A Programmer Performance Model and its Measurement Environment

Ken-ichi MATSUMOTO

August 1990

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

This thesis proposes a programmer performance model in order to evaluate the activities of a programmer in a quantitative and objective way. This model is then extended to evaluate the activities of the programmers of a team. Analysis of experimental evaluations shows the validity and the effectiveness of the proposed model. In addition, this thesis describes a measurement environment called GINGER which automatically collects and analyzes the data from the activities of programmers during software development and shows the obtained and analyzed data to the programmers as feedback information. By providing these features, GINGER aims to control the software development project in a meaningful and objective way.

The programmer performance model and the team performance model are defined based on a novel concept of error life span. The life span of an error is defined as the time duration from when the error manifests itself in the software to when the error is removed from the software. Results of experimental evaluations show that the programmer performance model has a high correlation with the "aptitude" of a student programmer. Additionally, the team performance model, which is defined by regarding a team as a virtual programmer, turns out to have a high correlation with the time a team spends debugging.

The proposed GINGER environment evaluates the activities of programmers by concentrating its attention on programmer productivity.

Additionally, the concept of program modification is introduced as a metric to estimate the activities of programmers and based on this metric, GINGER tries to analyze the activities of programmers in detail and to improve programmer productivity by using the analysis. A prototype system of GINGER is currently being developed and the validity and usefulness of the prototype system are shown by experimental evaluation in an academic environment.

In Chapter 1, related progress and topics in software engineering are briefly summarized for background and the outline of the thesis is described.

Chapter 2 describes the software development process, product, and software metrics. The software metrics include as objects of evaluation the software development process as well as the software product.

Chapter 3 introduces a new concept of error life span and proposes a programmer performance model based on the concept. Then, the programmer performance model is extended to a team performance model in order to evaluate the activities of programmers on a team. The team model makes it possible to devise an optimal team organization strategy based on the model.

Chapter 4 describes the experimental evaluation of the proposed models in both academic and industrial environments. The results of the experimental evaluations prove that the models are valid and effective in evaluating the activities of software development. Furthermore, a method is presented to automatically collect the estimated values of the error life spans based on the textual changes among successive versions of the program text made during the coding and debugging processes.

Chapter 5 describes the major functions of project management during software development. The chapter stresses that "controlling" is the most important of these functions.

Chapter 6 presents the system organization and functions of GINGER. GINGER supports collecting and analyzing data during software development. It supports information feedback to improve programmer productivity with respect to measurement-based control of the software development project. The first prototype system of GINGER is described.

Chapter 7 shows some experimental results of the prototype system. Results of experiments show that the prototype system provides the primitive functions needed to measure and control the software development process and product as well as to evaluate programmer productivity.

Chapter 8 presents a summary of the ideas discussed in the thesis and draws some conclusions. Finally, it summarizes future research work and describes key points for designing future measurement environments.

List of Major Publications

[1] K. Matsumoto, K. Inoue, H. Kudo, Y. Sugiyama and K. Torii, "Error life span and programmer performance," *Proceedings of the 11th International Computer Software and Applications Conference*, pp.259-265, Oct. 1987.

[2] K. Matsumoto, K. Inoue, T. Kikuno and K. Torii, "Experimental comparisons of software reliability growth models in academic environment," *Proceedings of Software Symposium '88*, pp.161-170, June 1988 (in Japanese).

[3] K. Matsumoto, K. Inoue, T. Kikuno and K. Torii, "Experimental comparison of software reliability growth models," *Proceedings of the 18th International Symposium on Fault-Tolerant Computing*, pp.148-153, June 1988.

[4] K. Matsumoto, K. Inoue, T. Kikuno and K. Torii, "An experimental evaluation of programmer performance based on error life span —For program development in academic environment—," *Transactions of IEICE Japan*, Vol.J71-D, No.10, pp.1959-1965, Oct. 1988 (in Japanese).

[5] K. Matsumoto, T. Kikuno and K. Torii, "An experimental evaluation of S-shaped software reliability growth models in academic environment —Comparison between models and determination of inflection rate—," *Transactions of IEICE Japan*, Vol.J73-D-I, No.2, pp.175-182, Feb. 1990 (in Japanese).

[6] S. Kusumoto, K. Matsumoto, T. Kikuno and K. Torii, "Experimental evaluation of metrics for review activities," *Proceedings of the 10th Software Symposium*, pp.236-241, June 1990.

[7] 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—," (Submitted to *Transactions of IPS of Japan*) (in Japanese).

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 Koji Torii for his continuous support, encouragement and guidance for this work.

I am also very grateful to the members of my thesis review committee: Professor Tadao Kasami, Professor Nobuki Tokura, Professor Ken'ichi Taniguchi, and Professor Hideo Miyahara for their invaluable comments and helpful criticism of this thesis.

I also wish to thank Associate Professor Tohru Kikuno and Assistant Professor Katsuro Inoue for their valuable suggestions and stimulating discussions.

Many of the 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 Jun'ichi Toyoda, Tadahiro Kitahashi, Mamoru Fujii, and Toshinobu Kashiwabara.

I would like to express my thanks to Dr. Lloyd G. Williams of Software Engineering Research for his insightful comments and valuable discussions on the paper which formed the basis for Chapter 3 of this thesis. I would also like to express my thanks to Mr. David F. Redmiles and Miss Kumiyo Nakakouji of 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 valuable comments of Mr. Kouichi Kishida of Software Research Associates, Inc. and Mr. Mitsuru Ohba of IBM Japan, Ltd. on the work presented in this thesis. I also acknowledge the support of Mr. Takahiro Jiro of Nihon Unisys, Ltd. for the industrial experimental project described in Chapter 4.

Finally, special thanks go to Mr. Satoshi Onishi of Sharp Corporation, Mr. Shinji Kusumoto, and Miss Satomi Nishida for their help and cooperation. I would also like to thank Mr. Satoshi Onishi and Mr. Shinji Kusumoto for their assistance in developing the GINGER system and in performing some of the academic experiments described in Chapters 4 and 7.

Contents

Chapter 1: Introduction

1.1 Progress in software engineering

Large software systems often provide incomplete functionality for what customers want, take too long to construct, cost too much time, use too much memory space or other resources to run, and rarely evolve to meet the changes needed [Lamb 1988]. These problems associated with development of software, especially large-scale software, have emphasized the need for a more disciplined and systematic approach. In the late 1960's, the term "software engineering" was coined as a rubric for a variety of techniques and tools to allow the production of cost-effective, reliable software within specified time constraints [Conte et al. 1986].

In the IEEE standard [IEEE 1983], software engineering is defined as the systematic approach to the development, operation, maintenance, and retirement of software. Boehm, on the other hand, defines software engineering as the application of science and mathematics by which the capabilities of a computer equipment are made useful to man via computer programs, procedures, and associated documentation [Boehm 1981].

In the intervening years, the practitioners and researchers have developed many techniques for addressing these problems mentioned above. These techniques are mainly for

- coping with the complexities of large systems,
- managing cooperating groups of programmers, and

- measuring the quality of a software system [Lamb 1988].

In addition, a number of notable concepts, which are still useful as the foundation for developing and maintaining the current software, have been established. These concepts include:

- software development and maintenance methods and models,
- assessment methods,
- software project management, and
- software development environments.

The DoD's STARS (Software Technology for Adaptable, Reliable System) program is one of the typical trials to apply these concepts to a practical software development [Druffel et al. 1983]. Actually, the STARS program intends to improve productivity while achieving greater system reliability and adaptability by using software engineering techniques in all phases of the software life cycle. The driving need is to have the capabilities of producing more powerful, reliable, and adaptable systems through software development, and in-service support processes that are more responsive, predictable, and cost-effective.

The major technological aspects within the STARS program are summarized as the following four areas [Conte et al. 1986]:

- Measurement and Project Management Tasks Area,
- Human Resources and Human Engineering Tasks Area,
- Application-Specific Task Area, and
- Support Systems Task Area.

1.2 Measurement in software development

The Measurement Task Area is considered to be the most important area within the STARS program since the ability to measure objectively is a foundation for all scientific and engineering disciplines [Dunham & Krusei 1983]. In other words, software engineering can attain the status of a scientific discipline only if it is built upon a solid foundation of objective measurement. Thus, the maturity of software engineering as a discipline may be reflected in the degree to which the use of metrics becomes normal and natural in the software development and maintenance process [Conte et al. 1986].

In general, activities for the measurement task area concern the development of evaluation criteria and their associated measures and metrics, and the experimental evaluation of techniques, methods, and tools. The strategy for progress needs, among other things, to establish success criteria for other task areas, and execute cost/benefit analysis of various opportunities. It also needs to collect baseline data against which to measure progress, instrument automated supports environments, and develop techniques for experimentally testing hypotheses related to software development and in-service support [Druffel et al. 1983].

Practical benefits of measurement consist of the following capabilities, (1) through (5) [Conte et al. 1986] [Dunham & Krusei 1983]:

(1) *Describing the current state of the world* — the ability to describe quantitatively the current state of software parameters, such as software quality, resources expended, and productivity.

(2) *Monitoring progress and providing feedback* — the ability to monitor progress, to anticipate problems, and to provide feedback to software personnel about potential problems.

(3) *Predicting project parameters such as cost, delivery time, functions, quality etc.* — the ability to predict software parameters, such as system cost, delivery time, and reliability.

(4) *Expressing requirement and goals quantitatively* — the ability to express requirements quantitatively both as goals and as acceptance criteria.

(5) *Analyzing costs and benefits* — the ability to quantify trade-offs that can be used by management in allocating resources.

This thesis focuses on the first two, and proposes a concrete method to implement them.

1.3 Understanding software development activities

The most fundamental function of measurement is to describe the current state of development. For complex software, this is extremely important because it allows us to discern trends and pattern [Druffel et al. 1983].

The measurement of resource expenditures is a good example of the benefit of this type of description. The resources expended on a project, particularly in terms of a human effort, are translated directly into costs. By collecting and analyzing information about exactly where these resources are being expended (for example, what phase of the life cycle, what types of activities, what parts of the system), one can identify the major cost *drivers* within a software organization. Then, one can answer questions such as "What types of activities consume large portions of the available manpower?" and "Where is the effort being wasted?" Therefore it can lead to the search for software tools or development methods designed to reduce the cost drivers [Druffel et al. 1983].

This thesis concerns the activities of programmers that contribute to improved software productivity and quality during software development, and proposes a programmer performance model to understand and evaluate these activities. To this end, the concept of error life span has been introduced as one metric to measure the negative effect of errors on software development [Matsumoto et al. 1988c] [Matsumoto et al. 1987].

In addition, we discuss the relation between programmer performance and team performance [Scott & Simmons 1975] and devise a strategy to organize reliable teams of programmers so that the activities of each team (thus, the activities of programmers) may increase. Three models, *M1*, *M2*, and *M3*, are presented to define the performance of a team based on the programer performance model. *M1* summarizes the performance of programmers. *M2* takes an average of the performance of programmers. *M3* evaluates the sum of error life spans under the assumption that the team is regarded to be a virtual programmer. These models are evaluated and compared by applying them to an experimental software development project.

