Quantitative Evaluation of Software Reviews and Testing Processes

Shinji KUSUMOTO

September 1993

Quantitative Evaluation of Software Reviews and Testing Processes

Shinji KUSUMOTO

September 1993

Dissertation submitted to the Faculty of the Engineering Science of Osaka University in partial fulfillment of the requirements for the degree of Doctor of Engineering

Abstract

Increasing the productivity and quality of software development processes has been an important research objective in software engineering. This thesis proposes a metric and a framework to improve the productivity and quality of software development from two perspectives; (1)taking the narrow view, a metric, M_k is proposed and tested for modeling the review and testing processes in order to quantitatively analyze their effectiveness, and (2) a framework is designed and tested for mathematically modeling software development processes in order to improve the development processes themselves.

Putting more emphases on testing has been recognized to be one of the most effective approaches. It is reported that most companies spend between 50-80% of their development cost on testing. Therefore, reducing the cost of testing is a key factor for increasing productivity in software development. Software reviews are one of the most effective techniques for reducing the cost of testing. But we do not have enough knowledge on how to do software reviews in order to reduce the testing cost drastically. Especially, there are few methods for evaluating and deciding if a current software review is going well with respect to the testing cost. Some metrics have already been proposed, but most of these calculate only the number of faults in software products and the number of faults which are detected by software reviews. Thus, metrics need to be developed that can evaluate and prove the effectiveness of software reviews with respect to software development cost.

This thesis proposes a new metric, M_k, for evaluating cost effectiveness of software reviews. It is based on the degree to which cost to detect and remove all faults from the software in a project are reduced by technical reviews. We present experimental evaluations of conventional metrics and the proposed metric M_k using data collected in an industrial environment. These data illustrate the validity and usefulness of the proposed metric M_k. If we obtain data from software reviews in many projects, it may also be possible to control review activities effectively and to improve productivity in software development projects.

In order to improve the quality of the final software product, we must improve the quality of the software product throughout the development process (requirement analysis, design and implementation) in addition to spending much time testing. For realizing improvements, it is necessary to (1) understand and analyze the current status of the software development process, and (2) execute the improvement plan of the process based on this analysis. In order to realize the most effective improvement, we must rely on quantitative measurements to objectively control software development processes. However there are few studies that present a formal method that is useful for systematically measuring software development processes.

This thesis proposes a new framework for measuring software development processes. The key idea of the proposed framework is that all activities to be measured can be defined based on a mathematical description of the process to be measured. The proposed framework consists of four steps: (1) process modeling, (2) metric definition, (3) process and metric implementation, and (4) process and metric execution. Process modeling transforms the software development process to be measured into a Petri net model which represents essential features of the process. Metric definition clarifies how to evaluate the features of the process in the model. Process and metric implementation means to enact the process based on the model, and collect data from the process, and compute the metric values based on the model. Process and metric execution

means to carry out the process, and to compute data collected from the process.

Finally, in order to present an application of the proposed framework a coding and debugging process in a student project in an academic environment was modeled using a Petri net, and four metrics for evaluating the debugging activities were defined. Data were collected from two successive experiments based on the model and were used to evaluate the effects of the design method improving debugging activities. The study showed that the framework can make it possible to measure software development processes in a systematic way and make it easy to interpret the experimental results.

In this dissertation, Chapter 1 briefly summarizes related progress and topics in software engineering and describes outline of the thesis.

In Chapter 2, we define software development processes, software reviews, and software testing to be discussed in this thesis.

Chapter 3 presents three conventional metrics: M_m by Myers, M_f by Fagan, and M_c by Collofello, for evaluating the effectiveness of technical reviews. Then, a new metric, M_k, is introduced for evaluating the cost effectiveness of technical reviews.

Chapter 4 shows an experimental evaluation of the metrics M_m, M_f, M_c and M_k using the data collected in industrial project. Experimental results are summarized to show the superiority of the M_k metric.

Chapter 5 describes an application of the M_k metric comparing three methods for allocating review effort.

Chapter 6 shows four major activities and the potential capability of the framework for measuring the software development processes and describing testing activities in an academic environment.

Chapter 7 presents an application of the framework, in which we tried to evaluate the effects of the design method in an academic environment improving testing activities.

Chapter 8 summarizes the main results of this thesis and presents some areas for future research.

List of Major Publications

(1) S. Kusumoto, K. Matsumoto, T. Kikuno and K. Torii : "On a measurement environment for controlling software development activities," *IEICE Transactions on Communications Electronics Information and Systems,* Vol.E 74, No.5, pp.1051-1054(1991).

(2) S. Kusumoto, K. Matsumoto, T. Kikuno and K. Torii: "Experimental evaluation of the cost effectiveness of software reviews," Proceedings of *15th Computer Software & Applications Conference,* pp.424-429(1991).

(3) S. Kusumoto, K. Matsumoto, T. Kikuno and K. Torii: "Approaches to improving effectiveness of review activities in technical review process," Proceedings of *International Software Quality Exchange,* pp. 7B1-7B16(1992).

(4) S. Kusumoto, K. Matsumoto, T. Kikuno and K. Torii : "A new metric for cost effectiveness of software reviews," *IEICE Transactions on Information and Systems*, Vol. E75-D, No. 5, pp.674-680(1992).

(5) K. Matsumoto, S. Kusumoto, T. Kikuno and K. Torii: "A new framework of measuring software development processes," Proceedings of *IEEE-CS International Software Metrics Symposium*, pp.108-118(May 1993).

(6) S. Kusumoto, K. Matsumoto, T. Kikuno and K. Torii: "Using a Petri-net model for quantitative analysis of debugging processes in academic environment," *IEICE Transactions on Information and Systems,*(1993)(to appear, in Japanese).

(7) S. Kusumoto, K. Matsumoto, T. Kikuno and K. Tanaka: "Application of fault tolerant techniques to software development process," *Pacific RIM International Symposium on Fault Tolerant Systems,*(1993)(Submitted).

Acknowledgments

During the course of this work, I have been fortunate to have received assistance from many individuals. I would especially like to thank my supervisor Professor Tohru Kikuno for his continuous support, encouragement and guidance for this work.

I am also very grateful to the members of my thesis review committee: Professor Koji Torii and Professor Nobuki Tokura for their invaluable comments and helpful criticisms of this thesis.

I also wish to thank Associate Professor Ken-ichi Matsumoto and Associate Professor Yoshiaki Kakuda for their valuable suggestions and stimulating discussions.

I also wish to thank Associate Professor Katsuro Inoue and Assistant Professor Takeshi Ogihara for their valuable advices.

Many of courses that I have taken during my graduate career have been helpful in preparing this thesis. I would especially like to acknowledge the guidance of Professors Mamoru Fujii, Akihiro Hashimoto, Tadao Kasami, Toshinobu Kashiwabara, Hideo Miyahara, Hirokazu Nishitani, Masaru Sudo and Ken-ichi Taniguti.

I would like to express my thanks to Professor David Notkin of the University of Washington and Dr. Margaret C. Thompson for their insightful comments and valuable discussions on the paper which formed the basis for Chapters 3, 4 and 5 of this thesis.

I wish to express my gratitude to Dr. Kumiyo Nakakoji and Miss Tamara Sumner of the University of Colorado at Boulder for their careful reading of a draft of this thesis. Their suggestions were very helpful.

I would also like to acknowledge the support of Mr. Yukio Mohri of Nihon

Unisys, Ltd. for the industrial experimental projects described Chapters 4 and 5 of this thesis.

Thanks are also due to many friends in the Department of Information and Computer Sciences at Osaka University who gave me many useful suggestions.

Contents

Chapter 1

Introduction

1.1 Software productivity and quality

The management of a large software development project is a complex and intrinsically difficult task. Since a large software is inherently very complex, its production may involve hundreds of man-years of skilled efforts with correspondingly large budgets [Rook 1986]. The discipline that deals with such problems is called *software engineering.* In the late 1960's, the term software engineering was coined as the technical and managerial discipline concerned with the systematic invention, production, and maintenance of high-quality software systems, delivered on time, at minimum cost [Frakes et al. 1991].

Basili and Musa illustrated how software engineering has shifted its focus since 1960's [Basili & Musa 1991]. In the 1970's, the need to develop software in a timely, planned, and controlled fashion became apparent. Several software life-cycle models and schedule tracking methods were introduced as a result of software engineering research. In the 1980's, software engineering focused on lowering development costs and attaining high productivity, since hardware costs continued to decrease, and the personal computer created a mass market that drove software prices down. Various cost models came into use and resource tracking became commonplace, but these methods do not go far enough towards meeting the goals of software engineering. The 1990's will be the quality era, in which software quality is quantified and brought to the center of the development process.

Thus, software quality is a major problem facing software engineering. It is very important to establish effective procedures, methods, notations, tools, and practices for promoting software quality. Software quality has been defined in several ways including : the absence of errors, conformance to requirements, fitness for use, and customer satisfaction[Frakes et al. 1991]. Moreover, quality affects productivity, since products with higher quality require less rework and maintenance.

In order to improve quality, the software development process has to be improved [Frakes et al. 1991] [Humphrey 1988]. That is, essential improvements in the software product can be attained by improving the software development process. For the software development process to be improved, it is necessary to grasp the current state of the process[Basili & Rombach 1988].

There are numerous research studies aimed at improving software development process. Basili and Rombach have proposed the *Goal/ Question/ Metric paradigm* (simply called *G/Q/M paradigm*) in the *TAME* project [Basili & Rombach 1988]. The *G/Q/M paradigm* explicitly describes the relation between the goal to be achieved in the process and the metric to be applied to the process. Next, Humphrey has proposed SEI Software Process Maturity Self-Assessment [Humphrey 1988]. Based on a *process maturity model*, SEI Self-Assessment builds a consensus view of an organization's maturity and the key issues facing it. Ultimately, it presents an improvement plan for software development processes that have been endorsed by general management.

The following key attributes (a)-(c) are stressed in these proposals: (a) the software development process is strictly defined (that is, the software development process is defined as a sequence of fundamental activities that are precisely specified), (b) for each activity, quality data are collected and analyzed statistically, and (c) based on the empirical analysis, each activity is improved or streamlined.

One of the studies based on the approach described above is presented in [Mohri & Kikuno 1991]. They defined the JSP(*Jackson Structured Programming*) [Cameron 1989] software development process that consists of thirteen activities. They collected and analyzed fault data (a type of quality data) from an actual software development process. They found that most of the faults found during testing could have been detected by design and code reviews. Their study concluded by noting the importance of correctly executing such reviews.

It is widely recognized that errors have a large impact on software productivity and quality. In attempting to reduce the number of delivered faults, it is estimated that most companies spend between 50-80% of their development effort on testing [Collofello & Woodfield 1989]. Therefore, reducing the effort or cost of testing is an important step towards increasing productivity and quality in software development.

Two types of activities to detect and remove faults are *review* and *testing*. *Review* is manually performed by a team of developers. *Testing* is executed on a terminal using various convenient tools. Although software reviews and testing are generally believed to be valid quality assurance techniques in that they help detect errors, there is very little evidence illustrating the effectiveness of these techniques. It is very important to quantitatively show the usefulness of these quality assurance techniques. In this thesis, we address the fault detection and removal process and try to quantitatively evaluate the effectiveness of the process. In order to clarify discussions, we discuss reviews and testing activities independently.

1.2 Cost-effectiveness of software reviews

Software reviews are said to be one of the most effective techniques for reducing the testing effort. Technical reviews are frequently executed to detect and remove faults in software throughout the software development life cycle. Faults detected by such reviews are removed at a lower cost compared to those removed by testing. The technical reviews are mainly implemented in the form of "design reviews" and "code reviews".

In order to evaluate the effectiveness of design reviews and code reviews, several metrics have been proposed. For example, Myers has proposed a metric based on the number of faults detected by reviews[Myers 1978]. Fagan has also proposed a metric called *Error Detection Efficiency*, which is calculated based on the number of faults detected by reviews and the total number of faults in the product before reviews[Fagan 1976].

However, there are few metrics that evaluate and prove the effectiveness of technical reviews with respect to software development cost. Only Collofello et al. has taken into account the costs consumed and saved by reviews, and proposed a metric called *Cost Effectiveness* [Collofello & Woodfield 1989]. However, as it does not take into account the total cost to detect and remove all faults from the software by reviews and testing, Collofello's metric is not sufficient for some software development processes.

In this thesis, we provide an example that proves the insufficiency of Collofello's metric (See Chapter 3). We propose a new metric, M_k, for evaluating the cost effectiveness of software reviews. It is based on the degree to which costs to detect and remove all faults from the software in a project are reduced by technical reviews. The proposed metric can be interpreted as a metric that combines Fagan's metric and Collofello's metric. As the value of the proposed metric is normalized by *virtual testing cost* (which is described in Chapter 3), we can use M_k to compare the results of review evaluation across many different kinds of projects.

1.3 Measuring software testing process

Software testing usually consists of unit testing, integration testing, system testing and acceptance testing. Software testing is the last chance to reduce the delivered errors. Such testing is believed to be one of the most effective approaches for improving software quality. In order to improve the efficiency of testing processes, numerous testing techniques and test case generation methods have been proposed. For example, the white-box testing method is proposed for unit testing and the black-box method for acceptance testing [Myers 1979]. In addition to the testing techniques, it is necessary to grasp and analyze quantitatively the current status of the software development process in order to improve the software development process (including the testing process). Process improvements can then be based on the results of this analysis. Measurement provides powerful and effective technologies for such a quantitative and objective approach.

The Software Reliability Growth Model (SRGM) is one of the most well-known models for quantitatively evaluating the software testing process

[Matsumoto et al. 1988]. SRGM expresses the testing process as a relation between the testing time t and the cumulative number of faults removed from the beginning of testing through time t. The testing time t and the cumulative number of faults, which are parameters of the model, are usually computed based on the testing report recorded by the testers. Thus SRGM gives us useful information, e.g., the number of residual faults in the software or MTBF, for deciding the shipping date. The testing time may be interpreted as elapsed time spent on testing, i.e., how long the testers have spent their time testing the software.

But the testing process is very complicated consisting of: planning the general approach, finding resources, and scheduling, determining features to be tested, designing the set of tests, implementing the plan and design, executing the test procedures, checking for termination, and evaluating the test effort and unit[IEEE 1008 1987]. We can easily imagine that it is not appropriate to sum up each of these elapsed time as the testing time t. Based on the definitions of these activities and the definition of SRGM, we should select some of these testing activities and sum up the elapsed time of the selected activities as the testing time t. Unfortunately, most SRGMs do not have enough information to select the activities to be measured [Matsumoto et al. 1988]. This reduces the usefulness of these SRGMs.

One of the best ways to explicitly define the features of the process to be parameterized and the relation between these features and the parameters of the metric, is to express the process to be measured as a mathematical model and define the metric based on this model. There already exist some product metrics which have been defined on models of the software product to be measured. For example, MaCabe proposed a complexity metric, called

Cyclomatic number [MaCabe 1976]. He modeled program code in a directed graph and defined the Cyclomatic number on the graph using graph theory. He also showed a simple rule for translating program code into a directed graph. His rule can be applied to almost all procedure-oriented programming languages. Consequently, it becomes easy to understand the implication of the Cyclomatic number and to apply it to the program code to be measured.

We propose a new framework of measuring software development processes [Matsumoto et al. 1993]. The key idea of the proposed framework is that all activities to be measured can be explained by a mathematical model of the process to be measured. In this framework, a Petri net model is used to model the process, since it is one of the most powerful models for representing concurrent processes such as software development processes [Peterson 1981]. The proposed framework consists of four steps: (1) process modeling, (2) metric definition, (3) process and metric implementation, and (4) process and metric execution. Process modeling transforms the software development process to be measured into a Petri net model which represents essential features of the process. Metric definition clarifies how to evaluate the features of the process in the model. Process and metric implementation means to enact the process based on the model, collect data from the process and compute the metric values based on the model. Process and metric execution means to carry out the process, and to compute data collected from the process.

This thesis presents an application of the proposed framework, in which a coding and debugging process in a students class project was modeled with a Petri net, and four metrics for evaluating the debugging activities were defined. Data were collected from two successive experiments based on the model and used to evaluate the effects of the design method improving debugging activities.

1.4 Outline of the thesis

First, this thesis proposes a new metric, M_k, for evaluating cost effectiveness of software reviews. It is based on the degree to which costs to detect and remove all faults from the software in a project are reduced by technical reviews. Then we show the validity and usefulness of the proposed metric M_k by presenting an experimental evaluation performed in an industrial environment which compared M_k and conventional metrics.

This thesis proposes a new framework for measuring software development processes. The key idea of the proposed framework is that all activities to be measured can be defined based on a mathematical description of the process to be measured.

In Chapter 2, we define software development processes, software reviews, and software testing to be discussed in this thesis. Chapter 3 presents three conventional metrics: M_m by Myers, M_f by Fagan, and M_c by Collofello, for evaluating the effectiveness of technical reviews. Then, a new metric, M_k, is introduced for evaluating the cost effectiveness of technical reviews. Chapter 4 shows an experimental evaluation of the metrics M_m, M_f, M_c and M_k using the data collected in industrial project. Experimental results are summarized to show the superiority of the M_k metric. Chapter 5 describes an application of the M_k metric, comparing three methods for allocating review effort.

