aimed at identifying fragments of code consistently associated with faulty behavior. However, reviewer must know beforehand what kind of source code is how unclear by development experience or training.
In general, recall tests usually involve presenting a subject with a segment of code and allowing them to study it for an allotted time [4]. Once this time is over, the code is removed/hidden and subjects are asked to recall as much of the code as possible. In some cases, both of these steps are repeated several times, in others subjects are also allowed to modify if they think it necessary, their previous attempts at recalling the code. Software overhaul does not have the necessity of learning about software beforehand. Recall is dependent on a subject's memory. So, Software overhaul differs from Recall.

Fill-in-the-blank usually involves presenting subjects with a piece of code with a line missing [4]. The code presented in the experiment in [10] had not been previously seen by the experiments subjects, who had to fill in a single blank line in the program. In the software overhaul, a subject is shown the portion of a source code in the state where it was arranged at random. In Fill-in-the-blank, source code is shown which corrects by making a specific portion blank.

7. CONCLUSION

This paper presented the experiment of evaluating software understandability using a probabilistic model. In the experiment, we provide 20 Modules (10 faulty modules and 10 non-faulty modules) in the same software for overhaul. The result of experiment, we clarify that faulty modules are worse understandability than non-faulty modules. In the future, we are planning to conduct further experimental investigation based on the model.

8. ACKNOWLEDGMENT

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

9. 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, 1998, pp. 24-27.
[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 [8].
[4] A. Dunsmore and M. Roper, "A Comparative Evaluation of Program Comprehension Measures", **The Journal of Systems and Software**, vol. 52, no. 3, 2000, pp. 121-129 .
[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, 2000, pp. 653-661.
[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, 1997, pp. 329-346.
[7] Sidney Siegel, and N. John Castellan, Jr., **Nonparametric Statistics for the Behavioral Sciences** (second edition), McGRAW-HILL Inc., ISBN 0-07-057357-3, 1988.
[8] W. Schafer, R. Prieto-Diaz, M. Matsumoto, **Software Reusability**, Ellis Horwood Limited, 1994, pp.117.
[9] K. Shima, Y. Takemura, and K. Matsumoto, "An approach to experimental evaluation of software understandability," **Proc. of International Symposium on Empirical Software Engineering** (ISESE2002), IEEE Computer Society Press, 2002, pp.48-55.
[10] E. Soloway and K. Ehrlich, "Empirical Studies of Programming Knowledge", **IEEE Transactions on Software Engineering**, Vol. SE-10, No. 5, 1984, pp. 595-609.