1.4 Controlling software development activities

In the classic management model [Mackenzie 1969] [Thayer 1988], management is partitioned into five distinct functions or components: planning, organizing, staffing, directing, and controlling. These functions can be classified into two types. The first type includes planning, organizing and staffing, which are executed before constructing the activities of the software project, in order to accomplish the objectives of the project effectively. In contrast, the second type includes directing and controlling, which are executed dynamically during the software construction phase of the project. These latter are done to carry out the project if deviations from this prescribed plan occur. Therefore, the second type of functions, directing and controlling, are more important than the first type of functions. Since it is impossible to forecast all phenomena in the project when it starts, we have to develop a mechanism to correct deviations, and ensure the execution of the project in pursuance of the prescribed plan.

From a manager's perspective, monitoring progress, foreseeing problems before they get out of control, and taking appropriate corrective actions are very important to a project's successful completion. In other words, for controlling a project, the manager has to know the actual state of the project, clarify the difference between the prescribed plan and the actual state of the project, and help the developers to accomplish the prescribed plan.

Thus, measurement is one of the most powerful and effective technologies for controlling software development activities in a quantitative and objective way. Furthermore, measurement adds visibility to the software project; tracking can be carried out in a meaningful and objective way [Druffel et al. 1983]. DeMarco succinctly makes this point in stating, "You can't control what you can't measure." [DeMarco 1982]

In this thesis, we concentrate on programmer productivity as a metric to control the software project. Then, we propose a system that automatically collects and analyzes the data from the activities of programmers during software development and shows the obtained and analyzed data to the programmers as feedback information [Onishi et al. 1986]. This feedback allows the programmers to recognize their weaknesses and improve their activities [Basili & Rombach 1987]. We expect that the overall productivity during development and the quality of the resulting products can be controlled by using this system.

Other systems and environments for improving both programmer productivity and the quality of products have been developed. One of them is the TAME (Tailoring A Measurement Environment) project at the University of Maryland [Basili & Rombach 1988] which provides a software engineering process model. This software engineering process model is based upon various kinds of improvement and goal/question/metric paradigms. The system will ultimately run on a distributed system consisting of at least one mainframe computer and a number of workstations. The mainframes are needed to host the experience base, which can be assumed to be very large. Thus, we can say that TAME will be a large-scale system.

The proposed measurement environment, described in this thesis, is also improvement-oriented and suitable for distributed environments. The prototype system of the measurement environment uses the existing functions of UNIX* in order to collect data from many workstations in software development. Therefore, it is easy to apply the prototype system to various kinds of projects if their software is developed on UNIX workstations.

1.5 Outlines of the thesis

First, this thesis proposes a programmer performance model and a team performance model based on the concept of error life span in order to understand and evaluate the activities of the programmer and the programmers of a team in a quantitative and objective way. Then, it describes a measurement environment called GINGER which automatically collects and analyzes the data from the activities of programmers during software development and shows the obtained and analyzed data to the programmers as feedback information in order to control (and manage) the software development project in a meaningful and objective way.

Chapters 2, 3, 4 outline a programmer performance model and a team performance model. Chapter 2 describes the software development process and product, and software metrics which evaluate the software development process and product in a quantitative and objective way. Chapter 3 introduces a new concept of error life span and proposes a programmer performance model based on this concept. Then, the programmer performance model is extended to a team performance model in order to evaluate the activities of programmers of a team and devise an optimal team organization strategy based on the model.

* UNIX is a registered trademark of AT & T Bell Laboratories.

Chapter 4 describes the experimental evaluation of the proposed models in both academic and industrial environments. The results of the experimental evaluation show the validity and the effectiveness of the model. Furthermore, a method is introduced for automatically collecting the estimated values of the error life spans based on the textual changes among successive versions of the program text during coding and debugging.

Chapters 5, 6, 7 outline a measurement environment, GINGER. Chapter 5 describes the major functions in software development project management and shows that controlling is the most important function among them. Chapter 6 presents the system organization and functions of GINGER that support collecting and analyzing data from a software development process and support information feedback to improve programmer productivity. Then, the first prototype of GINGER is described. Chapter 7 shows some experimental results using the prototype system.

Chapter 8 presents a summary of the ideas discussed in the thesis, draws some conclusions, and summarizes some areas for future research.

Chapter 2: Software Development

2.1 Product and process

One of the purposes of software engineering is to improve software productivity and quality. Various kinds of studies in this field have been undertaken over the years. According to the results of these studies, the scope of the targets to be analyzed and discussed can be classified into two groups: (1) software products and (2) software development processes.

In the IEEE standard [IEEE 1983], a software product is defined as a software entity (computer programs, procedures rules, and possibly associated documentation and data pertaining to the operation of a computer system) designated for delivery to a user. A software development process is defined to be the process by which user needs are translated into software requirements, software requirements transformed into design, the design implemented in code, and the code tested, documented, and certified for operational use.

When we want to understand and control a software product and a software development process in a specific way, the concept of a software life cycle is useful. The software life cycle is defined as the period of time that starts when a software product is conceived and that ends when the product is no longer available for use [IEEE 1983]. The software life cycle typically includes a requirements phase, a design phase, an implementation phase, a test phase, an installation and checkout phase, an operation and maintenance phase, and sometimes, a retirement phase. Each of the phases is defined as follows [IEEE 1983]:

(1) *Requirements phase*: the period of time during which the requirements for a software product, such as the functional and performance capabilities, are defined and documented.

(2) *Design phase*: the period of time during which the designs for the architecture, software components, interface, and data are created, documented, and verified to satisfy requirements.

(3) *Implementation phase*: the period of time during which a software product is created from design documentation and debugged.

(4) *Test phase*: the period of time during which the components of a software product are evaluated and integrated and the software product is evaluated to determine whether or not the requirements have been satisfied.

(5) *Installation and checkout phase*: the period of time during which a software product is integrated into its operational environment and tested in this environment to ensure that it performs as required.

(6) *Operation and maintenance phase*: the period of time during which a software product is employed in its operational environment, monitored for satisfactory performance, and modified as necessary to correct problems or to respond to changing requirements.

(7) *Retirement phase*: the period of time during which support for a software product is terminated.

2.2 Software metrics

Measuring the software product and the development process throughout the software life cycle, described in subsection 2.1, is essential for improving software productivity and quality. Software metrics are often classified into product metrics and process metrics, and are applied respectively to either the software product or the development process [Conte et al. 1986].

Product metrics are measures of the software product [Conte et al. 1986]. Product metrics include the size of the product (such as the number of lines of code or some count of tokens in the program), the logic structure complexity (such as flow of control, depth of nesting, or recursion), the data structure complexity (such as the number of variables used), the function (such as type of software: business, scientific, systems, and so on), and combinations of these [Conte et al. 1986].

Among these product metrics, the complexity metric for a program is the most well-known product metric. It is used for the implementation phase and the test phase. The complexity metric is often a good indicator of whether a product is well-designed, understandable, and easy to modify [Basili 1980]. Unfortunately, the complexity metric and most product metrics may reveal nothing about how the software product has evolved into its current state [Conte et al. 1986].

Process metrics quantify attributes of the development process (including product evolution) and of the development environment [Conte et al. 1986]. Process metrics can evaluate such various items as development techniques (the use of top-down or bottom-up development techniques, structured programming, and other software engineering techniques), developmental aids (the use of design languages and systems, editors, interactive systems, and version control systems), supervisory techniques (such as the type of team organization and number of communication paths), and resources (human, computer, time schedule, and so on) [Conte et al. 1986].

The software reliability growth model is a well-known process metric (model) used for both the test phase and the operation and maintenance phase [Matsumoto et al. 1990] [Musa et al. 1987]. The software reliability growth model can estimate the mean time to failure (MTTF), the number of residual faults in product and so on.

Most process metrics require greater efforts to collect and analyze data than do product metrics. It is especially difficult to collect reliable data on human activities. However, process metrics can provide more valuable information for improving software productivity and quality than can the product metrics.

2.3 Programmer activities

Among the attributes of the development process, mentioned in subsection 2.2, the human resource is one of the most important to be evaluated. The reason why we emphasize the human resource is that human activities are strongly related to productivity and to the quality of software [Curtis 1985]. Moreover, there are very large individual differences in human activities with respect to productivity and quality of software. For example, Sackman, Erikson and Grant [Sackman et al. 1968] showed that for most performance variables, there are very large individual differences in the programming performance.

Also in COCOMO (COnstructive COst MOdel)[Boehm 1981], the programmer capability is one of the major attributes, having a range of 2.03 for software productivity. The programmer capability is the highest cost driver among 14 cost drivers in COCOMO.

However, since there are no model-based approaches to evaluate programmer activities, it has been believed that programmer activities cannot be measured absolutely. Thus, very simple but insufficient measures have been widely used in the practical applications. For example, the number of years that a team has been using a programming language, the number of years that a programmer has been with the organization, the number of years that a programmer has been associated with a programming team, and the number of years of experience constructing similar software or managers' intuitive evaluations have been used.

2.4 Team activities

Generally, large software systems are developed by teams that consists of many analysts, designers, programmers and so on. Several ideas have already been proposed to organize a software development team efficiently [Myers 1976].

The chief programmer team concept was originated by Mills [Baker 1972]. The team is headed by a chief programmer (that is, a senior-level programmer who is highly skilled and experienced). The chief programmer performs all of the design tasks, writes the code for all critical modules, and performs the integration and testing of the team's code. He or she is also the primary interface to outside organizations such as other teams and the user organization and thus reduces the number of lines of communication among project members. The chief programmer is assisted by a back-up programmer and a programming librarian [Myers 1976]. The remainder of the team varies on the particular stage of the project [Myers 1976].

The specialist team has been proposed by Brooks [Brooks 1975]. The major differences between this team and the chief programmer team are that the team members remain within the same team for the entire project and each member of the team has a special assignment that takes advantage of his or her particular talents [Myers 1976].

Another proposal for programming teams is the democratic team [Weinberg 1971]. This team is different from other teams in that it has no formally appointed leader or initial individual assignments. A particular team member may become an informal leader when the team enters a stage for which that team member is most qualified. The team makes its own work assignments based on the talents of the members. One big difference between this team and the chief programmer team is that the democratic team stays together from project to project. When a project is completed, the team is not broken up but is assigned as a whole to a new project. This means that the rapport, working relationships, and group standards within the team are maintained from project to project.

These organizations are valuable because they recognize that programming is largely a social activity rather than an individual activity [Myers 1976]. However, programming teams do have a few disadvantages. The work of each individual programmer is less visible to the project manager, making performance evaluations more difficult. In the teams with a formal leader, the leader can become the sole interface between the manager and other team members, leading to significant morale problems. Therefore, a quantitative and objective method needs to be devised to evaluate the performance of team member and team.