Chapter 6 shows four major activities and the potential capability of the framework for measuring the software development process and describing testing activities in an academic environment. Chapter 7 presents an application of the framework, in which we tried to evaluate the effectiveness of the design method in an academic environment in terms of improvements in the testing activities.

Finally, Chapter 8 summarizes the main results of this thesis and presents some areas for future research.

Chapter 2

Preliminary Definitions

2.1 Error and fault

It is widely recognized that errors have a large impact on software productivity and quality. Numerous studies have been conducted in the field of "error analysis" in order to understand the effect of each error on software productivity and quality. For example, Mohri and Kikuno [Mohri & Kikuno 1991] proposed a fault analysis procedure for software development using JSP. The procedure determines the steps in the JSP process when fault introduction might occur, and when fault detection (and correction) should be executed.

Sometimes the term "error" and "fault" are used with the same meaning. In this thesis, we follow the IEEE standard with respect to the definition of error and fault. In the IEEE standard, an error is defined as a human action that results in software which contains a fault. Examples include omission or misinterpretation of user requirements in software specification and incorrect translation or omission of a requirement in the design specification. A fault is defined as a manifestation of an error in software. A fault, if encountered, may cause a failure (synonymous with bug).

2.2 Software development process

The software development process we use in this thesis is the standard waterfall model [Royce 1970] consisting of the following phases:

(1)Concept exploration and feasibility analysis phase: Identify a need to automate a process and analyze project feasibility.

(2)Requirement specification phase: Analyze and document system requirements. The requirements document must clearly state what the projected system will do, what elements the software product will have, and what characteristics the product elements must have.

(3)Design phase: Design the system and document the design. The design document specifies how to build a software system to satisfy the requirements.

(4)Implementation phase: Write the software.

(5)Testing phase: Exercise the software to verify that it satisfies its requirements.

(6)Maintenance phase: Following deployment of the software product, faults are corrected and the system is changed or enhanced.

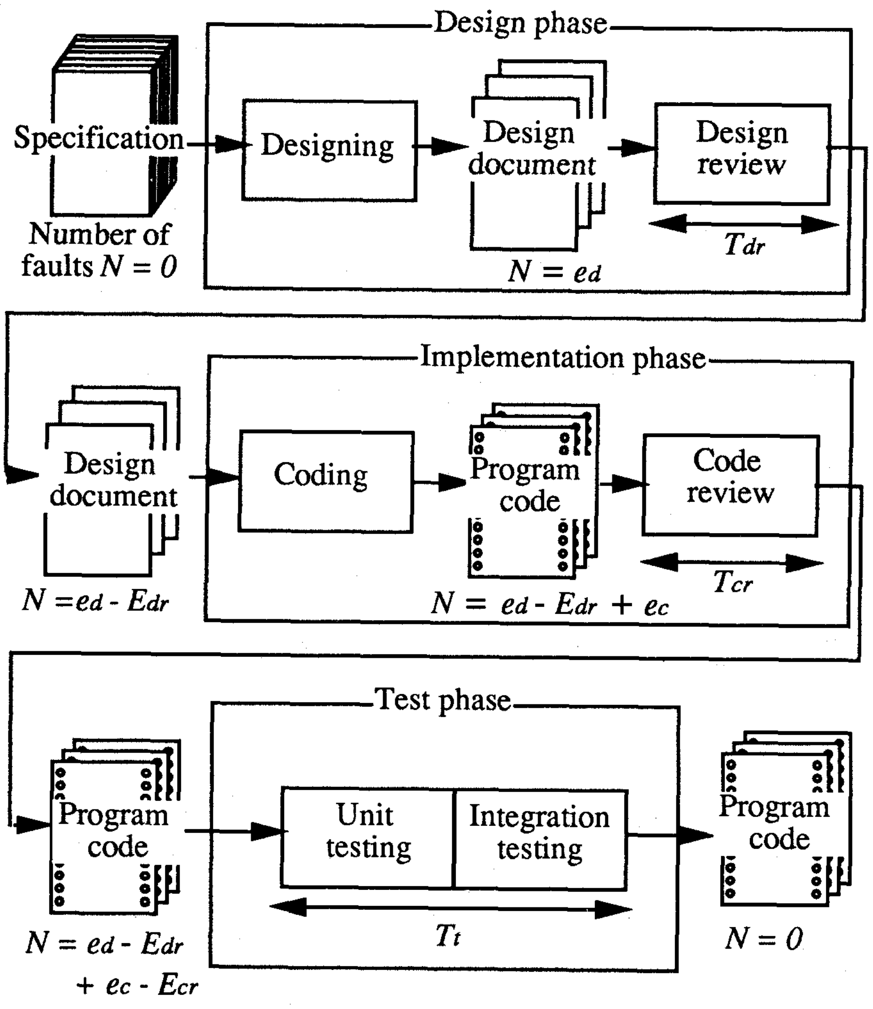
Though the waterfall model is often criticized as having little to do with project realities [Frakes et al. 1991], we adopt it in this thesis because it is still the model most often used on large software development projects. Phases (1) to (4) are development activities where faults are introduced [Coward 1982]. Technical reviews are frequently utilized in each phase, such as specification reviews, design reviews and code reviews. Generally, technical reviews and testing are the primary means for detecting faults in the product. Thus, in this thesis, we consider technical reviews and testing to consist of fault detection, fault localization and fault fixing.

In order to clarify discussions, we simplify the waterfall model as shown in Figure 2.1. We assume that the software development process consists of three successive phases: a design phase, an implementation phase and a test phase (See Figure 2.1). We also assume that software development is executed by a team. Additionally, we assume that a given specification has no faults and is not changed throughout the development process.

As shown in Figure 2.1, we introduce several parameters. In the design phase, each member of the team designs some parts of system to be developed and writes design documents. Let e_d be the total number of faults introduced into the design documents. We assume that during a design review, E_{dr} faults are detected in the design documents. Let T_{dr} be the total time expended for the design review.

Next, in the implementation phase, each member of the team creates program code based on the design documents. We assume that $(e_d - E_{dr})$ faults remaining in the design documents manifest themselves as faults in the program code. In addition, e_c faults are newly introduced into the program code by coding. Therefore, when coding is complete, the program code now contains $(e_d - E_{dr} + e_c)$ faults.

At the end of the implementation phase, the code is reviewed. In the code review, all members of the team examine the program and verify that it compiles with both the design documents and the specification. We assume that during the code review, E_{cr} faults are detected. Thus at the end of the



e_d : Number of faults introduced by designing
e_c : Number of faults introduced by coding
E_{dr} : Number of faults detected by design review
E_{cr} : Number of faults detected by code review
T_{dr} : Total time of design review
T_{cr} : Total time of code review
T_t: Total time of unit and integration testing

Figure 2.1 Software development process

implementation phase, $(e_d - E_{dr} + e_c - E_{cr})$ faults still remained in the program code. Let T_{cr} be the total time expended for the code review.

Finally, the test phase is the period of time during which components of a software product are evaluated and integrated and the final software product is evaluated to determine whether or not requirements have been satisfied [IEEE 610 1990]. The test phase consists of two subphases: unit testing and integration testing. During unit testing, each member of the team evaluates his or her own program code. During integration testing, all program units are integrated and evaluated. Supposedly all the faults from design and implementation are detected and removed by the unit and integration testing.

2.3 Software reviews

According to IEEE Standards, a software review is defined to be a formal evaluation of software elements or project status to ascertain discrepancies with planned results and to recommend improvements [IEEE 1028 1988]. Software elements include project planning documents, software requirements and design specifications, test effort documentation, program source code, representation of software solutions implemented in firmware, and reports and data.

Generally speaking, software reviews can be classified into two types: management reviews and technical reviews [IEEE 1028 1988]. A management review is an evaluation of a project level plan or project status relative to that plan by a designated review team. A technical review is a team evaluation of a software element. This thesis focuses on technical reviews, especially design reviews and code reviews.

Several practical review methods [Weinberg & Freedman 1984] have been proposed for the technical review. Among them, inspections and walkthroughs are well known and applied by many computer companies.

Walkthroughs are a method in which a designer or programmer leads members of the development team and other interested parties through a segment of documentation or code. The participants ask questions and make comments about possible errors, violations of development standards and other problems [IEEE 610 1990]. All review members except for the designer or programmer do not have to understand the details(structure, algorithm, data structure and so on) of all products in advance. Thus it is common that differences exists in the level of understanding of the products. As a result, applying walkthroughs to large-scale products may be difficult to complete in a relatively short period of time.

Inspections are another review method that relies on visual examination of development products to detect errors, violations of development standards and other problems [IEEE 610 1990]. The inspection method was described formally and rigorously by Fagan [Fagan 1976]. In contrast to the walkthrough method, the participants must understand the details of all products before the inspection is executed. Moreover, participants need to be trained in the inspection methodology. This is a fundamental difference between inspection and walkthrough review methods. Given these differences, we think that inspections are best carried out by skilled developers (reviewers) and walkthroughs are more appropriate for novices.

Fagan introduced a metric measuring review efficiency, called *error detection efficiency*, which is defined as the number of faults found by reviews over the total number of faults in the product before its review. Using this metric, Fagan has evaluated the effectiveness of detailed design reviews, code reviews, and unit test reviews[Fagan 1976].

2.4 Software testing

In the IEEE standard [IEEE 610 1990], testing is defined as the process of exercising or evaluating a system or system component by manual or automated means to verify that it satisfies specified requirements or to identify differences between expected and actual results.

Software systems are tested at several different levels. Generally, software testing consists of following four types [Frakes et al. 1991]:

Unit testing : A unit is a piece of software implemented by a single programmer. Units are typically single functions or small groups of functions that work together to accomplish some simple task. Programmers are usually responsible for testing units alone during their implementation before they are integrated with other parts of the system.

Integration testing : When several units are brought together to form a module, subsystem, or system, they are tested as a group.

System testing : Once a system has been completely integrated, it must be tested as a whole. System testing exercises a program with input generated from system requirements that may not reflect the use of the system by its intended users.

Acceptance testing : The whole system is exercised with data reflecting use of the system by its intended users. Often small groups of users participate in acceptance testing in an effort to provide a more realistic trial of the software.

In Chapters 3 and 4, we propose a new metric, M_k, for evaluating the cost effectiveness of software reviews. The metric is defined in terms of the costs of unit and integration test reduced by developing technical reviews. In Chapters 6 and 7, we evaluated the effectiveness of the design method in an academic environment improving the unit testing activities.

Chapter 3

M_k: A New Metric for Software Reviews

3.1 Conventional review metrics

In this Section, we present a brief overview of some metrics previously proposed for assessing the effectiveness of software reviews.

3.1.1 Myers's metric M_m

Myers has tried to evaluate the effectiveness of black-box testing, white-box testing, and code reviews individually, as well as in combination with one another[Myers 1978]. In his evaluations, a metric based on the number of faults detected by each technique is extensively utilized.

In order to evaluate the effectiveness of both design review and code review, applying Myers's metric M_m to the software development process shown in Figure 2.1, yields the following equation:

$$M_m = E_{dr} + E_{cr}. \tag{3.1}$$

As such, M_m depends on an assumption that (1) the same number of faults exist in each product before their reviews, and that (2) we do not take into account the cost expended by reviews.

3.1.2 Fagan's metric M_f

Fagan has evaluated the effectiveness of detailed design reviews, code reviews, and unit test reviews[Fagan 1976]. He introduced the *Error Detection Efficiency* metric M_f for measuring review efficiency. M_f is defined as the number of faults found by reviews over the total number of faults in the product existed before the reviews. ("Error" in [Fagan 1976] is almost the same as "fault".)

Applying Fagan's metric M_f to the software development process shown in Figure 2.1, yields the following equation:

$$M_f = \frac{E_{dr} + E_{cr}}{e_d + e_c}. \tag{3.2}$$

M_f is meaningful even if the same number of faults are not in each product before the reviews. But M_f also does not account for the cost expended by reviews.

3.1.3 Collofello's metric M_c

There may exist a reliability assurance technique (e.g. review or testing) that has a low *Error Detection Efficiency* but is still considered worthwhile. Collofello and Woodfield have proposed *Cost Effectiveness*, M_c, as a measure of worth[Collofello & Woodfield 1989]. The M_c of an error-detection process is defined to be the ratio of the "costs saved by the process" to the "cost consumed by the process."

Aapplying Collofello's metric M_c to the software development process shown in Figure 2.1, yields the following equation:

$$M_c = \frac{\Delta C_t}{C_r}, \tag{3.3}$$

where ΔC_t is the testing cost saved by design and code reviews, and C_r is the total cost consumed by those reviews. Then, C_r and ΔC_t can be expressed by the following equations [Collofello & Woodfield 1989], where c_r is the average cost to detect and remove a fault in reviews, and c_t is the average cost to detect and remove a fault in testing provided that $(E_{dr} + E_{cr})$ faults are detected by reviews:

$$C_r = (E_{dr} + E_{cr}) \cdot c_r. \tag{3.4}$$

$$\Delta C_t = (E_{dr} + E_{cr}) \cdot c_t. \tag{3.5}$$

From equations (3.3),(3.4) and (3.5), M_c is rewritten as follows:

$$\begin{aligned} M_c &= \frac{(E_{dr} + E_{cr}) \cdot c_t}{(E_{dr} + E_{cr}) \cdot c_r} \\ &= \frac{c_t}{c_r}. \end{aligned} \tag{3.6}$$

Equation (3.6) shows that M_c is the ratio of the average cost to detect and remove a fault in testing to that to detect and remove a fault in reviews.

Next, we assume that the total cost to detect and remove faults in reviews is the total time expended for both design and code reviews, and that the total cost to detect and remove faults in testing is the total time expended for unit testing and integration testing. Then c_r and c_t can be estimated by the following equations, respectively:

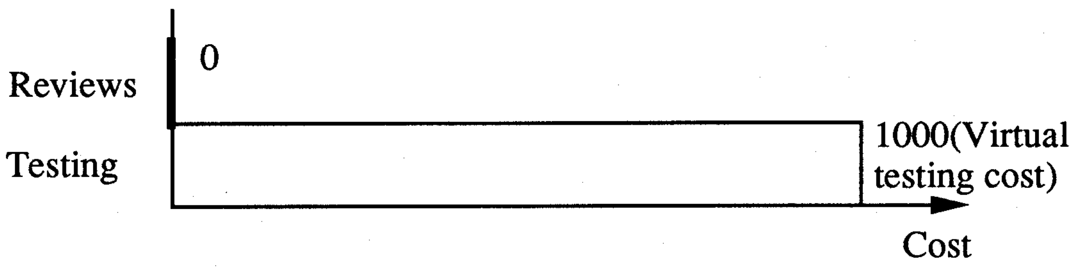


Figure 3.1 Virtual testing cost

$$c_r = \frac{T_{dr} + T_{cr}}{E_{dr} + E_{cr}} \tag{3.7}$$

and

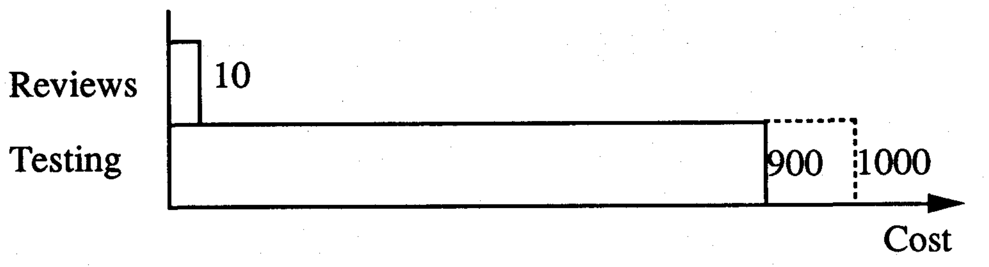
$$c_t = \frac{T_t}{(e_d + e_c) - (E_{dr} + E_{cr})}. \tag{3.8}$$

3.2 New metric M_k

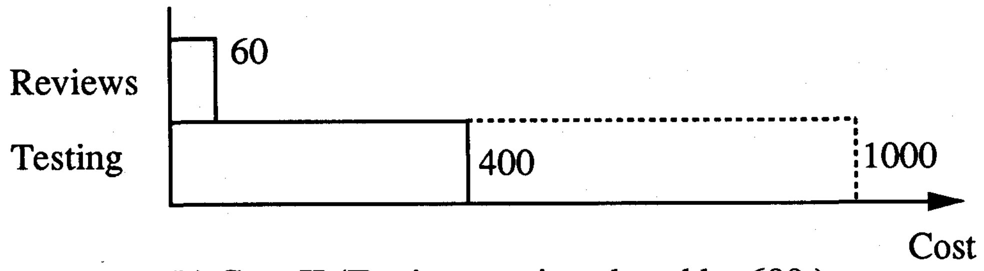
3.2.1 Why a new metric is needed

Among the three kinds of metrics discussed (M_m, M_f and M_c), Collofello's metric M_c is the most practical, which takes into account the costs consumed and saved by the reviews, because cost is one of the most important factors for software development project management.