Chapter 3: Programmer Performance Model

3.1 Error and fault

It is widely recognized that errors have a close relation to software productivity and quality. Numerous studies have been conducted in the field of so called "error analysis" in order to clarify the effect of each error on software productivity and quality. For example, Weiss [Weiss 1979] used errors as a way of evaluating the software development process. He investigated causes of errors and efforts involved in fixing errors and proposed methods of error detection and correction.

According to Basili and Rombach [Basili & Rombach 1987], an "error" is a defect in the human thought process made while trying to understand

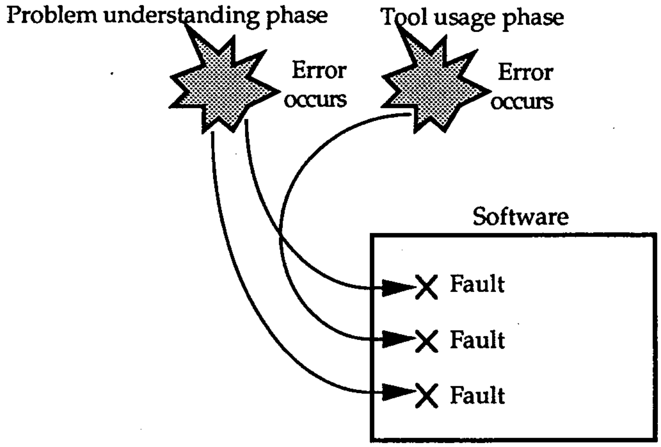


Figure 3.1 Error and fault

information for solving a problem or while trying to use methods and tools. A "fault" is the concrete manifestation of an error within the software. Figure 3.1 illustrates these definitions.

In the IEEE standard [IEEE 1983], 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 a software specification and incorrect translation or omission of a requirement in the design specification. And, a fault is defined as a manifestation of an error in software. A fault, if encountered, may cause a failure (synonymous with bug).

Myers has said that an error is a mistake in translating information [Myers 1976]. Software production, then, is simply a number of translation processes, translating the initial problem into various intermediate solutions until a detailed set of computer instructions is produced. Software errors (faults) are introduced whenever one fails to completely and accurately translate one representation or solution of the problem to another more detailed representation [Myers 1976].

Myers has also pointed out that a person has to perform the following four steps in order to translate information [Myers 1976].

Step 1: He (or she) receives the information using his read mechanism *R*.

Step 2: He stores this information in his memory *M*.

Step 3: He retrieves from his memory this information and other information describing the translation process, performs the translation, and sends the result to his writing mechanism *W*.

Step 4: The information is physically depressed by writing, typing on a terminal, or speaking.

Myers has summarized the errors that may be generated in each step [Myers 1976]:

Step 1: Errors are introduced by misreading the input information, seeing what is expected as opposed to what is actually there, making assumptions about missing facts, or simply overlooking information.

Step 2: Errors in this step result from misinterpreting or misunderstanding the input information. The reason may be that the information may be too complex, the person may not have the necessary education background, or the information may be ambiguous.

Step 3: The largest source of errors in this step is the phenomenon of forgetting the input information or how to perform the translation properly. Weaknesses in other mental abilities, such as clarity of thought and retrieval of related knowledge, also contributes errors.

Step 4: Many people do not write or express themselves clearly and that obscures their output. If there is a large amount of output information, the person takes shortcuts or assumes that facts will be "intuitively obvious" to his audience.

The emphasis on error and fault in these past studies shows its prominence in, and therefore the necessity of studying it in regards to, the activities of programmers. Unfortunately, it is practically impossible to count errors. We can however count faults which are manifestations of errors in software, to evaluate the activities of programmer. "Error" in this thesis is almost the same as "fault" in this subsection. However multiple faults which are clearly caused by one defect are treated as one error.

3.2 Error life span

In order to measure the negative effect of errors on the development processes, one may want simply to count the total number of errors involved in the program texts throughout the processes. We believe, however, that counting the total number of errors is insufficient since each error has different effects on the software productivity and quality. Thus, we introduce a weight into the error, which could represent a particular rate of effect of each error. The weight to be introduced in this thesis is called a life span (time duration) of the error.

An error life span T_e for an error e has been defined as time duration from when the error e manifests in the software to when the error e is removed from the software [Matsumoto et al. 1988c] [Matsumoto et al. 1987]. Figure 3.2 shows an example of error life span T_e. In Figure 3.2, × and ○ represent respectively the times of error manifestation and error removal.

For example, we consider a case in which an error causes some faults in a program text. If the life span of an error is long, that is, the faults remain for a long period of time in the program text, then the programmer would have a hard time removing them. One cause of this difficulties is that the programmer would forget the details of the old code relating to the faults. Also the erroneous codes affect other codes appended to the program text afterward. Hence, we naturally think that an error with a long life span has a high (negative) effect to the project progress and the program reliability.

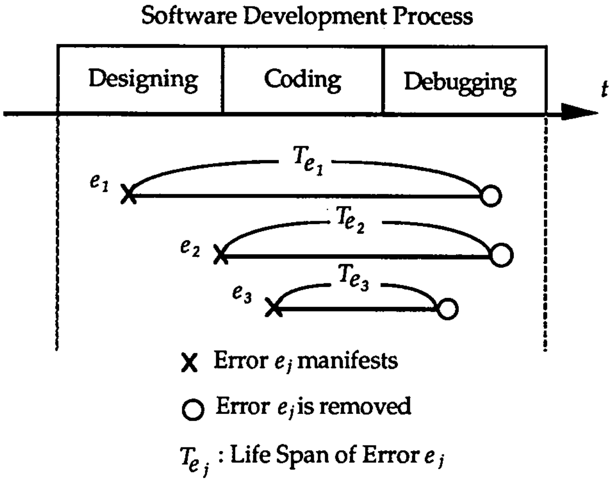


Figure 3.2 Error life span

Similar concepts to the error life span can be seen in several earlier papers [Mills 1976] [Weiss & Basili 1985]. Mills [Mills 1976] introduced the concept of "error days" for estimating the quality of an acceptable system. "Error days" is, for each error removed, defined to be the sum of the days from its creation to its detection. He has implied, though without empirical evidence, that this measure is an indication of probable future errors and of the effectiveness of the design and testing processes. Weiss and Basili [Weiss & Basili 1985] have used changes as a way of evaluating the software development processes. They mentioned that "the length of time each error remained in the system" would be useful information for evaluating the software development processes. However no collection and analysis of such data were made.

We have actually collected data from student projects and studied the relation between the value we obtained and the programmer performance. The time used for measuring the life span in our experiments is actual terminal access time rather than calendar days used for the error days.

3.3 Programmer performance model

We believe that the error life span could indicate some aspects of programmer performance, as well as the product quality as suggested by Mills [Mills 1976]. The error life span closely relates to the performance of the programmer in the following two ways:

(1) the number of errors made in the software development processes, and

(2) the rate of detection and removal of these errors.

The value called a "score", is defined to indicate some aspects of programmer performance. The score *SI* for each individual programmer is formally defined by formula (3.1).

$$SI = \left(\frac{\text{Sum of error life spans}}{f(p)}\right)^{-1} \qquad \text{... (3.1)}$$

where f : Normalizing function,

p : Complexity of given problem.

In this definition of *SI*, the following two assumptions are made.

(1) The specification (of a problem) is not modified during software development.

(2) Designing, coding, and debugging are completed by the same programmer.

This definition of *SI* comes from the fact that programmer, who makes less errors and removes these errors in shorter time, gets better performance.

For the normalizing function *f(p)*, the square of the final program size, i.e., L^2 is used in this thesis. The explanation of why we chose L^2 follows. ΣT_e is rewritten by

$$\Sigma T_e = avg \times N \qquad \text{... (3.2)}$$

where avg : The average of the error life spans,

N : The number of the total errors.

Since *avg* and *N* are considered to depend on the complexity *p* of the problem, both *avg* and *N* should be normalized by *p* in such a way that,

$$\frac{avg}{p} \times \frac{N}{p} \qquad \text{... (3.3)}$$

where p : Complexity of the problem.

In the experiments, to be described in Chapter 4, there is only a small difference among the specifications. Thus, we think that the complexity *p* of the problem is estimated by the final program size *L* (the number of the lines in the final program text). As the result, L^2 is employed as the normalizing function.

Thus, the score *SI* in the formula (3.1) for each individual programmer is rewritten by the formula (3.4).

$$SI = \left(\frac{\Sigma T_e}{L^2}\right)^{-1} \qquad \text{... (3.4)}$$

where T_e : An error life span.

L : The final program size.

3.4 Team performance model

Let us consider a case that software is developed by a team, which consists of n programmers. Based on the programmer performance model mentioned in Section 3.3, three models $M1, M2$ and $M3$ are defined to measure the performance of each team. In the following, let SI_j denote the score of a programmer j ($1 \leq j \leq n$). For the sake of simplicity, the following notation

$$SI_j = \left(\frac{E_j}{L_j^2}\right)^{-1} \qquad \text{... (3.5)}$$

where $E_j = \Sigma T_e,$

is used instead of the notation in (3.4).

The first two models $M1$ and $M2$ are defined by using the scores SI_j's (j = 1, 2, ..., n) of the programmers of the team.

Model $M1$

The score for a team (in short, called team score) $ST1$ is defined as follows;

$$ST1 = \sum_{j=1}^{n} SI_j \qquad \text{... (3.6)}$$

Model $M2$

The score $ST2$ is defined as follows;

$$ST2 = \frac{1}{n}\sum_{j=1}^{n} SI_j \qquad \text{... (3.7)}$$

The model $M3$ is defined by regarding a team as a virtual programmer.

Model $M3$

The team score $ST3$ is defined as follows;

$$ST3 = \left(\frac{\Sigma E_j}{(\Sigma L_j)^2}\right)^{-1} \qquad \text{... (3.8)}$$

Next, we discuss the strategies for team organization in order to maximize the team score. In the following discussion, we consider an application of models *M1*, *M2* and *M3* to a new project *P*. The following three assumptions are made.

(1) The scores SI_j ($j = 1, 2, ..., n$) are known beforehand (for example, for the past project *P'* similar to the project *P*).

(2) For each programmer, the value of the score doesn't depend on the project. (Thus the score for projects *P* and *P'* are the same.)

(3) For the project *P*, it is possible to estimate the final program size L ($= \Sigma L_j$) [Boehm 1981].

Let SI_j, L_j and E_j denote the values for a new project *P*, and SI_j', L_j' and E_j' denote the values for an old similar project *P'*.

Strategy for model *M1*

By assumption, the formula

$$ST1 = \sum_{j=1}^{n} SI_j = \sum_{j=1}^{n} SI_j' \qquad \text{... (3.9)}$$

is obtained. Thus, if there is no limit on the total number of programmers in the team, the best way to organize programmers is to collect as many as possible. However this organization seems impractical. But when there is a limit on the number, collecting only programmers with high score is the best way (which seems to be a practical conclusion).

Strategy for model *M2*

By assumptions, the formula

$$ST2 = \frac{1}{n}\sum_{j=1}^{n} SI_j = \frac{1}{n}\sum_{j=1}^{n} SI_j' \qquad \text{... (3.10)}$$

is obtained. Thus, the best approach is to collect only programmers with high scores and to keep the value of *n* as small as possible. An exceptional case is that a programmer with the highest score develops all modules of program.

It is clear that in both models *M1* and *M2*, collecting programmers is the only key factor. How to distribute *m* modules (to be developed) among *n* programmers in the team doesn't affect the optimality of the strategy.

Strategy for model *M3*

By the definition, if the relation

$$\frac{L_1}{SI_1'} = \frac{L_2}{SI_2'} = \dots = \frac{L_n}{SI_n'} \qquad \text{... (3.11)}$$

holds, then the value of *ST3* becomes maximum. At that time, the relation

$$ST3 = \sum_{j=1}^{n} SI_j' \qquad \text{... (3.12)}$$

is derived. Thus, the best way is that, in the new project *P*, each programmer j ($j = 1, 2, \dots, n$) develops program modules with size L_j, that is proportional SI_j'.

Chapter 4: Experimental Evaluation of Models

4.1 Overview

In order to evaluate the proposed models, three experimental software development projects (Experiments 1, 2 and 3) have been executed in academic and industrial environments. Experiments 1 and 2 have been done in an academic environment [Matsumoto et al. 1988c] [Matsumoto et al. 1987] [Matsumoto et al. 1986]. The purpose of Experiment 1 is to show the validity and the effectiveness of error life span T_e and score SI. In Experiment 1, project data on nine students were collected. Each student developed a compiler for a subset of PL/I, Pascal or C using Pascal or C. Final program sizes were about 1000~2500 lines.

The purpose of Experiment 2 is to compare the scores SI 's of the same students. In Experiment 2, project data on six students were collected. Each student developed a compiler for a subset of Pascal using C and a kind of inventory control program using Pascal. Final program sizes of compilers were about 1000 lines and those of inventory control programs were about 300 lines.

In both Experiments 1 and 2, to obtain error life span T_e, we traced and analyzed by hand all the files used in the projects. The time unit for T_e was terminal access time. In addition, we assumed the case that a programmer was given a specification of the program to develop and that the same programmer performed all the work, i.e., designing, coding and debugging. It was also assumed that the programmer completed the development within a specific time frame.

Experiment 3 has been done in an industrial environment to show the validity and the effectiveness of the team score *ST*. In Experiment 3, eight teams of programmers developed the same system, a file processing program in a business application, using COBOL. The system consisted of 18 program modules. The final program sizes ware about 2000 lines. In Experiment 3, to obtain error life span T_e, we used an automatic estimation method for the sum of error life span. The details of the automatic estimation method are described in Section 4.4.

4.2 Experiment 1

4.2.1 Experimental data

We collected project data on nine students. Each student developed a compiler for a subset of PL/I, PASCAL, or C using PASCAL or C. They had studied the theory of compiler construction in their classes; however, they had no previous experience in constructing compilers.

In this experiment, we did not measure actual error life spans, but we collected a closely related value T_e. T_e is the life span of the faults in the program text, caused by an error *e*. In other words, we started counting T_e when a fault caused by *e* was first embedded in the program text, and stopped counting when all of the faults caused by *e* were removed. Thus errors removed before coding were not counted here. This was because we had no appropriate method to count all the errors.

It might have been useful to investigate the details of errors which the programmer made. However, such an investigation would have been another task for the programmer, thus inhibiting the project's progress. Hence, we decided not to inquire directly as to the details of the programmer's errors, but to analyze the textual changes among successive versions of the program texts and to estimate the number of the errors from the faults found in the texts.

Textual changes reflect the coding and debugging processes and we can collect them automatically with less effort. For each textual change, we also collected the reason for the textual changes as annotated by the programmer himself using an on-line data collecting tool. From the textual changes and the reasons, we determined e and its life span T_e by hand. In this analysis, syntactic errors were not counted as e. We assumed that most errors which affected the coding and debugging processes were found as e by this analysis, and other errors which never gave faults on the texts could have had only a limited effect. In the following, we use the phrase, "error life span" in the sense of T_e.

We counted the successive time that each student accessed a terminal and used it as a time unit for error life span T_e. Although this did not precisely correspond to the actual time devoted to the project, we used this time unit for two reasons. One is that the terminal access time could be traced automatically. The second reason, which is more substantial, is that employing the terminal access time more sharply contrasted the difference of the computed values for each programmer, as compared with employing the actual time consumed. That is, the programmer who designed and debugged a large amount of his code on

Table 4.1 Data of Experiment 1

Student	Program size	Total of terminal access time (min.)	Total number of errors	Sum of error life spans (min.)	Average of error life spans (min.)	Score *SI*	Grade point average in C.S. courses
#1	2098	7955	77	93750	1218	46.9	69.6
#2	1685	6202	101	29715	294	95.5	71.5
#3	1530	5906	61	28620	469	81.8	74.0
#4	1789	8021	78	67020	859	47.8	65.8
#5	1094	4754	55	32145	584	37.2	72.9
#6	1661	3463	35	11550	330	238.9	82.8
#7	2111	5838	26	24045	923	185.3	71.6
#8	1084	8651	49	66765	1363	17.6	69.6
#9	2420	5139	33	49170	1490	119.1	73.4

the desk and who did not use the terminal extensively, had an advantage in the computed values.

Table 4.1 shows the program size (which is the number of lines in the program text when the program completes), the total of the terminal access time, the total number of errors, the sum of the error life spans, the average of the error life spans, the score *SI*, and the grade point average in computer science courses.

For example, student #2 made 101 errors and student #7 made only 26 errors. As for the sum of the error life spans, $\sum T_e$, however, student #1, who made 77 errors, had the highest value, 93750, and student #6, who made 35 errors, had the lowest value, 11550. It was not always true that a student who made the most errors had the highest value of $\sum T_e$, and a student who made the least errors had the lowest value of $\sum T_e$. This was because each student had a unique average value of his own error life spans, and there was not a high correlation between the average and the total number of errors.

Student #8 had the lowest score, 17.6, and student #6 had the highest score, 238.9. The order of students with respect to the scores was the same as the order with respect to our intuitive impression of student performance. Below, we discuss the reasons why some students received high scores and low scores.

High Scores

(1) Student #6 (238.9)

He spent a large amount of time in designing and coding on the desk (without using terminal) and only used the terminal for a very short period.

(2) Student #7 (185.3) and Student #9 (119.1)

They referred to text book that precisely describes how to construct a PASCAL compiler using PASCAL. Since they could obtain basic algorithms and data structures from that book, the number of errors was reduced and the development period was shortened.

Low Scores

(1) Student #8 (17.6)

He started coding using the terminal even though he did not have a good grasp of compiler theory at the beginning. Furthermore, he did not clearly understand the specification of the compiler at the beginning; thus, he had to modify the program text extensively.

(2) Student #5 (37.2)

He made many errors caused by poor understanding of the implementation language, C. He also incorrectly designed the parsing section of the compiler and he had trouble determining the errors when he detected the faults during the test phase.

4.2.2 Evaluation

(1) ΣT_e vs. terminal access time

In the definition of score *SI*, the life span T_e of the errors is introduced as a weight of the error. In order to prove the validity of this idea, ΣT_e (the sum of error life spans) is compared with the total terminal access time which seems to directly correspond to the programmer performance.

Figure 4.1 shows the scatter plots of ΣT_e versus the total terminal access time. A coefficient of correlation between them is 0.82. On the other hand, a coefficient of correlation between the total number of errors and the total terminal access time is 0.45.

Thus it can be said that the sum of error life spans would very closely relate to the performance of the programmer as compared with the total number of errors.

(2) Score *SI* vs. grade point average

In addition, we investigated the grade point average in computer science courses for each programmer and compared them with the scores. Figure 4.2 shows the scatter plots of the scores versus the grade point averages in computer science courses.

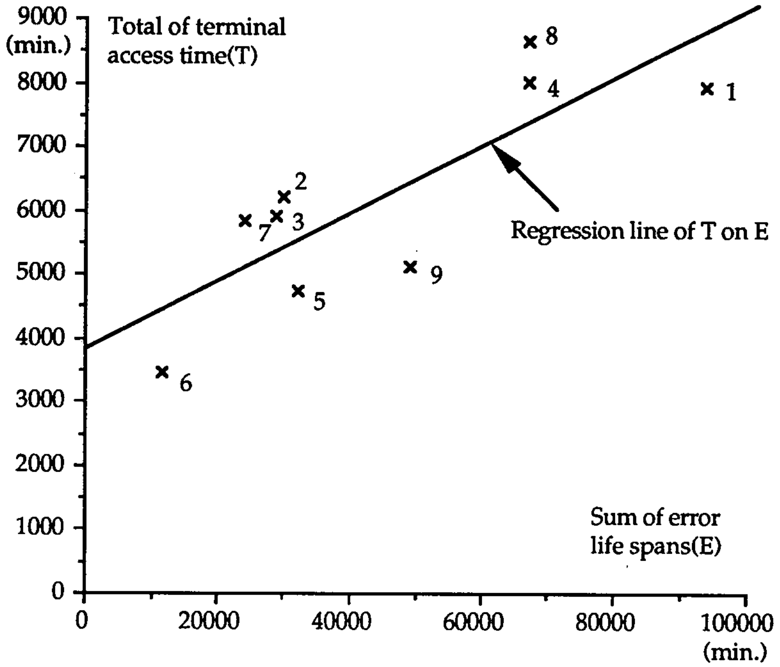


Figure 4.1 Sum of error life spans vs. total of terminal access time (Experiment 1)

We found that a coefficient of correlation between them was 0.75. Moher and Schneider [Moher & Schneider 1981] have found that "experience" (as measured by the number of computer science or programming courses) and "aptitude" (as measured by the grade point averages in computer science courses) are the major predictors of performance for student programmers. In our experiment, the experience of each student is almost the same. Hence, we believe that the difference of our obtained values can be explained simply by programmer performance. Of course, we do not think that the sum of the error life spans indicates the complete extent of the programmer performance; rather,

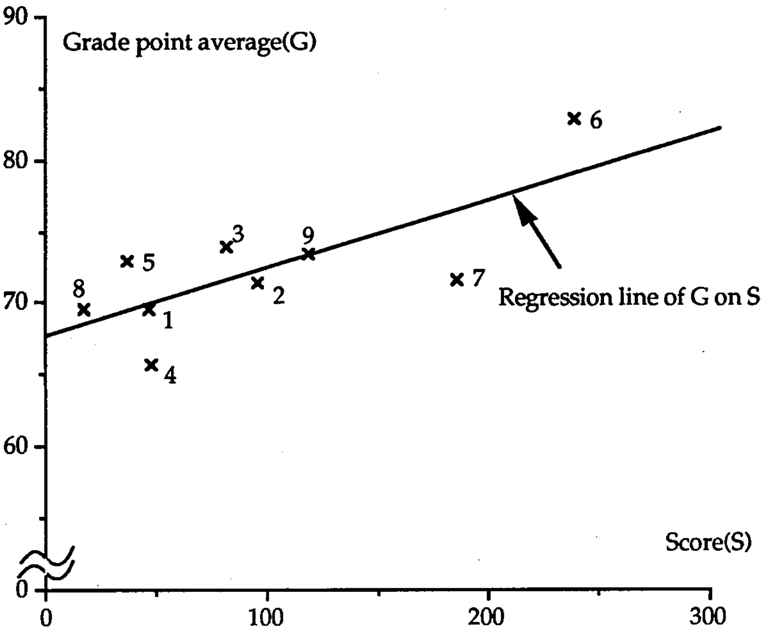


Figure 4.2 Score vs. grade point averages

we think it indicates one important aspect of programmer performance related to the productivity and quality of software.