But we should point out that M_c is still insufficient for evaluating the cost effectiveness of software reviews in that M_c does not take into account the total cost to detect and remove all faults from the software by reviews and testing. For example, let us assume that testing cost for program code is



(a) Case I (Testing cost is reduced by 100.)



(b) Case II (Testing cost is reduced by 600.)

Figure 3.2 Costs for reviews and testing in two cases

1,000 (person-months, thousands of dollars, etc.) if no reviews are executed (See Figure 3.1). We call this the *virtual testing cost*. Consider two cases of software development: (Case I) the testing cost saved by reviews is 100 and the total costs consumed by reviews is 10, and (Case II) the testing cost saved by reviews is 600 and the total costs consumed by reviews is 60. Both Case I and II are shown in Figure 3.2.

If we apply Collofello's metric M_c to Case I and II, then the value of M_c is 10 in each case, thereby, the cost effectiveness of these particular reviews cannot be distinguished. But, the total costs to detect and remove all faults from the design documents and program code by reviews and testing in Case

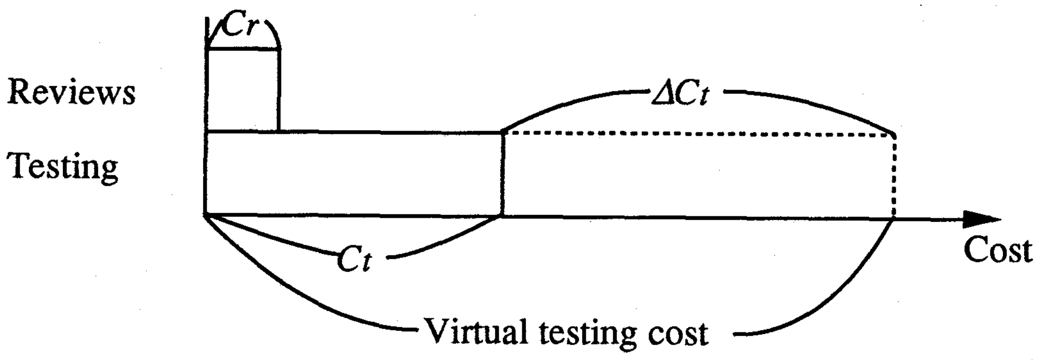


Figure 3.3 Parameters for new metric *Mk*

I and II are 910 and 460, respectively. It is obvious that the reviews in Case II are more effective than the ones in Case I in terms of reducing the total cost. The cost reduction of detecting and removing faults in the project is one of the most important objectives of software reviews.

3.2.2 Definition of M_k

Based on the discussion in subsection 3.2.1, we propose a new metric M_k for evaluating the cost effectiveness of software reviews in terms of reduction of cost to detect and remove all faults from the software.

Assume that by spending the cost C_r for reviews, the cost C_t is needed for testing. Additionally, in this case, the testing cost is reduced by ΔC_t compared to the virtual testing cost provided no review is executed (See Figure 3.3). Then, a new metric M_k is defined by the following equation:

$$M_k = \frac{\Delta C_t - C_r}{C_t + \Delta C_t} \tag{3.9}$$

Intuitively, M_k is a ratio of the reduction of the total costs to detect and remove all faults from the design documents and program code using design and code reviews in a project to the virtual testing cost (of the program code). Since the value of the proposed metric is normalized by the virtual testing cost, we can use M_k to compare the results of reviews across many different kinds of projects.

Applying a new metric M_k to the software development process shown in Figure 2.1 by using the same notation as in equation (3.3), yields the following equation:

$$\begin{aligned} M_k &= \frac{\Delta C_t - C_r}{C_t + \Delta C_t} \\ &= \frac{(E_{dr} + E_{cr}) \cdot c_t - (E_{dr} + E_{cr}) \cdot c_r}{(e_d + e_c) \cdot c_t} \\ &= \frac{E_{dr} + E_{cr}}{e_d + e_c} \cdot \frac{c_t - c_r}{c_t}. \end{aligned} \tag{3.10}$$

Now, if we apply M_k to Case I and II (See Figure 3.2), the values of M_k for Case I and Case II are 0.09 and 0.54, respectively. Thus, M_k identifies that the review in Case II is more effective than the review in Case I.

3.2.3 The relationship of M_k to conventional metrics

The proposed metric M_k is similar to Collofello's metric M_c with respect to including the testing cost saved by the reviews in the definition.

In this subsection, we discuss the relationship between the proposed metric

M_k and the conventional metrics M_m, M_f and M_c. These relationships will be discussed again in Chapter 4 with respect to the experimental data.

From the equations (3.2),(3.6) and (3.10), M_k can be rewritten as:

$$\begin{aligned} M_k &= \frac{E_{dr} + E_{cr}}{e_d + e_c} \cdot \frac{c_t - c_r}{c_t} \\ &= M_f \left(1 - \frac{1}{M_c}\right). \end{aligned} \tag{3.11}$$

Hence, M_k can be interpreted as being a metric that combines M_f with M_c.

Assume that the value of M_f is constant (*Assumption 1*), then from equation (3.11), it is clear that M_k is inversely proportional to M_c:

$$M_k = a_1 \left(1 - \frac{1}{M_c}\right), \tag{3.12}$$

where a_1 is a proportional factor or coefficient.

Next, similarly we assume that M_c, that is a ratio of the average cost c_t to the average cost c_r, is constant (*Assumption 2*), then it is clear that M_k is proportional to M_f. As an extreme, we assume that $c_t \gg c_r$ (*Assumption 3*), then, M_k is approximately equal to M_f.

Finally, we assume that either *Assumption 2* or *Assumption 3* holds, and the total number of faults $(e_d + e_c)$ is constant (*Assumption 4*). Then, from equations (3.1), (3.2) and (3.10), M_k is proportional to M_m:

$$M_k = a_2 M_m, \tag{3.13}$$

where a_2 is some coefficient.

Chapter 4

Experimental Evaluation of Metric M_k

To demonstrate the validity of metric M_k, we apply four metrics M_m, M_f, M_c and M_k to the data collected from an experimental software development project, and compare the resultant values obtained by these metrics.

4.1 Outline of the project

The experimental project was performed in a certain computer company [Kusumoto et al. 1990]. The main characteristics of the project were:

(1) Seven teams developed the same batch processing system. The system is a file processing program in a business application. Each team designed the system using the JSP(Jackson Structured Programming) method and implemented it in COBOL.

(2) Each team consisted of four or five novice developers. Teams were organized by an instructor in an attempt to minimize individual differences in performance. Two terminals were assigned to each team.

(3) The system consists of eighteen program modules. The same partition of program modules was given to each team. However, the assignment of program modules to team members was arbitrarily determined by each team's leader. The final programs consisted of approximately 2,000 lines of code.

(4) The instructor tested the programs developed by each team to ensure that no faults remained after the team's testing phase.

4.2 Experimental data

The experimental data are summarized in Table 4.1 which shows eight data points for each team. Among them, the total number of faults ($e_d + e_c$), the total number of faults detected in design review and code review ($E_{dr} + E_{cr}$), and the total cost of these reviews (C_r) are compiled based on the data reported manually by each programmer.

The testing cost (C_t) is computed based on automatically collected data concerning terminal access time [Kusumoto et al. 1991]. Because testing in this project is almost performed on the terminal. The rest of the data, c_r, c_t, ΔC_t, and ($C_t + \Delta C_t$), are computed from the data mentioned above. In particular, the value of ΔC_t is computed from equation (3.5) [Collofello & Woodfield 1989].

Table 4.1 Experimental data

Team	Total number of faults e_d+e_c	Number of faults detected by reviews $E_{dr}+E_{cr}(=M_m)$	Review cost $C_r\ (= T_{dr}+T_{cr})$ (min.)	Average cost to detect a fault in reviews $c_r(=C_r/M_m)$ (min.)	Testing cost $C_t\ (=T_t)$ (min.)	Average cost to detect a fault in testing c_t (min.)	Testing cost saved by reviews ΔC_t (min.)	Virtual testing cost $C_t+\Delta C_t$ (min.)
T1	21	8	500	62.5	5947	457.5	3660	9607
T2	30	15	620	41.3	5032	335.5	5033	10065
T3	22	7	510	72.9	7186	479.1	3354	10540
T4	18	6	570	95.0	4367	363.9	2183	6550
T5	28	13	570	43.8	5274	351.6	4571	9845
T6	38	18	560	31.1	4866	243.3	4379	9245
T7	47	15	420	28.0	7521	235.0	3525	11046

Table 4.2 Results of applying metrics

Team	*Mc* (rank)	*Mk* (rank)
T1	7.32 (5)	0.329 (4)
T2	8.12 (2)	0.438 (1)
T3	6.58 (6)	0.270 (6)
T4	3.83 (7)	0.246 (7)
T5	8.02 (3)	0.406 (3)
T6	7.82 (4)	0.413 (2)
T7	8.39 (1)	0.281 (5)

4.3 Analysis of M_k's usefulness

4.3.1 Comparison to M_c

In this subsection, we show the advantages of the proposed metric M_k over M_c. Among the three conventional metrics M_m, M_f and M_c, Collofello's metric M_c is the most practical one since M_c takes into account the costs consumed and saved by the reviews. The proposed metric M_k is similar to M_c in the sense that M_k includes the testing cost saved by the reviews.

Table 4.2 shows the values and ranks of two metrics M_c and M_k for seven teams. Spearman's rank correlation test indicates that there is not strong correlation between M_c and M_k (correlation coefficient is 0.61, level of significance is 0.05). But, we should note that the ranks of team $T7$ are quite different between M_c and M_k: the cost effectiveness of reviews by team $T7$ is best among these seven teams in terms of M_c, but it is not so good in terms of M_k. By examining the details of data collected on team $T7$, we have found that team $T7$ exhibited interesting behavior which will be discussed in the next subsection. Therefore, we eliminated team $T7$ from the following analysis. After removing team $T7$, the value of rank correlation between M_c and M_k increased to 0.94 with the same level of significance(0.05). This implies that M_c and M_k produce nearly identical measurements for teams $T1$ through $T6$.

4.3.2 Comparison with respect to team $T7$

Next, we compare the metrics M_k and M_c computed for team $T7$. The following characteristics of team $T7$ were observed from the collected data in Table 4.1:

(1) Average cost to detect and remove a fault in reviews (c_r) is the lowest among all seven teams (28.0 min.).

(2) Virtual testing cost ($C_t + \Delta C_t$) is the highest among seven teams (11,046 min.).

(3) Review cost (C_r) is the lowest among seven teams (420 min.).

Team $T7$ has done very efficient reviews with respect to c_r. But, their virtual testing cost is very high and their review cost is low. Thus, the reviews of team $T7$ were efficient even though $T7$ did not use the resources effectively. In this case, M_k is a more appropriate review metric for team $T7$, because it reveals that the cost effectiveness of reviews was not so good among these seven teams.

4.3.3 Stability of metric M_k

Stability is one of the most important aspects of a metric. From Table 4.2, we find that team $T3$ exhibits characteristics similar to team $T7$:

(1) The number of faults detected by reviews over the total number of faults in the product before its review is low(0.32),

(2) C_r is relatively low(510 min.),

(3) C_t is relatively high(7,186 min.).

The rank of M_k for these two teams are almost the same (sixth for $T3$ and fifth for $T7$). On the other hand, the ranks of M_c are quite different (first for $T7$ and sixth for $T3$).

Thus, the values of M_k are stable for teams that exhibit similar characteristics for reviews and testing, but the values of M_c are not stable.

From these observations, we conclude that M_k provides a more practical evaluation than M_c.

4.3.4 Estimation of M_k

To evaluate metric M_k, we need the values of the total number of faults (e_d+e_c) and the average cost c_t to detect and remove a fault in testing. Unfortunately, these values are obtained at the end of testing. Thus we can not evaluate review activity using M_k until the end of testing, which largely reduces the usefulness of the proposed metric.

In this subsection, we present a method for estimating M_k at the end of reviews. The correlations among the data collected from reviews ($(E_{dr} + E_{cr})$, $(T_{dr} + T_{cr})$, c_r and the reciprocal of c_r) and the data collected from testing ($(e_d + e_c)$ and the reciprocal of c_t) are summarized in Table 4.3. There are two high coefficients of correlations: 0.98 between $(e_d + e_c)$ and the reciprocal of

Table 4.3 Coefficient of correlation among experimental data

	$E_{dr} + E_{cr}$	$T_{dr} + T_{cr}$	c_r	$1/c_r$
e_d+e_c	0.84	-0.43	0.88	0.98
$1/c_t$	0.82	-0.26	0.73	0.92

Table 4.4 Estimated value and error of estimation of *Mk*

Team	$\widehat{M_k}$	$\left\lvert \frac{\widehat{M_k} - M_k}{M_k} \right\rvert \times 100$ (%)
T1	0.303	7
T2	0.420	4
T3	0.289	8
T4	0.286	14
T5	0.376	9
T6	0.400	4
T7	0.300	8
average		7

c_r, and 0.92 between the reciprocal of c_t and the reciprocal of c_r. Regression analysis indicates that $(e_d + e_c)$ and the reciprocal of c_t can be estimated by the following equations:

$$(e_d + e_c) = 4.88 + 1090 \left(\frac{1}{c_r}\right) \tag{4.1}$$

and

$$\frac{1}{c_t} = 0.00114 + 0.0848 \left(\frac{1}{c_r}\right). \tag{4.2}$$

The estimated values of M_k using equations (4.1) and (4.2) for the seven teams are summarized in Table 4.4, together with the estimation error of M_k. From Table 4.4, the average error estimation is about 7%. Thus we conclude that it is possible to estimate M_k, within $\pm 10\%$ at the end of reviews before the completion of testing.

Chapter 5

Application of M_k

5.1 Overview

We have already described the effectiveness of technical reviews towards reducing the testing effort. Several practical review methods have been proposed for the technical review [Weinberg & Freedman 1984]. Among them, inspections and walkthroughs are well known and applied by many computer companies. Several experiments have studied the effectiveness of these review methods [Fagan 1976] [Myers 1978]. However, previous studies do not clearly describe the effort required for each element in a review.

Most review methods are based on the principle that the same amount of effort is needed to review each product. In other words, the effort required to review a product is independent of its quality. Since products having low quality (e.g. containing many faults) are intermixed with products having high quality (e.g. containing a few faults), these review methods are not very effective. If we could estimate which products have low quality prior to the review process, then we could focus our review effort on these products. Such focusing would improve the effectiveness of technical reviews.

In this Chapter, we advocate a new principle stating that much effort should be spent reviewing products with lower quality. When applying this new principle, we estimate the quality of a product by considering the capabilities of the product's developer. That is, we assume that less capable developers create products of lower quality. Additionally, we propose two new methods for determining how to distribute review effort: a proportional distribution of effort and a concentrated distribution of effort. We also discuss an ordinal or uniform distribution of effort.

We applied these three methods to software development in an industrial environment where new employees developed a business application program. We compared the three methods using our new metric M_k. The analysis of experimental data showed the superiority of the proposed new principle over the conventional principle of uniform effort distribution.

5.2 Amount of effort in technical reviews

Most proposed review methods are based on the conventional principle that each product requires the same amount of effort to review. In other words, we usually spend the same amount of effort on each product, independent of the quality of the product. Since the quality of two products may differ to a large extent, this approach is generally not effective. If products having low quality could be distinguished before the review phase, then we could focus our review effort on them. This would improve the effectiveness of the review process.

It is, however, very difficult to estimate the quality of the product beforehand. In the next section, we describe a way to estimate product quality and how to use this information to enhance the review process.

5.3 Capability of developer

It is widely recognized that the individual capabilities of a developer is strongly related to the productivity and quality of software [Matsumoto et al. 1992] [Sackman et al. 1968]. For example, Sackman et al. showed that for most performance variables there are very large individual differences in programming performance[Sackman et al. 1968]. Additionally in the COCOMO(COnstructive COst MOdel) model[Boehm 1981], there are fifteen factors which affect the development cost. Among these factors, the capability of the developer or development team is most important and plays a dominant role. Moreover, our early experiments[Matsumoto et al. 1992] revealed that programmers with high performance constructed programs with fewer faults. Even when faults are introduced into programs, programmers with high performance can remove them faster [Matsumoto et al. 1992].

Based on these facts, we classify the capabilities of each developer into three categories: low, medium or high performance. (Subsection 5.5.1 will explain how to distinguish these three levels of performance.) We assume that developers with low levels of capabilities construct products with low quality. We make analogous assumptions about developers exhibiting medium or high levels of performance.

5.4 Three methods for allocating review effort

We introduce new methods for determining how much review effort to allocate across products. The decision regarding how much effort to expend on each product is based on the quality of that product.

	Products developed by *A*	Products developed by *B*	Products developed by *C*
(a) Method *U*	Review using CL_2 $l_2 = 20$	Review using CL_2 $l_2 = 20$	Review using CL_2 $l_2 = 20$
(b) Method *P*	Review using CL_3 $l_3 = 10$	Review using CL_2 $l_2 = 20$	Review using CL_1 $l_1 = 30$
(c) Method *C*	No review	Review using CL_1 $l_1 = 30$	Review using CL_1 $l_1 = 30$

Figure 5.1 Three effort allocation methods

Assume that each developer constructs the same number of modules of the same size. Let s be the size of a module.

Assume that a review is executed using a check list. We consider three kinds of check lists CL_1, CL_2 and CL_3. Let l_1, l_2 and l_3 ($l_1 > l_2 > l_3$) be the numbers of items in each check list CL_1, CL_2 and CL_3, respectively. Assume that each item requires the same amount of time to execute, and let u denote the time needed for each item.

We define three review methods as follows:

(1) *Uniform distribution*(called *Method U)*: Each product is reviewed using the same check list CL_2, independent of the quality of product.

(2) *Proportional distribution*(called *Method P)*: Products with low quality are reviewed using the largest check list, CL_1. Products with medium quality are reviewed using check list, CL_2. Products with high quality are reviewed using the smallest check list, CL_3.

(3) *Concentrated distribution*(called *Method C)*: Only products with low and medium qualities are reviewed using check list CL_1.

We give an intuitive explanation for these three review methods using an example. Consider a team consisting of three developers A, B and C. Assume that developers A, B and C have high, medium and low capabilities, respectively. Consider check list CL_1, CL_2 and CL_3 with sizes l_1=30, l_2=20 and l_3=10, respectively.

Figure 5.1 shows the distribution of effort for each review method. By Method U in Figure 5.1(a), total review time is $60us$ and review time of each product is $20us$. By Method P in Figure 5.1(b), total review time is $60us$. Review times of products constructed by A, B and C are $10u$, $20u$ and $30u$ for each product, respectively. The distribution of review times is proportional to the capabilities of the developers. Finally, by Method C in Figure 5.1(c), total review time is $60us$. Review time is $30u$ each for the products constructed by B and C. No time is spent for products constructed by A.

5.5 Experiment

To compare the usefulness of the proposed review methods, we applied these three methods(Method C, Method P and Method U) to an experimental software development project. The software development process in this experiment consisted of three successive phases: design, implementation and testing as shown in Figure 2.1.

5.5.1 Outline of the project

The experimental project was performed in a computer company from August 14, 1990 to September 6, 1990. The main characteristics of the project were:

(1) Developers were new employees of the computer company and had just graduated from college in March 1990.

(2) Thirteen teams ($T1$ through $T13$) of developers constructed the same batch processing system. The system is a file processing program in a business application. Each team designed the system using the JSP(Jackson Structured Programming) method and implemented it in COBOL.

(3) The system consists of eighteen program modules. The same partition of program modules was given to each team. However, the assignment of program modules to team members was arbitrarily determined by each team's leader. The final programs were approximately 2,000 lines of code.

(4) Each team consisted of five developers ($m1$ through $m5$). Teams were organized by an instructor who attempted to minimize individual differences in performance.

(5) Each team was assigned two terminals.

In the experiment, we determined the capabilities of each of the five members in a team as follows: Each member of the team selected one product he or she had created to review. All other team members jointly reviewed the product, trying to identify faults. Team members were ranked according to the number of faults found in the product and assigned a level of capability based on that rank. High capability levels were assigned to the two members whose products had the least number of faults, low to the two members whose products had the greatest number, and medium to the member whose product fell in the middle.

5.5.2 Details of review activities

The thirteen teams were divided into three groups: Three teams ($T1$ through $T3$) reviewed products using Method C, four teams ($T4$ through $T7$) reviewed products using Method P, and six teams ($T8$ through $T13$) reviewed products using Method U.

Since the system developed in this experiment is relatively small, and the developers were new employees of the company, we adopted the following review process:

- Step1(Presentation): The developer explains the product he or she has created.
- Step2(Question): Participants ask questions about the product.

- Step3(Check): All members check the product jointly using a specified check-list.
- Step4(Rework): The developer updates the product by correcting all faults pointed out in Step3.
- Step5(Follow): The team leader checks the correctness of the results of Step 4.

These steps are based on the inspection method proposed by[Fagan 1976].

The check lists CL_1, CL_2 and CL_3 were constructed based on the fault information collected from the same project developed the previous year, as follows: (1)sort check items in decreasing order of the number of the occurrences of the corresponding faults; (2)select 40 check items from the top to form CL_1; (3)select 25 check items from the top to form CL_2; and (4)select 10 items from the top to form CL_3.

5.5.3 Experimental data

The experimental data are summarized in Table 5.1 and Table 5.2. Table 5.1 includes the data for each team $Ti(1 \leq i \leq 13)$, and Table 5.2 includes the data for each member $mi(1 \leq i \leq 5)$ of the team.

In Table 5.1, the total number of faults ($e_d + e_c$), the number of faults detected by review ($E_{dr} + E_{cr}$), and the total review cost (C_r) are calculated based on data reported by each programmer and by the team leader. The testing cost (C_t) is calculated based on data concerning terminal access time, which is collected automatically [Kusumoto et al. 1992].

In Table 5.2, the number of faults introduced by each member, the total number of faults, the average number of faults, and the variance of each mem-

Table 5.1 Experimental data - 1

Effort allocation methods	Team	Total number of faults e_d+e_c	Number of faults detected by reviews $E_{dr}+E_{cr}(=M_m)$	Review cost C_r (= $T_{dr}+T_{cr}$) (min.)	Testing cost C_t (=T_t) (min.)
Method *C* (Concentrated)	*T1*	98	76	860	7724
	T2	61	46	600	4574
	T3	46	24	840	5641
Method *P* (Proportional)	*T4*	54	43	610	9680
	T5	36	20	820	4340
	T6	43	30	880	6688
	T7	40	19	890	5283
Method *U* (Uniform)	*T8*	30	22	1050	8682
	T9	23	13	870	6065
	T10	38	26	660	7909
	T11	49	20	430	7407
	T12	64	34	560	10150
	T13	42	18	840	6483

Table 5.2 Experimental data - 2

Effort allocation methods	Team	Number of faults introduced by each member					Total number of faults	Average number of faults	Variance number of faults
		m1	*m2*	*m3*	*m4*	*m5*			
Method *C* (Concentrated)	*T1*	25	24	23	11	15	98	19.6	38.8
	T2	20	16	6	15	4	61	12.2	47.2
	T3	12	5	6	17	6	46	9.2	26.7
Method *P* (Proportional)	*T4*	9	20	9	9	7	54	10.8	27.2
	T5	7	9	3	16	1	36	7.2	34.2
	T6	9	4	6	5	19	43	8.6	37.3
	T7	9	13	4	7	7	40	8.0	11.0
Method *U* (Uniform)	*T8*	6	4	6	11	3	30	6.0	9.5
	T9	3	8	7	4	1	23	4.6	8.3
	T10	6	9	9	10	4	38	7.6	6.3
	T11	7	12	9	5	16	49	9.8	18.7
	T12	8	26	12	11	7	64	12.8	58.7
	T13	12	8	8	7	7	42	8.4	4.3

Table 5.3 Average values for each method

Effort allocation methods	Average review cost (min.)	Average testing cost	Metric M_f
Method C	767	5980	0.71
Method P	800	6498	0.65
Method U	735	7783	0.54

ber's faults are given. As we mentioned before, the number of faults attributed to each member approximates the capability of that member.

5.6 Analysis of the effort allocation methods

5.6.1 Simple comparison

In this subsection, we compare the three effort allocation methods by evaluating review effort and testing effort. Table 5.3 shows the average review cost, average testing cost, and metric M_f for each method. As mentioned in subsection 3.1.2, M_f, *error detection efficiency*, is a metric for evaluating review activities[Fagan 1976].

(A)Average review cost: There is little difference among the three methods. Method C took 767 min., Method P took 800 min., and Method U took

735 min.

(B)Average testing cost: From Table 5.3, Method C took 5,980 min., Method P took 6,498 min., and Method U took 7,783 min. Thus the following relation is derived:

$$\text{Method } C < \text{Method } P < \text{Method } U$$

(C)M_f(*error detection efficiency*): The same relation as in (B) is derived.

These results imply that teams using Methods C and P can test in a shorter time and efficiently detect more faults in review as compared to Method U.

5.6.2 Effects of team organization

In this subsection, we evaluate review activity using two metrics M_f and M_k. Then, we compare the three methods with respect to variances in the capabilities of team members.

The first metric is *error detection efficiency* [Fagan 1976], M_f, which is defined by Fagan (See subsection 3.1.2). By applying Fagan's metric to the software development process shown in Figure 2.1, we get the following equation. (See equation 3.2, also.)

$$M_f = \frac{E_{dr} + E_{cr}}{e_d + e_c}. \qquad (5.1)$$

The second metric is M_k[Kusumoto et al. 1992], which is defined in Chapter 4. In this experiment, we consider time as the cost. M_k is then specified by the following equation. (See equation (3.9), also.)

Table 5.4 Evaluated values for two metrics

Effort allocation methods	Team	M_f	M_k
Method *C*	*T1*	0.78	0.75
	T2	0.75	0.72
	T3	0.52	0.45
Method *P*	*T4*	0.80	0.78
	T5	0.56	0.47
	T6	0.70	0.66
	T7	0.48	0.39
Method *U*	*T8*	0.73	0.70
	T9	0.57	0.50
	T10	0.68	0.66
	T11	0.41	0.37
	T12	0.53	0.51
	T13	0.43	0.35

$$M_k = \frac{E_{dr} + E_{cr}}{e_d + e_c} \cdot \frac{c_t - c_r}{c_t}. \qquad (5.2)$$

In equation (5.2), c_r is the average cost to detect and remove a fault in reviews, and c_t is the average cost to detect and remove a fault in testing provided that $(E_{dr} + E_{cr})$ faults are detected by reviews.

Table 5.4 shows the values of two metrics M_f and M_k for each team. Observe that the rank order among teams for each method is almost the same for both metrics. This provides further evidence indicating that M_k is approximately equal to M_f under *Assumption 3* in subsection 3.2.3. Thus we use the value for M_k in the following discussions.

We compare the three methods with respect to team organization. We divided the thirteen teams into two categories: even teams and uneven teams, based on the variance of capabilities across members of a team. Intuitively speaking, an even team implies that all members in the team have about the same capabilities, and an uneven team implies there is a relatively large difference among the capabilities of members.

It is very difficult to determine the boundary between even and uneven teams. Based on the variances in Table 5.2, we extracted five even teams (*T7, T8, T9, T10* and $T13$) and five uneven teams (*T1, T2, T5, T6* and $T12$). Table 5.5 illustrates the distribution of teams. Teams $T1$ and $T2$ have high M_k values. Both teams were uneven and used Method C. This implies that it is very effective for uneven teams to concentrate their review effort on products created by low capability developers. This supports our hypothesis that products having low quality are created by developers with low capabilities.

On the other hand, teams $T8$, $T9$ and $T10$ have relatively high values of M_k. All these teams were even and used Method U. This implies that when

Table 5.5 Distribution of values M_k's with respect to team organization

Effort allocation methods	Capabilities of members in each team	
	Uneven teams	Even teams
Method *C*	*T1* 0.75 *T2* 0.72	
Method *P*	*T5* 0.47 *T6* 0.66	*T7* 0.39
Method *U*	*T12* 0.51	*T8* 0.70 *T9* 0.50 *T10* 0.66 *T13* 0.35

the difference in capabilities of team members is low, reviews are most effective if the same amount of effort is applied to each product.

$T13$ is an exceptional case and is analyzed in more detail below.

5.6.3 Individual case studies

We analyze why some special teams have very low values of M_k. From Table 5.4, it is clear that teams $T7$, $T11$ and $T13$ have low values for both metrics M_f and M_k. These teams detected only a small number of faults at review time (thus, relatively large numbers of faults remained in the product), and the times spent on reviews is about average. Thus we can conclude that members of teams T7, T11 and T13 did not execute their reviews as meticulously as the other teams.

On the other hand, teams *T1, T2, T6* and $T10$ have high values of M_k. This implies that if they choose an appropriate review method based on the team organization, then their reviews will be successful. That is, if the capabilities of members in a team is almost even, then the team should adopt Method U. In contrast, if the capabilities differ to a considerable extent, then the team should adopt either Method C or Method P.

Finally, the low value of Team $T12$ indicates that an inappropriate review method was used. Team $T12$ should adopt Method C or P rather than Method U.

Chapter 6

Modeling the Testing Process

6.1 A framework for measuring software development processes

6.1.1 Overview

In this Section, we propose a new framework for measuring software development processes [Matsumoto et al. 1993]. The key idea of the proposed framework is that all activities to be measured can be explained using a mathematical model of the process. In this framework, a Petri net model is used to model the process, since it is one of the most powerful models for representing concurrent processes such as those occurring in software development[Peterson 1981].

The framework consists of four steps: (1) process modeling, (2) metric definition, (3) process and metric implementation , and (4) process and metric execution (See Figure 6.1). The following subsections summarize these four steps.

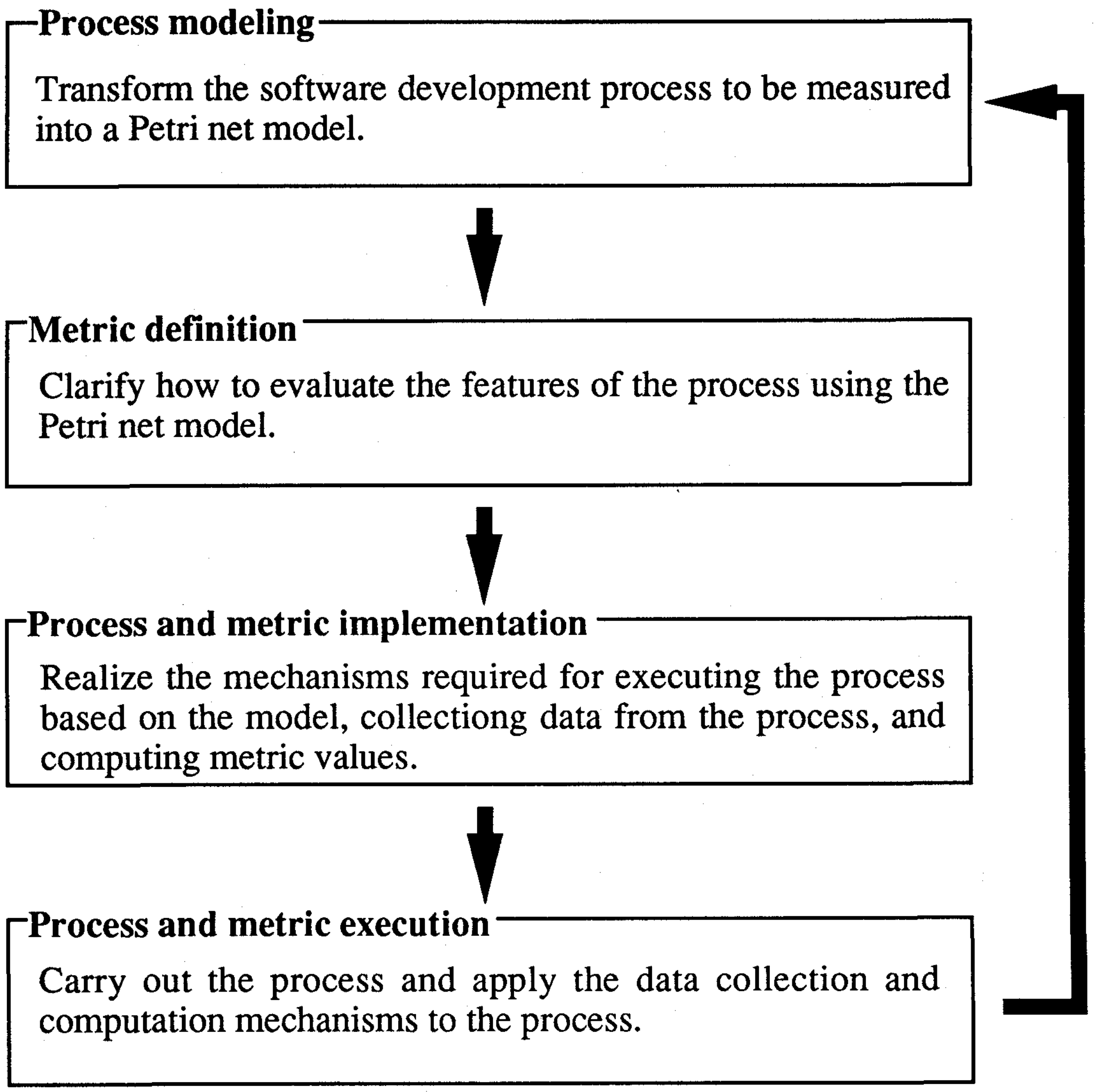

Process modeling
Transform the software development process to be measured into a Petri net model.
Metric definition
Clarify how to evaluate the features of the process using the Petri net model.
Process and metric implementation
Realize the mechanisms required for executing the process based on the model, collectiong data from the process, and computing metric values.
Process and metric execution
Carry out the process and apply the data collection and computation mechanisms to the process.