4.3 Experiment 2

4.3.1 Experimental data

We collected data on two different student projects, *compiler construction* (Project 1) described in Experiment 1 and a so-called *liquor wholesale* problem (Project 2) to be described now. Each had different types of difficulties. For the liquor wholesale problem, students were given the program specification and taught roughly how to design the program. We analyzed the data for six students who participated in both projects. Methods of collecting and analyzing data were the same as for Experiment 1; that is, the errors were determined by hand and the sums of the error life spans were normalized by L^2 (the square of the final program size). In this experiment, we observed the difference between the scores of each student for different projects.

4.3.2 Evaluation

Table 4.2 shows the sizes of the programs, the sums of the error life spans, and the scores in both projects. Figure 4.3 shows the scatter plots of the scores in both projects. The order of students with respect to the scores are almost the same between both projects except for one student. This student, #13, had misunderstood the syntax of subset-PASCAL when he tried to develop the subset-PASCAL compiler in Project 1. During this project, he had to change numerous errors related to the syntax differences. Consequently, this worsened his score from Project 1.

The averages of the scores are 31.3 in Project 1 and 54.6 in Project 2. (If we preclude student #13, the average will be 34.1 in Project 1 and 46.8 in Project 2.) We think that the scores are stable enough and L^2 is an appropriate normalizing function for a measure of programmer performance in such small-scale projects, even though the numerical values we have computed do not by themselves show the absolute performance of the programmers (in the sense that if programmer *A* took 20 and *B* took 40, then *B* can program twice as well as *A*). We expect that if further data is collected, a suitable normalizing function would be found which would give absolute meaning to the computed values.

Table 4.2 Data of Experiment 2

Student	Project 1: Compiler Construction			Project 2: Liquor Wholesale Problem		
	Program size	Sum of error life spans (min.)	Score	Program size	Sum of error life spans (min.)	Score
#11	1251	65205	24.0	322	3300	31.4
#12	963	24690	37.6	326	2430	43.7
#13	1366	107745	17.3	326	1140	93.2
#14	1260	35790	44.4	340	1800	64.2
#15	998	25830	38.6	298	2055	43.2
#16	1149	50940	25.9	296	1695	51.7

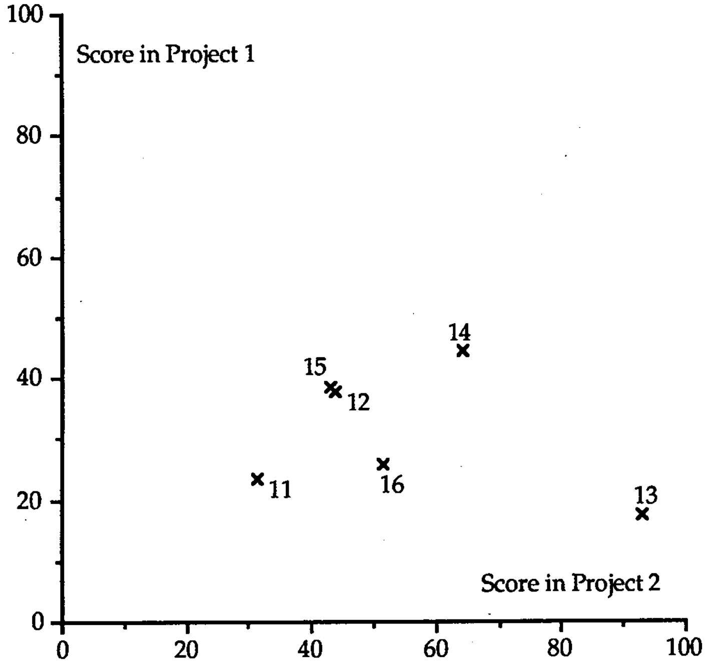


Figure 4.3 Score in two projects

4.4 Automatic estimation of error life span

In Experiments 1 and 2, obtaining the error life spans was very expensive. We had to trace and analyze all the files used in the development projects by hand and keep a large amount of data concerning the processes. Hence, obtaining the error life spans in various software development projects is in practice prohibitive. Furthermore, it follows that it is difficult to show the programmer the computed value of the error life span periodically during the development processes (to improve the activity of the programmer).

To find an equivalent value to the error life span, we investigated correlations among the sum of the error life spans, the average of the error life spans, and other collected data as shown in Table 4.3. If there were easily collectable data which had a high correlation with the sum of the error life spans or the average of the error life spans, we could compute the estimated value

Table 4.3 Coefficient of correlation among data on Experiment 1

	Sum of error life spans (min.)	Average of error life spans (min.)
Program size	0.15	0.29
Total of terminal access time(min.)	0.82	0.52
Number of total errors	0.35	-0.35

very easily. However, since we could not produce such data, we devised a way to obtain an estimated value of the error life spans automatically.

In order to do this, we had to determine the errors automatically in some manner; but, in general, we had no satisfactory mechanical way to recognize the errors. Dunsmore and Gannon [Dunsmore & Gannon 1980] demonstrated that program changes (i.e. textual changes between successive version of the program) were correlated with the total error occurrences in a program written by 33 programmers. Therefore, we simply estimate that in the program text, each line modified at each edit session corresponds to one error to be counted. If the created and deleted times of each line are known, the estimated error life spans can be collected easily.

Here we assume that software development consists of a sequence of edit sessions of the program text. For each line j in the program text, we define the life span l_{ij} of j at an edit session i as follows:

$$l_{ij} = \begin{cases} 0 & \text{if line } j \text{ is not modified at edit session } i. \\ t_i - t'_{ij} & \text{if line } j \text{ is modified at edit session } i. \end{cases} \quad \text{... (4.1)}$$

where $1 \leq j \leq max_i$, and

max_i: Number of lines when edit session i begins.

t_i : Time when edit session i terminates.

t'_{ij} : Time of the latest modification of line j before t_i.

As mentioned above, we estimate that each non-zero value l_{ij} corresponds to the life span of an error. Therefore, the estimated value L for the sum of the error life spans is given by

$$L = \sum_{i=1}^{M} \sum_{j=1}^{max_i} l_{ij} \qquad \text{... (4.2)}$$

where M is the aggregate of the edit sessions.

We computed L using the data of Experiment 1, and compared L with the sum $\sum T_e$ of the actual error life span. A coefficient of correlation between them is 0.86. Thus, we can conclude that the value L estimates the sum of the error life span quite well.

In the above definition of L, it seems that the number of errors is overestimated in general. It may be more realistic to state that, instead of each line, a set of lines which were created at an edit session and modified at another edit session corresponds to one error. If a programmer modifies several lines of text in an edit session, and if those modified lines were originally created in the same edit session, we consider that he or she fixed only one error. But if those modified lines were originally created in different edit sessions, we count the number of those edit sessions and treat that count as the number of errors fixed. Each edit session is distinguished from others by time stamp. We simply sum up only the distinct l_{ij} for each j to find the total of the life spans of the sets at edit session i.

Now, we define another estimated value L' for the sum of the error life spans as follows:

$$L' = \sum_{i=1}^{M} l'_i \qquad \text{... (4.3)}$$

where l'_i : the sum of distinct values of l_{ij} for $1 \leq j \leq max_i$

In this definition, the following two assumptions are made.

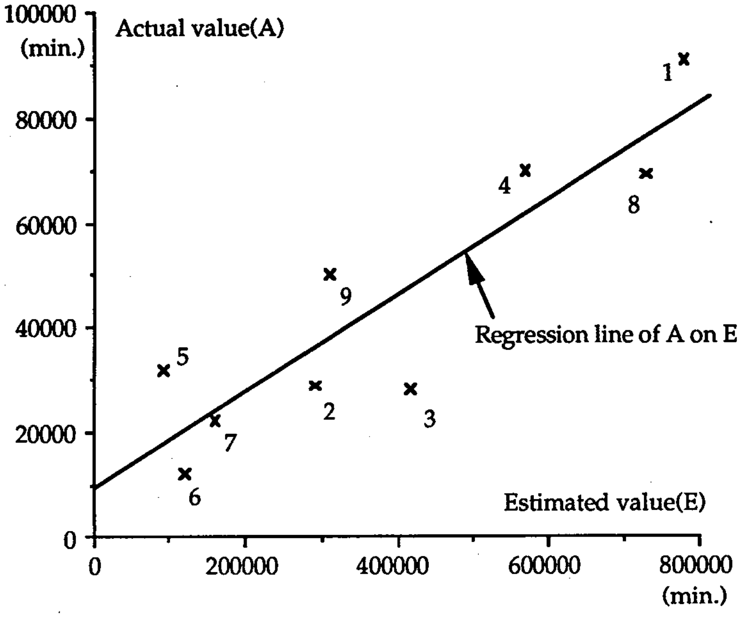


Figure 4.4 Estimated values vs. actual values ΣT_e in Experiment 1.

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

(2) A set of lines, which are created at one edit session and modified at another edit session, corresponds to one error.

We computed L' again using the data of Experiment 1, and compared L' with the sum $\sum T_e$ of the actual error life spans. Figure 4.4 shows the scatter plots of L' versus $\sum T_e$. A coefficient of correlation between them is improved to 0.90. Thus, we can conclude that the estimated value L' is sufficiently equivalent to the sum of the error life span.

4.5 Experiment 3

4.5.1 Experimental data

Experiment 3 evaluated team activities in software development. The programmers were newcomers of a certain computer company. The main characteristics were summarized as follows.

(1) Eight teams of programmers developed the file processing program in a business application using COBOL.

(2) The system (file processing program) consisted of 18 program modules. This partition of program modules was given to each team. However, distribution of modules to members of team was freely determined by a leader of each team.

(3) Each team consisted of 3 to 5 programmers. Teams were organized by an instructor so that the difference among team performances, in an intuitive sense, might be low.

(4) Each team was assigned two terminals. Thus, the capability of accessing terminals seemed relatively to be limited, compared with Experiments 1 and 2.

In Experiment 3, the successive time of each programmer accessing a terminal was counted and used as the time unit for evaluating T_e. In addition, each programmer had to fill in a form of individual effort time for designing, coding, and unit debugging. On the other hand, each team leader also had to fill in a form of team effort time, mainly covering integration of the individuals' work.

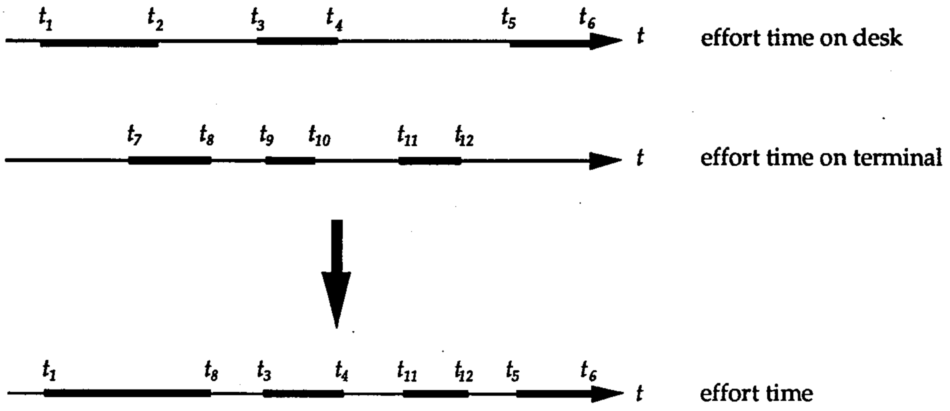


Figure 4.5 Explanation of effort time

The effort time on a terminal was a time duration counting when a programmer or team worked on terminal. Similarly, the effort time on a desk was a time duration of when a programmer or team worked on a desk (not on a terminal). The effort time on a desk was reported from forms. We used a new effort time gotten by merging these two as shown in Figure 4.5.

Only 9 modules out of 18 were studied for data based on the following criteria.

(1) The average of the module size is more than 100 lines. By this, too small modules are excluded from evaluation.

(2) The average of the ratio of data division size over module size is less than 0.5. Thus, the programs, which mainly consist of data definitions, are also excluded.

Table 4.4 Data of Experiment 3

Team	Member	Program size	Sum of error life spans	Total effort time(min.)	Score
#1	m1	289	1490	2009	56
	m2	263	10608	5488	7
	m3	385	7192	2652	21
	m4	137	6109	3633	3
	m5	95	6679	3523	1
#2	m1	365	7899	3999	17
	m2	278	8510	2706	9
	m3	249	6877	2730	9
	m4	155	1855	3766	12
	m5	107	101	2646	113
#3	m1	221	13329	3730	4
	m2	600	37620	3409	10
	m3	362	27689	4809	5
#4	m1	333	22972	4354	5
	m2	230	4896	3039	11
	m3	364	3970	4220	33
	m4	319	21612	3214	5
#5	m1	393	12035	3681	13
	m2	270	1569	4061	46
	m3	342	11907	4173	10
	m4	240	15147	3886	4
#6	m1	569	22470	3429	14
	m2	375	11789	3243	12
	m3	155	1818	2874	13
#7	m1	387	17634	3768	8
	m2	328	14194	3407	8
	m3	264	4747	2704	15
	m4	126	5092	3627	3
	m5	172	208	2467	142
#8	m1	583	14497	4426	23
	m2	203	2268	3211	18
	m3	233	14775	4699	4
	m4	169	25621	5366	1

The experimental data are summarized in Table 4.4. The values for the sum of error life spans are calculated using formula (4.3) in Section 4.4. Table 4.4 shows the program size (which is the number of lines in the final program text), the sum of error life spans, the total effort time (time estimated, as shown in

Figure 4.5, by effort time reported by programmer and terminal access time traced automatically) and score *SI*. As the time unit for error life span, we used effort time.

4.5.2 Evaluation

(1) ΣT_e vs. effort time of a programmer

Figure 4.6 shows the scatter plots of ΣT_e versus the total effort time of each programmer. (In Section 4.2, we compared ΣT_e with the total terminal access

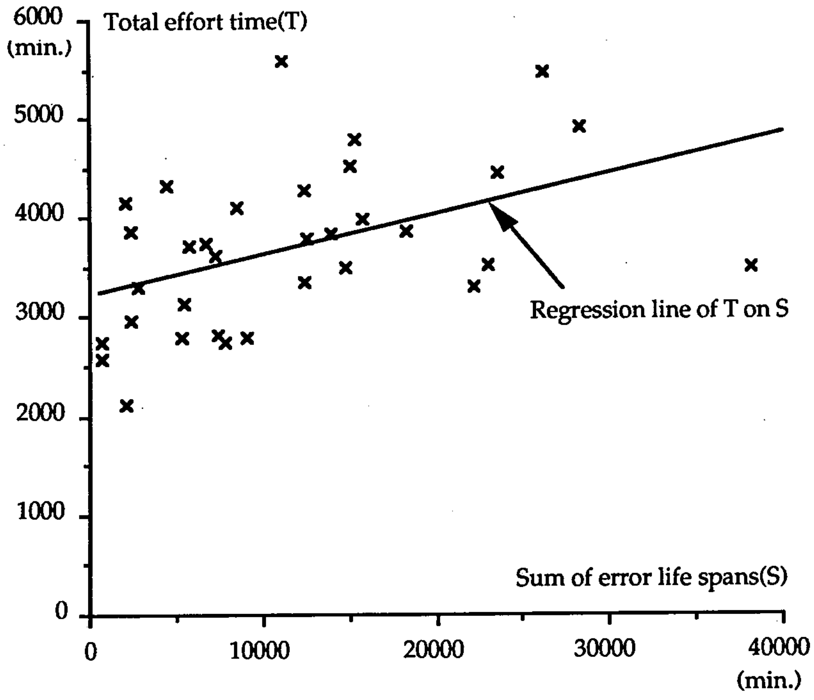


Figure 4.6 Sum of error life spans vs. total effort time (Experiment 3)

Table 4.5 Coefficient of correlation between ΣTe and total effort time

Module	1	2	3	4	5	6	7	8	9
r	-0.46	0.85	0.62	0.58	0.51	0.39	0.66	0.87	0.44

time.) A coefficient of correlation between them is 0.46. It is not high compared with the result of Section 4.2. (In Section 4.2, a coefficient of correlation between them is 0.82.)

(2) ΣT_e vs. effort time of module

In addition, we have investigated for the 9 modules the relation between the sum of error life spans and the total effort time devoted to the module development. Table 4.5 shows a coefficient of correlation between these two. It is clear that there is much difference among these values (the highest value is 0.87 and the lowest value is 0.39 among positive values). The main reason for these results is the difference in the degree of coupling among modules. For module 1, it is unexpected that a coefficient becomes negative.

(3) Team performance

Table 4.6 shows three scores (*ST1*, *ST2* and *ST3*) and the team debugging effort time of 8 teams (#1, #2, ..., #8). The team debugging effort time is the total effort time for debugging after each unit test for each module has been completed.

From Table 4.6, the following aspects are observed as for team performance.

Team performance score *ST1* (model *M1*)

the highest score	team #7	176

Table 4.6 Data of team scores

Team	*ST1*	*ST2*	*ST3*	Team debugging effort time(min.)
#1	88	18	43	490
#2	160	32	53	460
#3	19	6	18	2200
#4	54	14	29	2170
#5	73	18	38	980
#6	39	13	33	650
#7	176	35	39	570
#8	46	12	25	1430

the lowest score team #3 19

Team performance score *ST2* (model *M2*)

the highest score team #7 35

the lowest score team #3 6

Team performance score *ST3* (model *M3*)

the highest score team #2 53

the lowest score team #3 18

We have evaluated the correlation among the three team performance scores *ST1, ST2* and *ST3* (see Table 4.7). There are high correlations among *ST1, ST2* and *ST3,* especially between *ST1* and *ST2* (A coefficient of correlation is 0.99).

(4) Team performance vs. debugging effort time

We have investigated team debugging effort time as shown in Table 4.6 and compared it with team performance scores (*ST1, ST2* and *ST3*). Table 4.7 shows the coefficients of correlations between team scores and team debugging

Table 4.7 Coefficient of correlation among team scores and team debugging effort time

	ST1	*ST2*	*ST3*
ST2	0.99	——	——
ST3	0.80	0.79	——
Team debugging effort time	-0.69	-0.66	-0.83

effort time. Among the three team performance scores, *ST3* has the highest correlation with team debugging effort time (A coefficient of correlation is -0.83). It might be said that the team performance score *ST3* (thus, the model *M3*) is the most appropriate one for evaluating team performance in software development.

(5) Team score vs. load distribution

The ratio, final program size to score of programmer j (represented by K_j), is compared in Table 4.8. We also evaluate the difference between an optimal ratio of final program size (K_{opt}) and actual ratio. The result shows the maximum value of K_{opt} is 62.3, and the minimum value of K_{opt} is 7.2. It is observed that the value of K_{opt} is apt to become larger as the number of programmers becomes smaller (then the size of program, to be developed by one programmer, becomes larger).

To evaluate the difference between K_j and K_{opt}, we have calculated the following value D_K as shown in Table 4.8.

$$D_K = \frac{\sum_{j=1}^{n} | K_j - K_{opt} |}{K_{opt}} \qquad \text{... (4.4)}$$

Table 4.8 Ratio of size over score $\left(K_j = \frac{L_j}{SI_j}\right)$

Team	K_1	K_2	K_3	K_4	K_5	K_{opt}	$D_K = \frac{\sum_{j=1}^{n} \lvert K_j - K_{opt} \rvert}{K_{opt}}$	$D_{ST} = \frac{ST1\text{-}ST3}{ST1}$ (%)
#1	5.2	37.6	18.3	45.7	95.0	13.3	11.4	51
#2	21.5	30.9	27.7	12.9	0.9	7.2	9.8	67
#3	55.3	60.0	72.4	—	—	62.3	0.3	5
#4	66.6	20.9	11.0	63.8	—	23.1	4.2	46
#5	30.2	5.9	34.2	60.0	—	17.1	4.9	48
#6	40.6	31.3	11.9	—	—	28.2	1.1	15
#7	48.4	41.0	17.6	42.0	1.2	7.3	17.3	78
#8	25.3	11.3	58.3	169.0	—	25.8	7.4	46

If there exists no difference between K_j and K_{opt} then D_K becomes 0. At that time, the following relation holds:

$$ST3 = \sum_{j=1}^{n} ST_j = ST1 \qquad \text{... (4.5)}$$

So, to evaluate the difference between $ST3$ and $ST1$, we have calculated the following value D_{ST} as shown in Table 4.8.

$$D_{ST} = \frac{ST1 - ST3}{ST1} \times 100\ (\%) \qquad \text{... (4.6)}$$

Naturally, there is a high correlation between D_K and D_{ST}.

Consider teams #2 and #3 which take respectively the highest and the lowest values of team performance score $ST3$ in Table 4.6. Table 4.8 shows that for team #2 there are large differences among K_j for each programmer and K_{opt}. As the result, the value of D_K of team #2 is relatively large. On the other hand, Table 4.8 shows that K_j for each programmer in team #3 are almost equal to K_{opt}.

(As for D_K, team #3 takes the lowest value among these eight teams.) Therefore, it is concluded that team #2 is superior to team #3 with respect to total team performance. But in contrast to this, with respect to load distribution in a team, team #3 is superior to team #2. This tendency is clearly observed: from the values of D_{ST} in Table 4.8, team #2 decreases by 67% in their performance and team #3 decreases by only 5% from their optimal (maximum) performance.