Figure 6. 1 The proposed framework for measuring software development processes

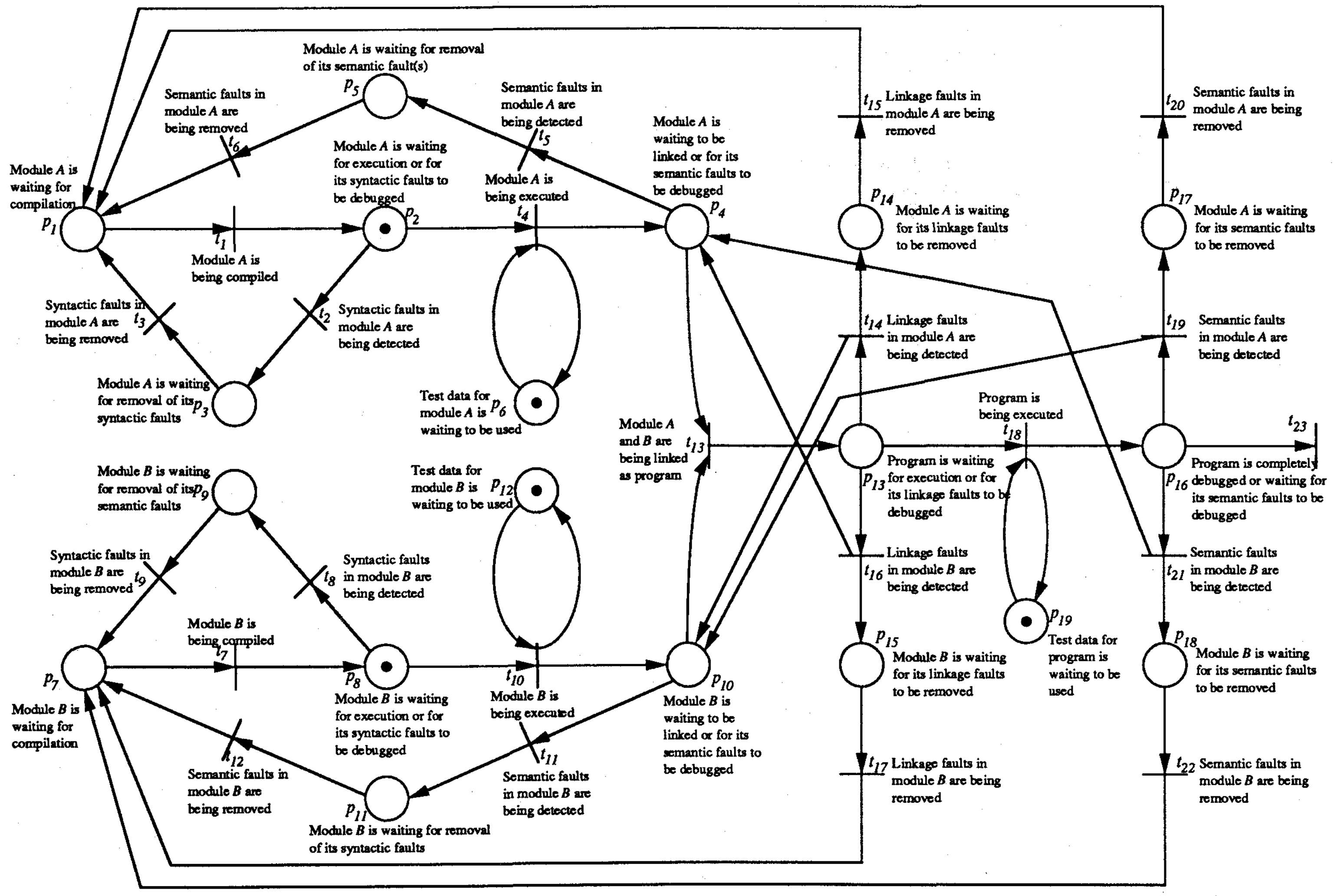
6.1.2 Process modeling

During process modeling, the software development process to be measured is transformed into a Petri net model. The Petri net model used in the proposed framework is "safe", that is, the number of tokens in each place never exceeds one.

The software development process consists of many activities whose interactions depend on the type of product being developed. In the framework, such interacting activities are modeled by transitions which represent nonprimitive events. Products are modeled using tokens. The places in the Petri net represent states (or conditions) of the process which are waiting for the execution of the activity in the process.

Figure 6.2 shows an example of a Petri net representing unit and integration testing processes. In this example, the program to be developed consists of two modules: module A and module B. Tokens in places p_2 and p_8 represent module A and module B, respectively. In this figure, both of the modules are waiting for execution in the unit testing. Tokens in places p_6, p_{12}, and p_{19} represent test data required for unit testing of module A, unit testing of module B, and for integration testing, respectively. In the current version of the proposed framework, we do not distinguish product types in the process.

Modules A and B are tested concurrently in each unit testing process. The unit testing process of module A is represented by a set of transitions $T_a = \{t_1, t_2, t_3, t_4, t_5, t_6\}$ and a set of places $P_a = \{p_1, p_2, p_3, p_4, p_5\}$. The unit testing process of module B is represented by a set of transitions $T_a = \{t_7, t_8, t_9, t_{10}, t_{11}, t_{12}\}$ and a set of places $P_a = \{p_7, p_8, p_9, p_{10}, p_{11}\}$. If both of them pass the unit testing, they are linked (transition t_{13}) and then tested (transition t_{18}).



Module A is waiting for compilation
p1
t1
Module A is being compiled
p2
Module A is waiting for execution or for its syntactic faults to be debugged
t2
Syntactic faults in module A are being detected
p3
Module A is waiting for removal of its syntactic faults
t3
Syntactic faults in module A are being removed
t4
Module A is being executed
p4
Module A is waiting to be linked or for its semantic faults to be debugged
t5
Semantic faults in module A are being detected
p5
Module A is waiting for removal of its semantic fault(s)
t6
Semantic faults in module A are being removed
p6
Test data for module A is waiting to be used
p7
Module B is waiting for compilation
t7
Module B is being compiled
p8
Module B is waiting for execution or for its syntactic faults to be debugged
t8
Syntactic faults in module B are being detected
p9
Module B is waiting for removal of its semantic faults
t9
Syntactic faults in module B are being removed
t10
Module B is being executed
p10
Module B is waiting to be linked or for its semantic faults to be debugged
t11
Semantic faults in module B are being detected
p11
Module B is waiting for removal of its syntactic faults
t12
Semantic faults in module B are being removed
p12
Test data for module B is waiting to be used
t13
Module A and B are being linked as program
p13
Program is waiting for execution or for its linkage faults to be debugged
t14
Linkage faults in module A are being detected
p14
Module A is waiting for its linkage faults to be removed
t15
Linkage faults in module A are being removed
t16
Linkage faults in module B are being detected
p15
Module B is waiting for its linkage faults to be removed
t17
Linkage faults in module B are being removed
t18
Program is being executed
p19
Test data for program is waiting to be used
p16
Program is completely debugged or waiting for its semantic faults to be debugged
t23
t19
Semantic faults in module A are being detected
p17
Module A is waiting for its semantic faults to be removed
t20
Semantic faults in module A are being removed
t21
Semantic faults in module B are being detected
p18
Module B is waiting for its semantic faults to be removed
t22
Semantic faults in module B are being removed

Figure 6.2 A Petri net model of unit and integration testing processes

In the model shown in Figure 6.2, "debugging processes" are represented by the following eight subsequences of transition firings:

(1) Debugging of syntactic faults:

$\delta_{1a} = t_2 t_3 t_1$ (module A)

$\delta_{1b} = t_8 t_9 t_7$ (module B)

(2) Debugging of semantic faults by unit testing:

$\delta_{2a} = t_5 t_6 t_1 \delta^*_{1a} t_4$ (module A)

$\delta_{2b} = t_{11} t_{12} t_7 \delta^*_{1b} t_{10}$ (module B)

(3) Debugging of linkage faults:

$\delta_{3a} = t_{14} t_{15} t_1 \delta^*_{1a} t_4 \delta^*_{2a} t_{13}$ (module A)

$\delta_{3b} = t_{16} t_{17} t_7 \delta^*_{1b} t_{10} \delta^*_{2b} t_{13}$ (module B)

(4) Debugging of semantic faults by integration testing:

$\delta_{4a} = t_{19} t_{20} t_1 \delta^*_{1a} t_4 \delta^*_{2a} t_{13} \delta^*_{3a} t_{18}$ (module A)

$\delta_{4b} = t_{21} t_{22} t_7 \delta^*_{1b} t_{10} \delta^*_{2b} t_{13} \delta^*_{3b} t_{18}$ (module B)

6.1.3 Metric definition

Metric definition clarifies how the features of the process depicted by the Petri net model will be evaluated. The metric is defined as a mathematical relationship among parameters which represent a structure or behavior of the Petri net model. We call such parameters "primitive parameters" and call metrics which include only primitive parameters "primitive metrics".

Primitive parameters in the current version of the proposed framework are summarized as follows:

(1) Primitive parameters of model structure[Peterson 1981]

$P = \{p_1, p_2, ..., p_n\}$: a finite set of places in the model.

$T = \{t_1, t_2, ..., t_m\}$: a finite set of transitions in the model.

$I : T \rightarrow P^{\infty}$: an input function, that is a mapping from transitions to bags of places. (the input places of the transition.)

$O : T \rightarrow P^{\infty}$: an output function, that is a mapping from transitions to bags of places. (the output places of the transition.)

$\mu = (\mu_1, \mu_2, ..., \mu_n)$: a vector of the number of tokens in the places.

n : the cardinality of the set P. (the number of places.)

m : the cardinality of the set T. (the number of transitions.)

$\#(p_i, I(t_j))$: the number of occurrences of a place p_i in the input bag of a transition t_j.

$\#(p_i, O(t_j))$: the number of occurrences of a place p_i in the output bag of a transition t_j.

(2)Primitive parameters of model behavior

$\mu(t) = (\mu_1(t), \mu_2(t), ..., \mu_n(t))$: a vector of the number of tokens in the places at time t.

$\delta_{ki} = t_{ki_1} t_{ki_2} ... t_{ki_l}$: a sequence of transition firings.

Γ_τ : a sequence of transition firings executed from time 0 through time t.

$N_p(p_i, \Gamma_\tau)$: the number of tokens residing in place p_i in Γ_τ.

$N_t(t_j, \Gamma_\tau)$: the number of occurrences of transition t_j in Γ_τ.

$N_\delta(\delta_{k1}, \delta_{k2}, \Gamma_\tau)$: the number of occurrences of firing sequence δ_{k1} in Γ_τ, where δ_{k1} is a subsequence of δ_{k2} and δ_{k2} is a subsequence of Γ_τ.

$E_p(p_i, \Gamma_\tau)$: total elapsed time in place p_i in Γ_τ.

$E_t(t_j, \Gamma_\tau)$: total elapsed time of transition t_j in Γ_τ.

$E_\delta(\delta_{k1}, \delta_{k2}, \Gamma_\tau)$: total elapsed time of firing sequence δ_{k1} in Γ_τ, where δ_{k1} is a subsequence of δ_{k2} and δ_{k2} is a subsequence of Γ_τ.

For example, eight sequences, $\delta_{1a}, \delta_{1b}, \delta_{2a}, \delta_{2b}, \delta_{3a}, \delta_{3b}, \delta_{4a}$, and δ_{4b}, of the transition firing shown in subsection 6.1.2, can be used as primitive parameters in the debugging processes. In addition, we can define the following primitive metrics using only these primitive parameters:

(1) The integration testing cost:

$$C_{it}(\Gamma_\tau) = E_\delta(\delta_{4a}, \Gamma_\tau, \Gamma_\tau) + E_\delta(\delta_{4b}, \Gamma_\tau, \Gamma_\tau)$$

(2) Productivity in unit and integration testing:

$$PRD(\Gamma_\tau) = \frac{\sum_{j=1}^{m} E_t(t_j, \Gamma_\tau)}{\sum_{i=1}^{n} E_p(p_i, \Gamma_\tau) + \sum_{j=1}^{m} E_t(t_j, \Gamma_\tau)}$$

(3) Unit testing cost of module A before integration testing:

$$C_{uta}(\Gamma_\tau) = E_\delta(\delta_{2a}, \Gamma_\tau, \Gamma_\tau) - (E_\delta(\delta_{2a}, t_{14}t_{15}t_1t_4\delta_{2a}, \Gamma_\tau) + E_\delta(\delta_{2a}, t_{19}t_{20}t_1t_4\delta_{2a}, \Gamma_\tau))$$

We give an intuitive explanation for the definition of $C_{uta}(\Gamma_\tau)$. Let Δ_{a1} be the total sum of elapsed times for firing sequence δ_{2a} in each firing subsequence $t_{14}t_{15}t_1t_4\delta_{2a}$ of sequence Γ_τ from time 0 through time t.

Similarly, let Δ_{a2} be the total sum of elapsed times of δ_{2a} for each firing subsequence $t_{19}t_{20}t_1t_4\delta_{2a}$ of sequence Γ_τ from time 0 through time t. If Γ_τ does not contain such a subsequence, then the value of Δ_{ai} (i=1,2) is zero and,

$$C_{uta}(\Gamma_\tau) = E_\delta(\delta_{2a}, \Gamma_\tau, \Gamma_\tau) - (\Delta_{a1} + \Delta_{a2})$$

(4)Unit testing cost of module B before integration testing:

$$C_{utb}(\Gamma_\tau) = E_\delta(\delta_{2b}, \Gamma_\tau, \Gamma_\tau) - (E_\delta(\delta_{2b}, t_{16}t_{17}t_7t_{10}\delta_{2b}, \Gamma_\tau) + E_\delta(\delta_{2b}, t_{21}t_{22}t_7t_{10}\delta_{2b}, \Gamma_\tau))$$

If the number of primitive parameters increases by extending the process model of the framework, then most of the conventional process metrics could be defined as primitive metrics. For example, if the product is modeled by a "colored token" in order to show explicitly what types of product are used in the process to be modeled, then we could define and use "product type" as a new primitive parameter, and we could then define some new primitive metrics.

6.1.4 Process and metric implementation

During process and metric implementation, mechanisms are realized for executing the process based on the model for collecting data from the process in order to obtain values of primitive parameters, and for computing values of primitive metrics.

One way to support process implementation is to provide developers with a tool-based software development environment. That is, devise or gather a set of software tools that correspond to all activities in the process to be implemented. Many primitive tools are already provided by operating systems. For example, tools for program compilation, linkage, and execution are provided as well-known operating system commands. Text editors are essential tools for creating and modifying program code, e.g., removing syntactic or semantic faults in the program code. If we can execute the process in a CASE environment, then it is relatively easy to find appropriate tools or commands for supporting the activities of the process. (In some cases, the tools provided in the development environment determine how the process will be modeled.) When there is an activity with no supporting tools then an alternative is to design a worksheet for the activity and devise a tool which can edit the worksheet. Such tools can be created by customizing ordinary text and/or graphical editors provided by the operating system.

During metric implementation, mechanisms should be devised that automatically and reliably collect the necessary data. This data should be collected without interrupting and restricting the developers' activities. However, for small-scale projects, the development systems are typically not very powerful computationally. Any data collection requiring large amounts of computation will interfere with program development. In this case, the project may not succeed and reliable data may not be collected. Therefore, mechanisms should be designed to minimize the impact of data collection on the system.

As mentioned in subsection 6.1.3, primitive parameters consist of two kinds of parameters: parameters of model structure and of model behavior. The values of parameters of model structure can be obtained from the definition of the model independent of the process implementation. On the other hand, the values relating to model behavior have to be obtained by processing the data collected from the process to be measured. In the current version of the proposed framework, all parameters can be obtained by clarifying when and which transition fires on the model as the process executes.

If we provide a set of software tools supporting all activities of the process to be implemented, then it would be relatively easy to know when and which transition fires on the model as the process executes. To do so, we only have to collect execution data from the tools that correspond to the transitions. One of the easiest ways to collect tool execution data is to use some existing functions provided by the accounting system of the operating system. For example, the UNIX operating system records the following six data items concerning command execution: (1) the command name, (2) the programmer's name, (3) the identifier of the terminal, (4) the amount of CPU time (in 1/100 seconds) necessary to execute the command, (5) the date, and (6) the time when the command was executed. These six data items are recorded for each command execution [Matsumoto 1990]. It is easy to extract the necessary information from them to know which transition is firing.

6.1.5 Process and metric execution

During process and metric execution, the process being modeled is executed and the data collection and computation mechanisms are applied to the executing process.

If a new and unfamiliar process is introduced into the project, developers must be trained in the process to avoid collecting meaningless or unreliable data. If data reliability is a major concern, the developers' activities can be restricted to prevent any deviations from the Petri net model of the process. For example, developers could be prohibited from using any tools and commands with no corresponding model transitions.