Chapter 5: Software Development Project Management

5.1 Major functions of management

In the classic management model [Mackenzie 1969] [Thayer 1988], management is partitioned into five separate functions or components: planning, organizing, staffing, directing, and controlling. Definitions or explanations for these five functions are shown in Table 5.1.

These functions can be classified into two types. The first type includes planning, organizing and staffing. These are executed before constructing the activities of the software project. Their purpose is to enable the objectives of the project to be accomplished effectively. The second type of functions includes directing and controlling. These are executed dynamically during the software

Table 5.1 Major functions of management [Thayer 1988]

Activity	Definition or Explanation
Planning	Predetermining a course of action for accomplishing organizational objectives.
Organizing	Arranging and relating work for accomplishment of objectives and the granting of responsibility and authority to obtain those objectives.
Staffing	Selecting and training people for positions in the organization.
Directing	Creating an atmosphere that will assist and motivate people to achieve desired end results.
Controlling	Measuring and correcting performance of activities toward objectives according to plan.

construction phase in the project and are done to carry out the project in pursuance of the prescribed plan if deviation from the prescribed plan occurs.

The second type of functions, directing and controlling, are more important than the first type. Since it is impossible when a project starts to predict exactly all phenomena that will affect it, we have to develop a mechanism to correct deviations and ensure the execution of the project in pursuance of the prescribed plan.

Therefore, this thesis focuses on the second type of functions: directing and controlling. We focus especially on controlling since we want to devise an environment which can manage the software development project.

5.2 Project control

Controlling a software development project is defined as all the management activities that ensure that the actual work goes according to plan [Thayer 1988]. To control the project, a manager has to know the actual state of the project, know the difference between the prescribed plan and the actual state, and help the developers accomplish the prescribed plan.

Figure 5.1 shows the data flow between the development environment and the management environment for controlling a project. The manager collects process/product data to assess the actual state of the project. After analyzing the collected data, the manager provides feedback to a developer to help him or her correct their activities and accomplish the prescribed plan. Therefore, data collection and information feedback are essential activities for controlling the software development project. The full support of a

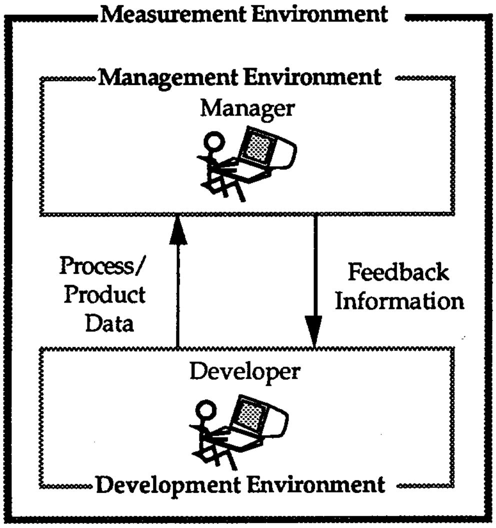


Figure 5.1 Controlling software development project

development environment and a management environment is necessary to control the software development project.

Based on these considerations, we propose a new environment, measurement environment, which consists of a development environment and a management environment. The measurement environment consists mainly of four logical units: Data Collection, Data Management, Data Analysis and Information Feedback (see Figure 5.2). Among these four units, Data Collection is included in the development environment and the rest are included in the management environment. The details of the measurement environment will be described in Chapter 6.

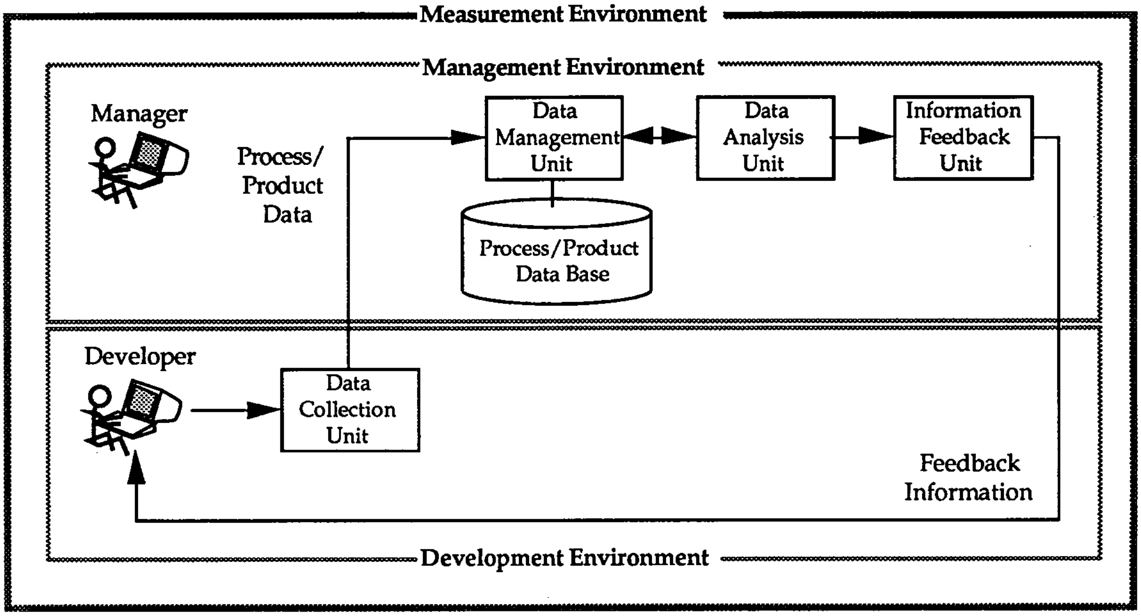


Figure 5.2 Basic units of measurement environment

5.3 Data for controlling

To control the software project effectively, we have to collect and analyze objective and quantitative data which represent the activities of developers. In this thesis, we concentrate on programmer productivity as the metric for evaluating the activities of developers.

Programmer productivity is well explained by using an input-process-output scheme (see Figure 5.3) as follows [Chen 1978]: the programmer is a processor, the input is a program specification, and the output is a set of programs written in a good programming style. Then a measure of programmer productivity can be defined as the number of valid source statements coded per

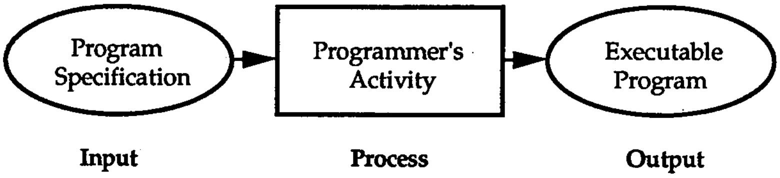


Figure 5.3 Input-process-output scheme

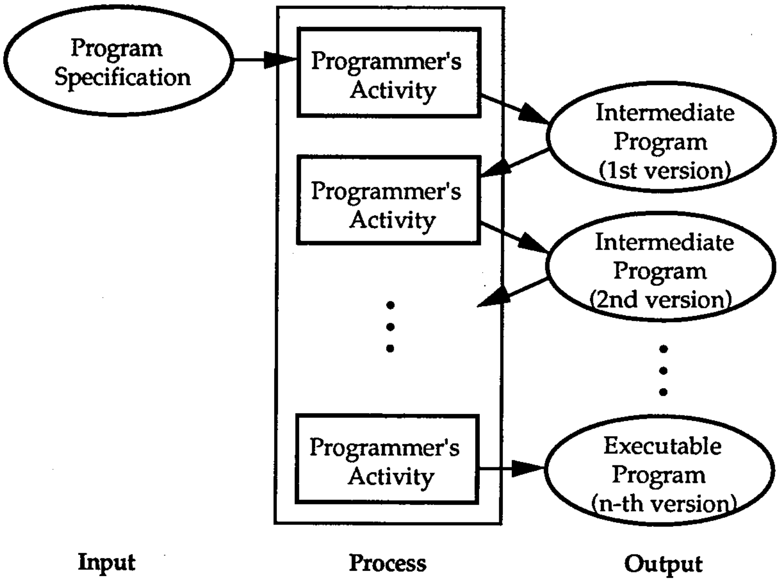


Figure 5.4 Extended input-process-output scheme

busy hours, where valid source statements are the source statements of an executable computer source program.

In several earlier papers [Chen 1978] [Walston & Felix 1977], the measure of programmer productivity in software development was defined similarly as the ratio of the quality/quantity of the resulting program to the programming efforts necessary to arrive at a satisfactory program. For example, Walston and

Felix [Walston & Felix 1977] defined the measure of programming productivity as the ratio of delivered source code to the total effort (in man-months) required to produce the code.

In this thesis, we are interested in not only measuring programmer productivity, but also in improving programmer productivity based on analysis results from software developmental data. However, the input-process-output scheme in Figure 5.3 is not sufficient to describe programmer activity since the scheme cannot answer questions of why a programmer expends such a large (or small) amount of effort. Therefore, we extend the input-process-output scheme to the one shown in Figure 5.4. In practical software development, the first version of a program usually does not satisfy a given specification since the program may not fulfill all the functions required in the specification, or may contain many errors. Then the programmer modifies or debugs the program a number of times and finally gets a program satisfying the specification.

Generally, it is very hard to follow and analyze programmer activity directly during the software development process. Thus we try to estimate programmer activity indirectly by analyzing the history of the program modifications. By taking the programming efforts into consideration as well, a more detailed analysis may be possible and more valuable information can be collected from programmer. Programming efforts are divided into two categories: (1) efforts to produce code that implements a given specification and (2) efforts to modify the code, up through the completion of testing (see Figure 5.5). Out of these two types of efforts, the second may be a more appropriate focus for the improvement of programmer productivity.

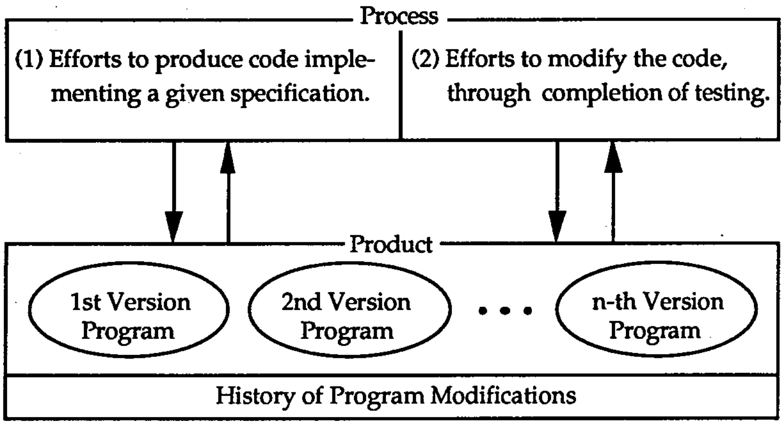

Process
(1) Efforts to produce code implementing a given specification.
(2) Efforts to modify the code, through completion of testing.
Product
1st Version Program
2nd Version Program
• • •
n-th Version Program
History of Program Modifications

Figure 5.5 Analytical viewpoint for programmer activity

According to earlier studies [Walston & Felix 1977], we use the programming efforts and the quality/quantity of the resulting program as two typical primitive metrics of programmer productivity. In addition, in order to trace and analyze the programmer's activities, we introduce a concept of program modifications as the third metric. The following subsections summarize these three metrics.