During metric execution, we can choose whether developers will be informed about what data is being collected and how to interpret their performance based on that data. Such a decision depends on the objective (or goal) of the project. For example, when the objective is to control the process, the collected data provides valuable feedback helping developers direct their activities towards accomplishing the project plan. On the other hand, when the objective is to investigate the features or characteristics of the process itself, then it is probably best not to inform developers since their knowledge could impact the data being collected. Of course, collecting data from developers without their knowledge requires devising a mechanism that collects data without interrupting and restricting the developers' construction activities.

On the completion of the process and metric execution, the various pieces of the framework (i.e. the model, the metric definitions, the mechanisms, the metric values, and the collected data) are assembled and packaged as a new experience for measuring the software development process.

6.2 Testing process

The activities on the terminal during the testing process consist of editing, compiling and executing programs. When editing or implementing programs, syntactic and semantic faults are introduced into the code. During compilation, source programs are translated into the executable code by the compiler. Finally, the correctness of the program can be validated by executing the program with some test data.

When syntactic faults are detected during compilation, these faults need to be isolated. Usually, this can be done using the compile list. Similarly, when semantic faults are detected during execution; i.e. a failure happens, the faults that triggered the failure also need to be isolated. These activities are repeated until no failures happen during program execution.

6.3 Petri net model of the testing process

Figure 6.3 shows the Petri net description of the activities presented in subsection 6.2. It includes the following four places and six transitions:

Places:

p_1: Program is waiting to be created.

p_2: Program is waiting for compilation.

p_3: Program is waiting for execution or for its syntactic faults to be debugged.

p_4: Program is completely debugged or waiting for its semantic faults to be debugged.

Transitions:

t_1: Program is being created.

t_2: Program is being compiled.

t_3: Program is being executed.

t_4: Syntactic faults in program are being fixed.

t_5: Semantic faults in program are being fixed.

t_6: Process is over.

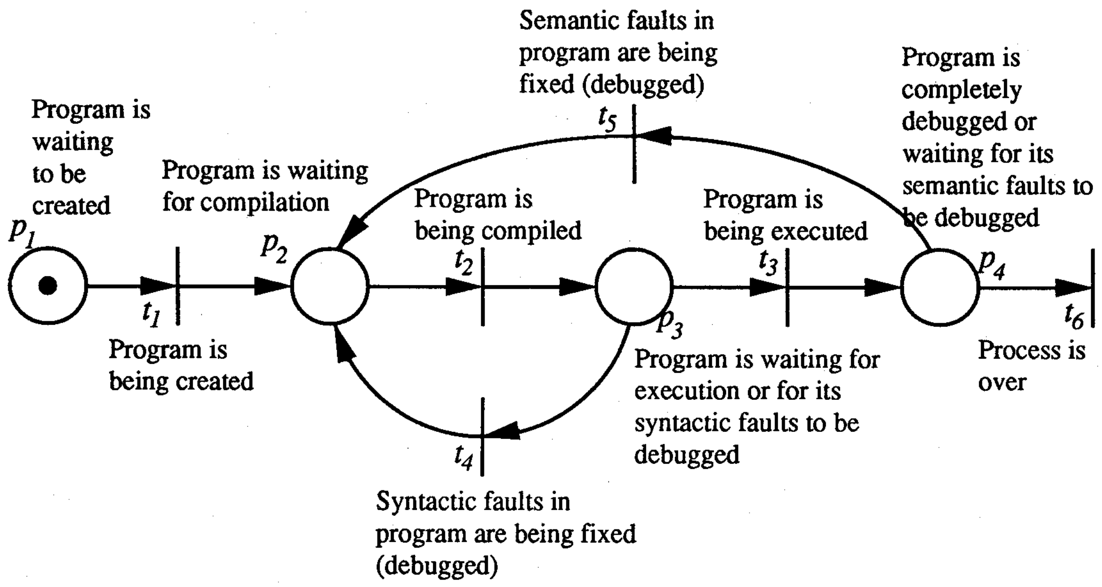

Semantic faults in program are being fixed (debugged)
t_5
Program is completely debugged or waiting for its semantic faults to be debugged
Program is waiting to be created
p_1
Program is waiting for compilation
p_2
Program is being compiled
t_2
Program is being executed
t_3
p_4
t_1
Program is being created
p_3
Program is waiting for execution or for its syntactic faults to be debugged
t_6
Process is over
t_4
Syntactic faults in program are being fixed (debugged)

Figure 6. 3 A Petri net model of a coding and debugging process in a student project

The token in place p_1 represents a program. Before firing transition t_1, this token represents an empty program containing no code. Test data, used in the execution of the program (represented by transition t_3), are omitted.

For example, the firing sequence $\delta_1 = t_1t_2t_4t_2t_3t_5t_2t_3t_6$ corresponds to the following sequence of basic activities:

(1)Editing the program

(2)Compiling the program

(3)Fixing the syntactic faults

(4)Re-compiling the program

(5)Executing the program (Failure happens.)

(6)Fixing the semantic faults

(7)Re-compiling the program

(8)Executing the program

(9)Process completes.

6.4 Example of firing sequences

In this Section, we discuss how the following three firing sequences (δ_2, δ_3, and δ_4) correspond to the behavior of the programmer.

(1) Ideal behavior

$$\delta_2 = t_1t_2t_3t_6$$

δ_2 is the process where no syntactic and semantic faults are detected. δ_2 can happen when a small program is being developed by an experienced programmer.

(2) Behavior of a novice programmer

$$\delta_3 = t_1t_2t_4t_2t_4t_2t_4t_2t_3t_5t_2t_3t_5t_2t_3t_5t_2t_4t_2t_4t_2t_3t_5t_2t_3t_6$$

The difference between δ_3 and δ_2 is that the sequences $t_4t_2(=\delta_{sy})$ and $t_5t_2t_3(=\delta_{sm})$ are inserted between t_1 and t_6. Each occurrence of δ_{sy} and δ_{sm} corresponds to fixing a syntactic or a semantic fault, respectively. Thus, δ_3 illustrates the development process of a novice programmer where numerous fault fixing activities are executed repeatedly.

(3) Behavior of an average programmer

$$\delta_4 = t_1t_2t_4t_2t_4t_2t_3t_5t_2t_3t_5t_2t_3t_5t_2t_3t_6$$

In δ_4, as with δ_3, δ_{sy} and δ_{sm} are inserted between t_1 and t_6. But, for δ_4 there are fewer occurrences of δ_{sy} than in δ_3. δ_4 illustrates a development process with fewer syntactic faults but still many semantic faults; i.e. δ_4 depicts the average programmer.

6.5 Metrics for the testing process

As described in Section 6.4, the model includes two types of debugging processes. They are represented by the following sequences of transition firings:

(1) Debugging of syntactic faults: $\delta_{sy} = t_4t_2$

(2) Debugging of semantic faults: $\delta_{sm} = t_5t_2\ d_{sy}^*t_3$

We defined the following primitive metrics using only the primitive parameters δ_{sy} and δ_{sm}. In these definitions, τ_e denotes the time it takes the process to complete, and thus G_τ (in subsection 6.1.3) $= G_{\tau_e}$.

(1) Total cost of the process:

$$C = \sum_{j=1}^{5} E_t(t_j, \Gamma_{\tau_e})$$

(2) Productivity of the process:

$$PRD = \frac{\sum_{j=1}^{5} E_t(t_j, \Gamma_{\tau_e})}{\sum_{i=1}^{4} E_p(p_i, \Gamma_{\tau_e}) + \sum_{j=1}^{5} E_t(t_j, \Gamma_{\tau_e})}$$

(3) Total cost for debugging syntactic faults:

$$\begin{aligned} C_{sy} &= E_\delta(\delta_{sy}, \Gamma_{\tau_e}, \Gamma_{\tau_e}) \\ &= E_\delta(t_4, \Gamma_{\tau_e}, \Gamma_{\tau_e}) + E_\delta(t_2, t_4t_2, \Gamma_{\tau_e}) \\ &= E_\delta(t_4, \Gamma_{\tau_e}) + E_\delta(t_2, t_4t_2, \Gamma_{\tau_e}) \end{aligned}$$

(4) Average cost for debugging syntactic faults during a single execution:

$$\begin{aligned} c_{sy} &= \frac{C_{sy}}{N_\delta(\delta_{sy}, \Gamma_{\tau_e}, \Gamma_{\tau_e})} \\ &= \frac{E_\delta(t_4, \Gamma_{\tau_e}) + E_\delta(t_2, t_4t_2, \Gamma_{tau_e})}{N_t(t_4, \Gamma_{\tau_e})} \end{aligned}$$

(5) Total cost for debugging semantic faults:

$$\begin{aligned} C_{sm} &= E_\delta(t_5t_2, \delta_{sm}, \Gamma_{\tau_e}) + E_\delta(t_3\delta_{sm}, \Gamma_{\tau_e}) \\ &= (E_\delta(t_5, \Gamma_{\tau_e}, \Gamma_{\tau_e}) + E_\delta(t_2, t_5t_2, \Gamma_{\tau_e})) + E_\delta(t_3, \Gamma_{\tau_e}, \Gamma_{\tau_e}) \\ &= E_t(t_5, \Gamma_{\tau_e}) + E_\delta(t_2, t_5t_2, \Gamma_{\tau_e}) + E_t(t_3, \Gamma_{\tau_e}) \end{aligned}$$

(6) Average cost for debugging semantic faults during a single execution:

$$
\begin{aligned}
c_{sm} &= \frac{C_{sm}}{N_\delta(\delta_{sm}, \Gamma_{\tau_e}, \Gamma_{\tau_e})} \\
&= \frac{E_t(t_5, \Gamma_{\tau_e}) + E_\delta(t_2, t_5 t_2, \Gamma_{\tau_e}) + E_t(t_3, \Gamma_{\tau_e})}{N_t(t_5, \Gamma_{\tau_e})}
\end{aligned}
$$

Chapter 7

Experimental Evaluation of the Testing Process in a Student Project

7.1 Outline

In this Section, we present an example application of the proposed framework. We evaluated the effectiveness of the design method in an academic environment by examining its effect on debugging activities. The experiment was conducted using participants from the Department of Information and Computer Sciences at Osaka University. The following steps outline the experimental process:

Step 1: Process modeling

Based on observations in the academic environment extending over several years, we modeled the coding and debugging processes of student participants as a Petri net(See Figure 6.3) consisting of four places and six transitions. The token in the model represents the program to be developed.

Step 2: Metric definition

We defined six metrics for evaluating the process(See Section 6.5). One of them evaluates the cost of executing the process in terms of the total elapsed time of transitions in the model. Another metric computes productivity in the process based on the elapsed time of both places and transitions. The other four metrics evaluate the total cost and average cost to debug syntactic and semantic faults.

Step 3: Process and metric implementation

We specified a set of UNIX[UNIX 1986] commands which correspond to all activities of the process to be implemented. Next, we devised a data collection mechanism using a measurement environment called GINGER which automatically collects data concerning the activities of programmers during software development in the UNIX environment [Kusumoto et al. 1991] [Torii et al. 1990].

Step 4: Process and metric execution

Two projects (Project 1 and Project 2) were carried out and data were collected using participants from the Department of Information and Computer Sciences at Osaka University. Twelve students participated in both projects. In Project 1, students developed programs using no particular design method. In Project 2, the same students developed programs by using the Structured Design Method [Yourdon & Constantine].

Step 5: Interpretation

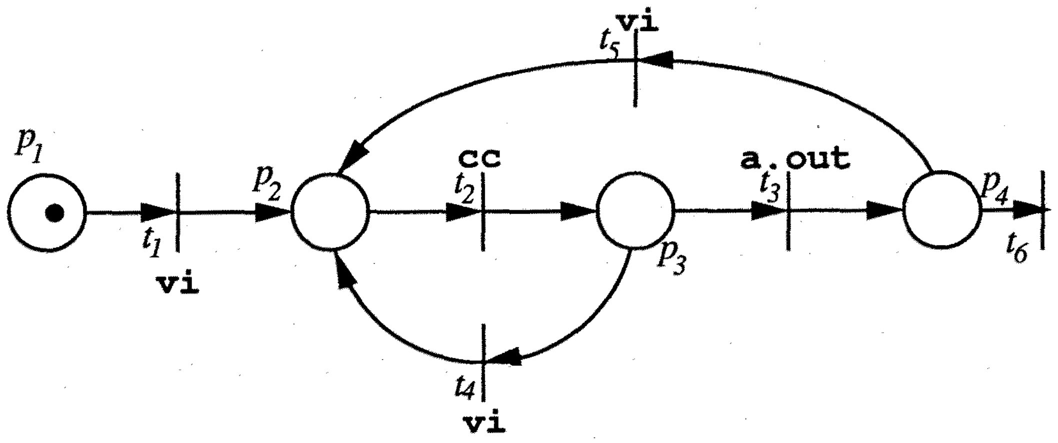


Figure 7.1 A correspondence of transitions and commands

We examined the impact using the Structured Design Method had on the debugging processes in the two projects. Specifically, we compared the values of primitive metrics generated in these two projects.

7.2 Implementation on UNIX environment

We implemented the mechanisms for executing the process model in a UNIX environment. Our task was simplified because the tools necessary for program creation, compilation, execution, and debugging were already provided by UNIX and the participants in this experiment were already familiar with these tools. The following mappings illustrate the correspondence between model transitions and well-known UNIX commands (See Figure 7.1):

t_1: vi (Program is being created.)

```
1988-11-11 11:42 11:56 vi       1.38 secs
1988-11-11 11:58 11:58 cc       2.71 secs
1988-11-11 11:58 11:59 vi       1.95 secs
1988-11-11 12:01 12:01 cc       2.69 secs
1988-11-11 12:02 12:02 a.out    0.78 secs
1988-11-11 12:07 12:14 vi       4.61 secs
```

Figure 7.2 Example of command execution data provided by *GINGER*

t_2: cc (Program is being compiled.)

t_3: a.out (Program is being executed.)

t_4: vi (Syntactic faults in program are being fixed (debugged).)

t_5: vi (Semantic faults in program are being fixed (debugged).)

t_6: none (Process is over.)

In order to know when a particular transition fires as the process executes, we need to collect data concerning the execution of the commands: "vi", "cc", and "a.out". The data collection mechanism uses a measurement environment called GINGER which automatically collects data concerning the activities of programmers during software development [Kusumoto et al. 1991] [Torii et al. 1990]. GINGER can collect data about command execution using existing functions available in an accounting system in the UNIX environment.

GINGER provides us with selected command execution data including: the execution date, the time the command started, the time the command exited, the name of the command, and the amount of CPU time used (in 1/100 seconds). All times are reported in 24-hour time format except CPU time. Figure 7.2 shows an example of the data provided by GINGER [Kusumoto et al. 1991] [Matsumoto 1990].

7.3 GINGER system

7.3.1 System organization

Figure 7.3 outlines the system architecture of GINGER and describes the information flow between the development environment and the management environment.

(A) Data Collection Unit

The (CM1) Process Management component supports and controls a programmer's regular activities during software development. The (CM2) Product Management component maintains the product, i.e., the program text developed by the programmer. The (CM3) Process Data Collection component accumulates data concerning the programming efforts. The (CM4) Product Data Collection component accumulates a series of intermediate programs (including the final program) and collects historical data about program modifications.

(B) Data Management Unit

The (CM5) Data Compression component saves memory by storing as little data as possible. For a series of intermediate products, only the

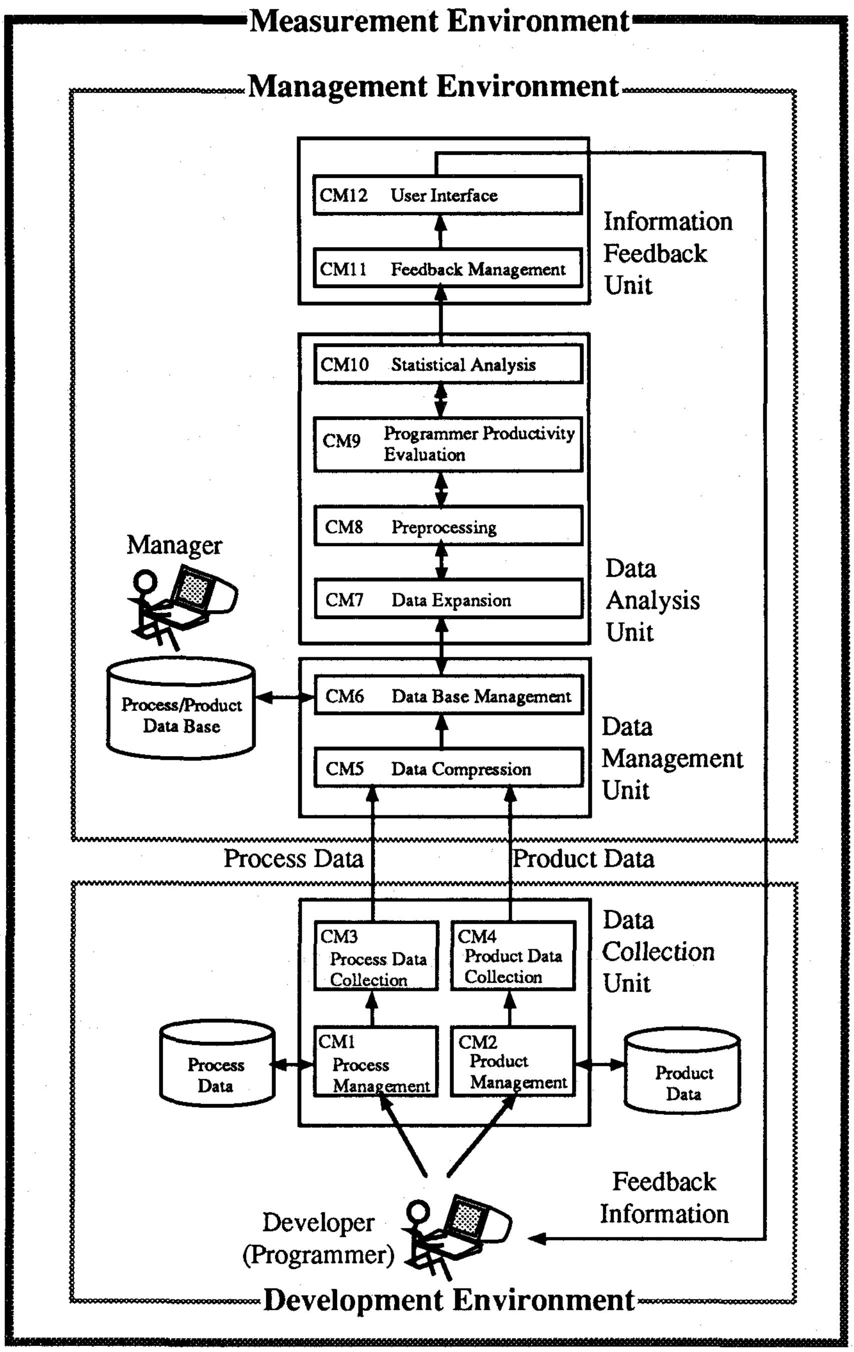


Figure 7.3 System architecture of the measurement environment

difference between the two is stored. The (CM6) Data Base Management component supports data storage and information retrieval. The data from the Data Collection Unit and the information from the Data Analysis Unit are stored in the Process/Product Data Base.

(C) Data Analysis Unit

The (CM7) Data Expansion component restructures the data by reinserting the compressed parts previously deleted by the Data Compression component. The (CM8) Preprocessing component prepares the expanded data for evaluation. Preprocessing includes transforming the data which was collected in the physical unit (i.e., file) into data for the logical unit (i.e., module). The (CM9) Programmer Productivity Evaluation component calculates several values according to the algorithms or guidelines for measurement.

The (CM10) Statistical Analysis component applies statistical analyses to the collected data (CM8) and to the results of the evaluation (CM9). In these analyses, historical data on the programmer and data on similar prior projects may be used extensively.

(D) Information Feedback Unit

The (CM11) Feedback Management component determines the timing of feedback and the details of information to be returned. Feedback is also provided whenever the programmer requests it. The (CM12) User Interface component manages the presentation of feedback information to the programmer.

7.3.2 Implementation

(A)Characteristics of the prototype

The first prototype system is currently being developed in a UNIX environment. The main characteristics of the prototype's implementation are summarized below:

(I1) The system has been implemented in C since portability is an important issue in a data collection and analysis environment.

(I2) Many of the functions and tools provided in UNIX have been integrated into the prototype. In particular, when implementing the Data Collection, Data Management and Information Feedback Units, provided UNIX functions and tools are used as much as possible.

(I3) The environment is running on a local area network of workstations linked by an Ethernet. The Data Collection Unit is implemented on the programmers' workstations, and all other units are implemented on the manager's workstation, as shown in Figure 7.3.

(B)Data collection

(1) Process data

The process data consists of two kinds of data: terminal access times and command execution data. These data are accumulated by the Process Data Collection component which is (CM3) based on the UNIX accounting system. This step can be considered as Process Management (CM1) in the prototype system. Accounting data for each programmer are obtained using the commands "last" and "lastcomm" provided by the accounting system.

(2) Product data

All of the program texts are collected as process data whenever they are updated. Modification times are also collected. The program text is managed as files on UNIX. For each file, relevant data such as date, time and access rights are maintained by the file system. We check the file update times recorded by the UNIX file system at five minutes intervals. All files that were updated within the past five minutes are collected by the Product Data Collection component (CM4).

(C)Data management

A basic function of the Data Management Unit in the prototype system is management of the Process/Product Data Base (See Figure 7.3). Another function of the Data Management Unit is compression of the product data. As mentioned in (B), all files are collected as process data whenever they are updated. Since this is a very large amount of data, it is impractical to store all of them in their original form.

The product data, that is, all of the updated files and their update times are compressed into a history of modifications plus the latest version of the file by the Data Compression component (CM5). The history of modifications are accumulated by computing the difference between the new version of the file and the last version of the file using the file comparator, "diff". This is done whenever the file is updated within the past five minutes.

(D)Data analysis

The analysis generates information on programming efforts, the quality/quantity of the resulting program, and program modifications. An example analysis is shown in Figure 7.4.

The following four kinds of measures concerning programming efforts are currently calculated: (1) calendar days, (2) total terminal access time (in minutes), (3) total number of commands that were executed, and (4) total CPU time (in 1/100 seconds) for command execution. The latter two are further classified in four ways, depending on the type of command: program editing, compilation, linking, and execution. These values are calculated based on terminal access data and command execution data.

The quality/quantity of the final program is calculated from the latest version (Fl) of the file in product data. Currently, for measuring quantity, two concrete measures are adopted: the total number of lines and the total number of modules. As for quality, the ratio of successful tests to the total number of test cases and test coverage are used.

Finally, program modification statistics are calculated based on the history of modifications in the product data. For basic measures concerning program modifications, the six $L's$ (L_i, L_a, L_c, $L_{c'}$, L_d, L_r) shown in Figure 7.4 are currently adopted according to the type of modification.

Additionally, the number of errors removed during the program development process is estimated (See Figure 7.4). The sum of the error

Programming Efforts

Calendar days	10
Total terminal access time (min.)	1231
Total number of commands executed	115
- Program editing	42
- Program compilation	30
- Program linking	18
- Program execution	25
Total CPU time for command execution (sec.)	323.90
- Program editing	251.03
- Program compilation	31.12
- Program linking	26.13
- Program execution	15.62

Quality/Quantity of Resulting Program

Total number of lines	326
The number of modules	21
Rate of successful test	100%

Program Modifications

The number of lines of the initial program (Li)	285
The number of lines appended to program (La)	97
The number of lines deleted by change (Lc)	178
The number of lines appended by change (Lc')	163
The number of lines deleted from program (Ld)	41
The number of lines of the resulting program (Lr)	326
Total number of errors (estimated value)	127
Sum of error life span (estimated value)	3291

Figure 7. 4 Example of measurement

life span [Matsumoto 1990], which has been introduced as a measure of programmer productivity, is also estimated. An error life span (T_e) for an error e is defined as the length of time from when error e is first manifested to when error e is removed. The number of errors and the error life span can be calculated automatically from the program modifications.

In order to estimate these two measures automatically, the followings two assumptions are made:

(1) The purpose of modifying program text at each edit session is to remove errors.

(2) The set of lines which is created at one edit session and modified at another edit session corresponds to one error.

From these assumptions, we define the number of errors removed at the edit session as the number of the sets described in assumption (2). Then, summing the errors for all edit sessions yields the total number of errors removed during the program development process. Next, for each set of lines, we can calculate a time duration from when lines belonging to the set were created to when they were removed. Then we define the sum of error life span to be the time durations thus calculated.

(E)Information feedback

In the prototype system, the programmer can retrieve the collected data and the analysis results stored in the process/product Data Base. But, it is difficult for programmers to understand and control their own activities given only these raw data. Therefore, the prototype system provides a mechanism to make these raw data more understandable us-

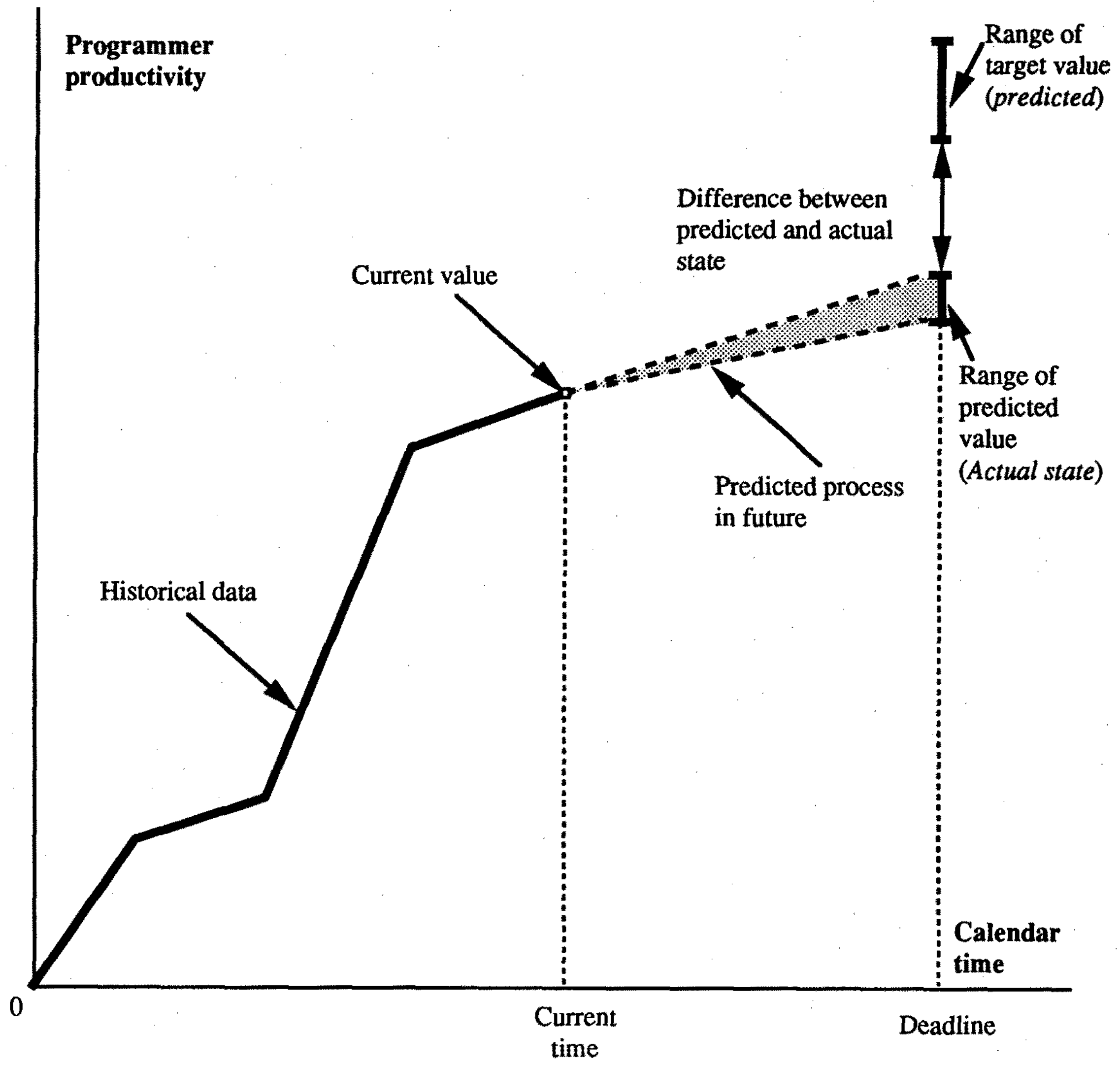

Programmer productivity
Range of target value (*predicted*)
Difference between predicted and actual state
Current value
Range of predicted value (*Actual state*)
Predicted process in future
Historical data
Calendar time
0
Current time
Deadline

Figure 7. 5 Conceptual drawing of feedback information

ing the Feedback Management (CM11) and User Interface (CM12) components. An example of the information provided is shown in Figure 7.5. Figure 7.5 represents a conceptional drawing of the feedback information provided by GINGER.

The results of some experimental evaluations showed that the feedback information shown in Figure 7.5 helps programmers manage their own activities and thus improves programmer productivity [Torii et al. 1990].

7.4 Experimental data in two projects

In order to evaluate the impact of the design method on debugging activities in an academic environment, two projects (Project 1 and Project 2) were carried out in Department of Information and Computer Sciences at Osaka University. The main characteristics of both Project 1 and Project 2 are summarized below:

(1) Twelve programmers participated. All were undergraduate students in the Department of Information and Computer Sciences at Osaka University.

(2) Each programmer developed an inventory control program in Project 1 and a data processing program in Project 2. In both projects, they developed their programs based on the given specification using the C language.

(3) The resulting programs contained about 300 lines of code in both projects.

(4) Project 2 was conducted about 2 months after the completion of Project 1. In Project 1, programmers designed their programs in ad hoc approach.

In Project 2, programmers designed their programs using the Structured Design Method.

Table 7.1 and Table 7.2 show the values of primitive parameters for twelve programmers upon completion of Project 1 and Project 2. These values are used to compute the primitive metrics defined in Section 6.5. Table 7.3 and Table 7.4 show the values of the primitive metrics for the same twelve programmers.

7.5 Results and interpretation

In this subsection, we present how the framework makes it possible to analyze the software development process in more detail as compared with other conventional approaches. To compare the two projects, the averages of C, PRD, C_{sy}, C_{sm}, c_{sy}, and c_{sm} for each project were calculated and summarized in Table 7.5. From Table 7.5, the following relation ($R1$) is derived with respect to the average total cost (C). [1]

$$\text{(}R1\text{) Average total cost } (C)\text{:}$$
$$\text{Project 1} > \text{Project 2}$$

Namely, the total cost of Project 2, in which each student used the Structured Design Method, is less than the total cost of Project 1, in which no design method was used. Total cost is one of the most well-known metrics for evaluating the process.

[1]Relations from ($R1$) through ($R6$) shown in this Section are confirmed by the paired-difference test. The level of significance was chosen as 0.05.

Table 7.1 Primitive parameters in Project 1

Student	$E_p(p_1,G_{t_e})$	$E_p(p_2,G_{t_e})$	$E_p(p_3,G_{t_e})$	$E_p(p_4,G_{t_e})$	$E_t(t_1,G_{t_e})$	$E_t(t_2,G_{t_e})$	$E_t(t_3,G_{t_e})$	$E_t(t_4,G_{t_e})$	$E_t(t_5,G_{t_e})$	$E_d(t_2,t_4,G_{t_e})$	$E_d(t_2,t_5,G_{t_e})$	$N_t(t_4,G_{t_e})$	$N_t(t_5,G_{t_e})$
#1	6	1770	104	27	29	97	53	49	446	44	53	44	53
#2	1	562	32	11	2	91	44	31	40	47	43	47	43
#3	3	1070	51	62	5	81	49	49	462	32	49	32	49
#4	5	483	49	100	39	50	35	25	227	15	34	15	34
#5	1	1290	82	134	35	93	45	47	184	48	44	48	44
#6	1	975	59	26	19	141	66	80	303	75	65	75	65
#7	4	1069	29	18	83	28	12	34	323	16	11	16	11
#8	2	811	14	88	73	40	12	187	187	28	12	28	12
#9	2	1197	76	36	3	63	27	68	398	36	26	36	26
#10	1	1263	117	249	12	227	135	63	194	92	135	92	135
#11	1	222	19	62	42	43	23	24	145	20	23	20	23
#12	1	1059	27	53	22	66	41	14	140	25	41	25	41

Table 7.2 Primitive parameters in Project 2

Student	$E_p(p_1,G_{t_e})$	$E_p(p_2,G_{t_e})$	$E_p(p_3,G_{t_e})$	$E_p(p_4,G_{t_e})$	$E_t(t_1,G_{t_e})$	$E_t(t_2,G_{t_e})$	$E_t(t_3,G_{t_e})$	$E_t(t_4,G_{t_e})$	$E_t(t_5,G_{t_e})$	$E_d(t_2,t_4,G_{t_e})$	$E_d(t_2,t_5,G_{t_e})$	$N_t(t_4,G_{t_e})$	$N_t(t_5,G_{t_e})$
#1	3	415	30	14	210	46	19	64	91	27	18	27	18
#2	1	170	18	6	1	14	4	41	37	10	4	10	4
#3	1	628	59	5	52	30	8	207	38	22	8	22	8
#4	2	416	49	38	3	50	22	65	102	28	22	28	22
#5	1	367	35	24	40	40	23	96	83	17	23	17	23
#6	1	226	10	1	1	16	5	59	18	11	5	11	5
#7	4	224	20	2	46	16	2	58	9	14	2	14	2
#8	1	454	8	3	40	11	2	217	4	9	2	9	2
#9	1	342	32	23	49	27	15	83	35	12	14	12	14
#10	1	796	69	25	47	102	31	243	251	71	31	71	31
#11	1	220	27	16	2	17	7	39	24	10	7	10	7
#12	2	352	33	17	166	32	16	21	95	16	16	16	16

Table 7.3 Primitive metrics in Project 1

Student	C	PRD	C_{sy}	c_{sy}	C_{sm}	c_{sm}
#1	674	0.26	93	2.11	552	10.42
#2	208	0.26	78	1.66	127	2.95
#3	646	0.35	81	2.53	560	11.43
#4	376	0.37	40	2.67	296	8.71
#5	404	0.21	95	1.98	273	6.20
#6	609	0.36	155	2.07	434	6.68
#7	480	0.30	50	3.13	346	31.45
#8	499	0.35	215	7.68	211	17.58
#9	559	0.30	104	2.89	451	17.35
#10	631	0.28	155	1.68	464	3.44
#11	277	0.48	44	2.20	191	8.30
#12	283	0.20	39	1.56	222	5.41

Table 7.4 Primitive metrics in Project 2

Student	C	PRD	C_{sy}	c_{sy}	C_{sm}	c_{sm}
#1	430	0.48	91	3.37	128	7.11
#2	97	0.33	51	5.10	45	11.25
#3	335	0.33	229	10.41	54	6.75
#4	242	0.32	93	3.32	146	6.64
#5	282	0.40	113	6.65	129	5.61
#6	99	0.29	70	6.36	28	5.60
#7	131	0.34	72	5.14	13	6.50
#8	274	0.37	226	25.11	8	4.00
#9	209	0.34	95	7.92	64	4.57
#10	674	0.43	314	4.42	313	10.10
#11	89	0.25	49	4.90	38	5.43
#12	330	0.45	37	2.31	127	7.94

But, comparing only the total cost of the process across the two projects does not reveal why the Structured Design Method reduced the cost. Nor can we examine the validity of the experimental results.

The primitive metrics defined in Section 6.5 provide us with valuable information about the relation between the usage of the Structured Design Method and the reduction of the total cost. For example, from Table 7.5, the following relation ($R2$) is derived with respect to the average productivity (PRD).

($R2$) Average productivity (PRD):

Project 1 $\cong$ Project 2

The relation ($R2$) tells us that students worked just as intensively in Project 1 as they did in Project 2. Therefore, the higher cost of Project 1 cannot be attributed to more waiting or idle time. It may be useful to look more closely at changes in the process itself.

With respect to the process of debugging syntactic faults, the following relations ($R3$) and ($R4$) are derived from Table 7.5.

($R3$) Total cost for debugging syntactic faults(C_{sy}):

Project 1 $\cong$ Project 2

($R4$) Average cost for debugging syntactic faults during a single execution(c_{sy}):

Project 1 $<$ Project 2

If we assume that the cost for debugging a syntactic fault is almost the same in both projects, we can say from the relation ($R3$) that students made about the same number of syntactic mistakes in both projects. From the relation ($R4$), we conclude that students examined the compiler's output, which

Table 7.5 Average values of primitive metrics in two projects

	C	PRD	C_{sy}	c_{sy}	C_{sm}	c_{sm}
Project 1	471	0.31	96	2.68	344	10.83
Project 2	266	0.36	120	7.08	91	6.79

included relevant debugging information, more carefully and fixed more faults during a single compilation in Project 2. However, these changes do not contribute to the cost reduction of the process, and, of course, these changes were not due to the usage of the Structured Design Method.

With respect to the process of debugging semantic faults, the following relations ($R5$) and ($R6$) are derived from Table 7.5.

($R5$) Total cost for debugging semantic faults(C_{sm}):

Project 1 $>$ Project 2

($R6$) Average cost for debugging semantic faults during a single execution(c_{sm}):

Project 1 $\cong$ Project 2

If we assume that the cost of debugging a semantic fault is the same in both projects, we conclude from relation ($R5$) that students made fewer semantic mistakes and spent less time debugging their semantic faults in Project 2. From the relation ($R6$), we conclude that students fixed approximately the same number of semantic faults per execution in both projects.

We, therefore, conclude that using the Structured Design Method reduced the total cost of the coding and debugging processes in these student projects. That is, the Structured Design Method prevented occurrences of semantic faults and reduced the total cost of debugging semantic faults. However, the method did not provide any additional information to help novice programmers detect and remove semantic faults from their programs. The results of our experiment may be considered "commonsense", but the point we wish to emphasize is that the proposed framework made it possible to systematically measure the software development process and made it easy to interpret the experimental results.

Chapter 8

Conclusion

8.1 Summary of major results

In this thesis, a new metric M_k is proposed for measuring review activities. This metric monitors the cost of detecting and removing faults during technical reviews. We have evaluated the usefulness of M_k using experimental data collected from software projects in a computer company. Moreover, an analysis of the relationship between the values collected from reviews and the values from testing made it possible to present a method for estimating the value of M_k by using the values collected from reviews only. As the value of M_k is normalized by the virtual testing cost, we can compare review metrics across projects. Using data from many projects, it may be possible to establish a criterion for determining when project review activities should be terminated.

Then, we used M_k to compare three methods for allocating review effort: a proportional distribution of effort(Method P), a concentrated distribution of effort(Method C) and a uniform distribution of effort(Method U). This analysis revealed that the most effective review effort allocation method was dependent on the type of team organization. If the capabilities of members in a team are almost even, then the team should adopt Method U. If the capabilities of members in a team are different to a considerable extent, then the team should adopt either Method C or Method P. So, if we can evaluate the developers' capabilities quantitatively and select the appropriate effort allocation method, then the effectiveness of review activities can be substantially improved.

Finally, we described a new framework for measuring software development processes. The key idea of this proposed framework is that all activities to be measured can be explained by a mathematical model of the process to be measured. A Petri net is used for modeling in the proposed framework. We presented an example application of the proposed framework where we examined the impact of the Structured Design Method on testing activities in an academic environment. The example showed that the framework can make it possible to measure software development processes in a systematic way and can make it easy to interpret experimental results.

In this thesis, we have focused on the software review and testing processes. However, our framework can also be used to quantitatively compare projects if the projects and their collected data are defined by common rules or models. Additionally, a Reference database [Basili & Cantone 1992] for software measurements, which consists of a set of reference projects and their data, can be formed on the basis of a few projects that are designed using standard software processes, products, tools, and environments, executed and analyzed. Such a database would allow future research ideas or technologies to be easily evaluated against existing data without running expensive and time-consuming experiments. This would enhance the exchange of experimental results and knowledge between researchers and practitioners of software engineering. The proposed framework is an important step towards establishing such the Reference database for software measurements. Because, in the proposed framework, software development processes are specifically defined by a mathematical Petri net model, and software metrics are explicitly defined as mathematical relationships among parameters representing a structure or behavior of the Petri net model.

8.2 Future work

Future research work includes the following:

(1)Extension of Metric M_k

In this thesis, we have assumed that after the test/debug cycle, no more faults remain in the software. But in a real environment containing tens of thousands of lines of code, this assumption probably does not hold. In order to deal with such cases, we are going to extend M_k such that M_k can be applied to the residual faults detected after testing. It is very important to collect data on residual faults and apply this data towards extending M_k.

In this thesis, we also assume that the average cost to detect and remove a design fault in testing versus a coding fault is the same. Usually, the cost of detecting and removing a design faults are much larger than for coding faults. So, we should take into account this cost difference for removing faults detected by design reviews, code reviews and testing, respectively. It is essential to collect and analyze data about the cost of each type of fault.

(2)Data collection and analysis

In order to evaluate the entire software development process, it is essential to assure the reliability of data from the process. We have to devise automatic data collection mechanisms that do not interrupt or restrict developers' activities. Automatic data collection could adversely affect system performance on less powerful platforms and thus interface with program development. In such cases, the project may not succeed and reliable data may not be collected. Therefore, mechanisms must be designed so as to minimize the impact of automatic data collection on system performance.

We already use the measurement environment GINGER that automatically collects and analyzes data concerning software development. The current prototype of GINGER is mainly intended for the implementation(coding) and testing phases on the workstation. New mechanisms in GINGER need to be developed to collect and analyze data from the requirements and design phases.

(3)Evaluation of research methodology

Numerous studies in software engineering develop new methods, tools, or techniques to improve some aspect of software development or maintenance. However, relatively little evidence has been gathered on which of these new developments are effective[Tichy et al. 1993]. We have examined the cost-effectiveness of software reviews in this thesis.

It is very important to carefully evaluate the research results. Towards this end, we have to define the problem being addressed, specify the assumptions, and clearly state the hypotheses. We consider the framework used in Chapters 6 and 7 to be useful for systematically evaluating them. For example, consider creating a transition representing the design activity. Many design methods such as JSD ($Jackson\ System\ Design$), SD ($Structured\ Design$), OOD ($Object - Oriented\ Design$) have already been proposed. The steps for any one of these methods must be specified. Once the steps are specified, we can apply our framework and evaluate the effectiveness of the design method quantitatively. Eventually, we will extend our Petri net model to describe all aspects of software development, such as the software product, process, tools, human resources, and environments.

Bibliography

[Aoyama & Chang 1992] M. Aoyama and C. K. Chang :“A Petri net based platform for developing communication software systems,” *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences,* Vol.E75-A, No.10, pp.1348- 1359(1992).

[Basili & Musa 1991] V.R.Basili and J.D.Musa:“The future engineering of software: A management perspective,” *IEEE Computer,* Vol.24, No.9, pp.90-96(1991).

[Basili & Rombach 1988] V. R. Basili and H. D. Rombach : “The TAME project: Towards improvement-oriented software environment,” *IEEE Transactions on Software Engineering,* Vol.14, No.6, pp.758-773(1988).

[Basili 1989] V. R. Basili: “Software development: A paradigm for the future,” *Proceedings of the 13th International Computer Software and Application Conference,* pp.471-483(1989).

[Basili & Cantone 1992] V. R. Basili and G. Cantone: “A reference architecture for the component factory,” *ACM Transactions on Software Engineering and Methodology,* Vol.1, No.1, pp.53-80(1992).

[Bisant & Lyle 1989] D.B. Bisant and J. R. Lyle: "A two-person inspection method to improve programming productivity," *IEEE Transactions on Software Engineering,* Vol.15, No.10, pp.1294-1304(1989).

[Boehm 1981] B. W. Boehm: *Software Engineering Economics*, Prentice-Hall (1981).

[Cameron 1989] J. Cameron: *JSP & JSD: The Jackson Approach to Software Development,* IEEE Computer Society, (1989).

[Collofello & Woodfield 1989] J. S. Collofello and S. N. Woodfield: "Evaluating the effectiveness of reliability-assurance techniques," *Journal of Systems & Software,* Vol.9, No.3, pp.191-195 (1989).

[Conte et al. 1986] S. D. Conte, H. E. Dunsmore and V. Y. Shen: *Software Engineering Metrics and Models*, The Benjamin/Cummings Pub., (1986).

[Coward 1982] J. M. Coward: "Today's risks in software development - Can they be significantly reduced?," *The Journal of Defense Systems Acquisition Management,* Vol.5, No.4, pp.73-94(1982).

[DeMarco 1982] T. DeMarco: *Controlling Projects: Management, and Estimation*, Yourdon Press, (1982).

[Druffel 1983] L. E. Druffel, S. T. Redwine, Jr. and W. E. Riddle: "The STARS program: Overview and rationale," *IEEE Computer*, Vol.16, No.11, pp.21-29(1983).

[Fagan 1976] M. E. Fagan: "Design and code inspections to reduce errors in program development," *IBM Systems Journal,* Vol.15, No.3, pp.182-211 (1976).

[Frakes et al. 1991] W. B. Frakes, C. J. Fox and B. A. Nejmeh: *Software Engineering in the UNIX/C Environment*, Prentice-Hall(1991).

[Humphrey 1988] Humphrey W.S.: "Characterizing the software process: A maturity framework," *IEEE Software*, Vol.5, No.2, pp.73-79(1988).

[IEEE 610 1990] "IEEE Standard Glossary of Software Engineering Terminology", IEEE, ANSI/IEEE Std 610.12-1990 (1990).

[IEEE 1008 1987] "IEEE Standard for Unit Testing", IEEE. Rep. IEEE-Std-1008-1987, (1987).

[IEEE 1028 1988] "IEEE Standard for Software Reviews and Audits," IEEE, ANSI/IEEE Std 1028-1988 (1988).

[Kusumoto et al. 1990] S. Kusumoto, K. Matsumoto, T. Kikuno and K. Torii: "Experimental evaluation of metrics for review activities," Proceedings of *10th Software Symposium,* pp.236-241 (1990).

[Kusumoto et al. 1991] S. Kusumoto, K. Matsumoto, T. Kikuno and K. Torii : "On a measurement environment for controlling software development activities," *IEICE Transactions on Communications Electronics Information and Systems,* Vol.E 74, No.5, pp.1051-1054(1991).

[Kusumoto et al. 1991] S. Kusumoto, K. Matsumoto, T. Kikuno and K. Torii: "Experimental evaluation of the cost effectiveness of software reviews, " Proceedings of *15th Computer Software & Applications Conference,* pp.424-429(1991).

[Kusumoto et al. 1992] S. Kusumoto, K. Matsumoto, T. Kikuno and K. Torii: "Approaches to improving effectiveness of review activities in technical review process," Proceedings of *International Software Quality Exchange,* pp. 7B1-7B16(1992).

[Kusumoto et al. 1992] S. Kusumoto, K. Matsumoto, T. Kikuno and K. Torii : "A new metric for cost effectiveness of software reviews," *IEICE Transactions on Information and Systems*, Vol. E75-D, No. 5, pp.674-680(1992).

[Kusumoto et al. 1993] S. Kusumoto, K. Matsumoto, T. Kikuno and K. Torii: "Using a Petri-net model for quantitative analysis of debugging processes in academic environment," *IEICE Transactions on Information and Systems,*(1993).(to appear)

[MaCabe 1976] T. J. MaCabe: "A complexity measure," *IEEE Transactions on Software Engineering,* Vol.SE-2, No.4, pp.308-320(1976).

[Matsumoto et al. 1988] K. Matsumoto, K. Inoue, T. Kikuno and K. Torii: "Experimental evaluation of software reliability growth models," Proceedings of *the 18th International Symposium on Fault-Tolerant Computing,* pp.148-153 (1988).

[Matsumoto et al. 1992] K. Matsumoto, S. Kusumoto, T. Kikuno and K. Torii : "An experimental evaluation of team performance in program development based on model —Extension of programmer performance model," *Journal of Information Processing*, Vol.15, No.3, pp.466-473(1992).

[Matsumoto et al. 1993] K. Matsumoto, S. Kusumoto, T. Kikuno and K. Torii: "A new framework of measuring software development processes," Proceedings of *IEEE-CS International Software Metrics Symposium,* pp.108-118(May 1993).

[Matsumoto 1990] K. Matsumoto: *A Programmer Performance Model and its Measurement Environment,* Ph.D. dissertation, Faculty of the Engineering Science, Osaka University, Japan,(1990).

[Mohri & Kikuno 1991] Y. Mohri and T. Kikuno: "Fault analysis based on fault reporting in JSP software development," Proceedings of *15th Computer Software & Applications Conference,* pp.591-596 (1991).

[Musa et al. 1987] J.D. Musa, A. Iannino and K. Okumoto: *Software Reliability: Measurement, Prediction, Application,* McGraw-Hill(1987).

[Myers 1978] G. J. Myers: "A controlled experiment in program testing and code walkthroughs / inspections," *Communications of the ACM,* Vol.21, No.9, pp760-768 (1978).

[Myers 1979] G. J. Myers: *The Art of Software Testing,* John Wiley & Sons, Inc. (1979).

[Peterson 1981] J. L. Peterson: *Petri Net Theory and the Modeling of Systems,* Prentice-Hall(1981).

[Royce 1970] W. W. Royce: "Managing the development of large software systems: Concepts and techniques," Proceedings of *WESCON*, pp.1-6(1970).

[Rook 1986] P. Rook: "Controlling software projects," *Software Engineering Journal*, pp.7-16(1986).

[Sackman et al. 1968] H. Sackman, W.J. Erikson and E.E. Grant: " Exploratory experimental studies comparing online and offline programming performance," *Communication of ACM,* Vol.11, No.1, pp.3-11(1968).

[Sommerville 1992] I. Sommerville: *Software Engineering,* Addison-Wesley, (1992).

[Tichy et al. 1993] W. F. Tichy, N. Harbermann and L. Prechelt:"Future directions in software engineering," *ACM SIGSOFT, Software Engineering Notes*, Vol.18, No.1, pp.35-48(1993).

[Torii et al. 1990] K. Torii, T Kikuno, K. Matsumoto and S. Kusumoto, "A measurement environment and some results at class experiments," Proceedings of *the 2nd International Workshop on Software Quality Improvement*, pp.88-91(1990).

[UNIX 1986] "UNIX User's Reference Manual –4.3 Berkeley Software Distribution Virtual VAX-11 Version–," (1986).

[Weinberg & Freedman 1984] G. M. Weinberg and D. P. Freedman: "Reviews, Walkthroughs, and Inspections," *IEEE Transactions on Software Engineering,* Vol.10, No.1, pp.68-72(1984).

[Yourdon & Constantine] E. Yourdon and L. L. Constantine: *Structured Design: Fundamentals of a Discipline of Computer Program and System Design*, Prentice-Hall, (1979).