5.3.1 Programming efforts

Programming efforts are usually defined as the time, cost or work necessary to produce a satisfactory program during software development [Chen 1978] [Dunsmore & Gannon 1980] [Musa et al. 1987] [Walston & Felix 1977]. Typically, the programmer's computer usage or execution time is used as a measure to evaluate programming efforts since it can be collected automatically using, for example, functions of the operating system.

The following four kinds of data (1) through (4) below, are potentially useful for measuring programming efforts. Unfortunately, none of them represents programmer working time precisely. Therefore, it is necessary to select an appropriate measure or to use several measures at the same time, when programming activities are evaluated under a particular project environment.

(1) Calendar time

Calendar time is the familiar time with which we are normally acquainted [Musa et al. 1987]. Man-months or man-hours is a measure in this category. This measure includes both working time at the desk and on the computer (terminal), and can be collected very easily. Therefore, calendar time is considered to be a basic measure of the programming efforts. However, the following must be assumed to validate calendar time as a measure: (1) the programmer's working time is relatively fixed for each day, (2) the range of fluctuation of the working time for each day is negligibly small, and (3) the working time is measurable.

(2) Terminal access (usage) time

Terminal access time represents the elapsed time from the beginning of a programmer's work on the computer terminal to the end. Working time at the desk must be excluded. Thus terminal access time is effectively and precisely used to evaluate effort time only when the programmer spends most of his or her working time on the terminal or when it is a small-scale project [Matsumoto et al. 1987].

(3) Number of command executions

This number represents the frequency of computer operations (i.e., program edits, compilations, and program executions for testing and debugging) on the computer terminal. The number of program testings and job-steps are similar measures in this category [Basili 1980] [Basili & Reiter 1979]. A job-step is a single programmer-oriented activity performed on the computer at the operating system command level [Basili 1980] [Basili & Reiter 1979]. Basili and Reiter [Basili & Reiter 1979] found that the job-step measure significantly differentiates development environments and that good methodology leads to a small number of job-steps. The number of command executions should be used as a measure instead of physical time when idle time on the terminal is relatively high and cannot be disregarded.

(4) CPU time for command execution

CPU time represents the time that is actually spent by a processor in executing each command. For a certain type of command (i.e., a command to execute a program), the execution time depends strongly on the input data. Thus, the CPU time spent by the processor can be regarded as the programmer's working time. It is generally accepted that software reliability models based on execution time are superior to ones based on calendar time [Musa et al. 1987].

5.3.2 Quality/Quantity of resulting program

There have been many studies [Basili 1980] [Matsumoto et al. 1988a] [Musa et al. 1987] evaluating the quality/quantity of programs. Out of these studies have come several practical models of reliability [Matsumoto et al. 1988a] [Musa et al. 1987]. The reliability of a resulting program is generally considered one of the most important aspects of quality. Unfortunately, the evaluations based on the models still include subjective decisions.

5.3.3 Program modifications

In addition to the programming efforts and the quality/quantity of the resulting program, the modifications of the program are taken into consideration in evaluating programmer activity. The modifications of the program may include valuable information for improving programmer productivity.

Dunsmore and Gannon [Dunsmore & Gannon 1980] defined a measure of program modifications as a textual revision in the source code of a module. The rule for counting program modifications is that one program modification is concerned with a contiguous set of concrete statements that represent a single abstract instruction. They showed that program modifications had a high correlation with total error occurrences.

In this thesis, program modifications are evaluated by counting the changes for each line (statement) of the program. Program modifications are classified into three patterns, according to the corresponding commands: 1) append, which creates new program lines and adds new statements to the program, 2) change, which replaces some existing statements with new statements, and 3) delete, which removes some existing statements.

In this thesis, the concept of error life span is also introduced to evaluate programmer productivity [Matsumoto et al. 1988c] [Matsumoto et al. 1987]. As mentioned in Section 3.2, the life span of error T_e is defined as the length of time from when error e is manifested itself in the software to the time when error *e* is removed from the software. Concepts related to error life span can be seen in several earlier papers [Mills 1976] [Weiss & Basili 1985]. For example, Mills [Mills 1976] introduced the concept of error days for estimating the quality of an acceptable system. Next, Weiss and Basili [Weiss & Basili 1985] used program modifications as a way of evaluating the software development processes.

The usefulness of error life span has already been verified by experimental data analysis [Matsumoto et al. 1988c] [Matsumoto et al. 1987], as described in Chapter 4. In the analysis, experimental data was collected from student projects. It was shown that the error life span has a high correlation with programmer productivity. The time used for counting error life span is actual terminal access time (rather than calendar days used for the error days). A method to evaluate automatically the number of errors based on the program modifications has already been prepared for error life span [Matsumoto et al. 1988c] [Matsumoto et al. 1987], as described in Section 4.4.

Chapter 6: Measurement Environment : GINGER

6.1 Measurement-based control

We believe that the proper use of *software metrics, measurement,* and *models* is essential to the successful management of software development and maintenance. Generally, software metrics are used to characterize quantitatively the essential features of software so that classification, comparison, and mathematical analysis can be applied. After a number of useful metrics are identified, the measurement is executed by applying the selected metrics to software in an algorithmic and objective fashion. For the measurement, it is required that the values of the selected metrics are consistent among different software products and are independent of the measurer.

In order to control the software development and maintenance processes, it is important to model certain interesting factors (metrics) such as effort and defects, based on other metrics that are available. Appropriate management decisions can be made to influence these factors so that management goals can be realized. In other words, the proper use of software metrics, measurement, and models has the potential of allowing us to estimate accurately the cost of producing software (or the cost in maintaining the same) [Conte et al. 1986].

6.2 System organization

Figure 6.1 outlines the system architecture, which describes interactive information processing between the development environment and the

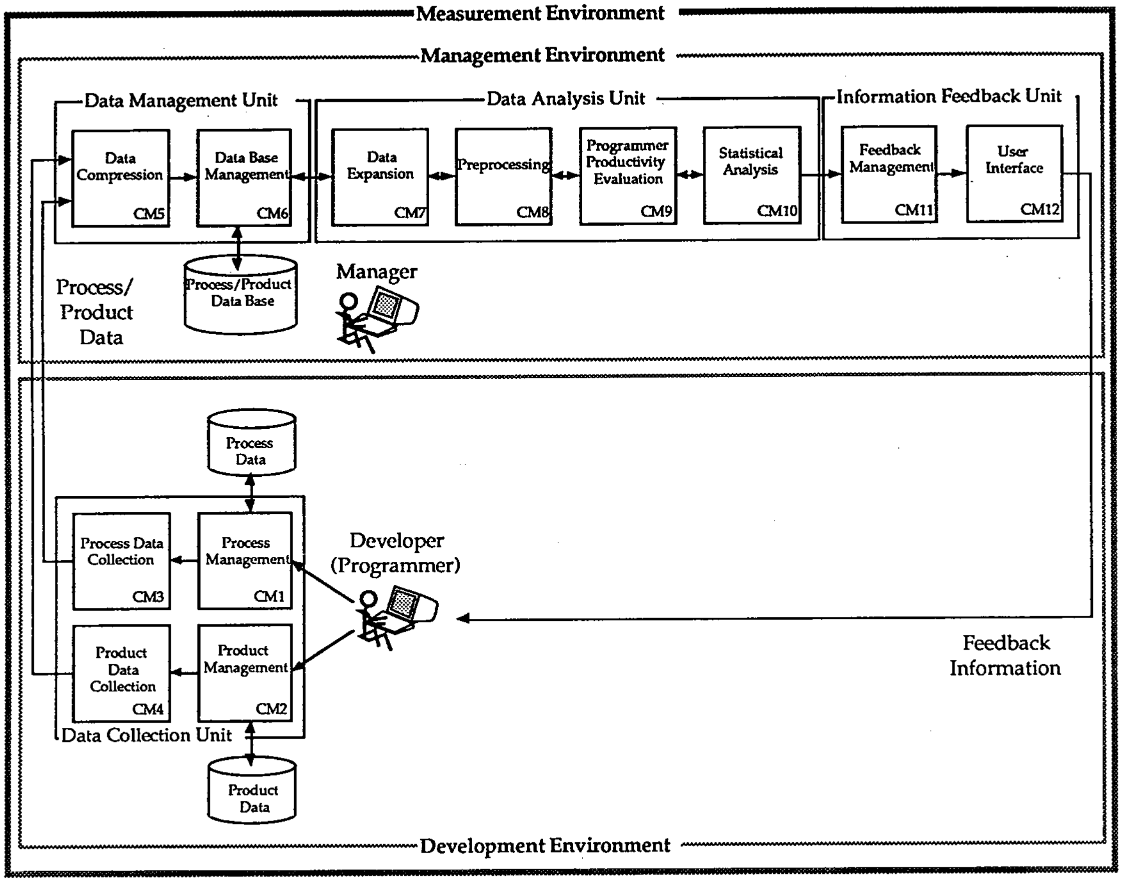


Figure 6.1 System architecture of the measurement environment

management environment. The data flow is also explicitly given in Figure 6.1. The system consists mainly of four logical units: Data Collection, Data Management, Data Analysis and Information Feedback. The architectural components of each of these four units are described in detail below and in [Torii 1990].

(A) Data Collection Unit

The Data Collection Unit consists of four components, (CM1) through (CM4):

The (CM1) Process Management component supports and controls a programmer's regular activities for software development. It allows the interactive operations between the programmer and the computer to be executed at the operating system command level.

The (CM2) Product Management component maintains the product, i.e., the program text developed by programmer. It also maintains relevant information for each product such as the date, time, and access rights. It also controls access to the product.

The (CM3) Process Data Collection component accumulates all data concerned with the programming efforts, which are passed from the Process Management component indirectly. (Note that the data are not taken directly from the programmer. The details will be described in subsection 6.4.2.)

The (CM4) Product Data Collection component accumulates a series of intermediate programs (including the resulting programs) and collects historical data about program modifications. Data collection also occurs indirectly through the Product Management component. (Note that the data are not taken directly from the programmer. The details will be described in subsection 6.4.2.)

The functions mentioned in (CM1) and (CM2) are usually provided as basic functions in an operating system. The data collection in (CM3) and (CM4) should be executed completely automatically. In particular, the data should be collected indirectly from the programmer's activities without obstructing his or her regular activities.

(B) Data Management Unit

The Data Management Unit consists of two components, (CM5) and (CM6):

The (CM5) Data Compression component implements a memory save by storing as little data as possible. For a series of intermediate products, only the difference between two successive data is stored. The details will be described in subsection 6.4.2.

The (CM6) Data Base Management component provides 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. The relevant information, such as the date and the names of project, team, programmer, file, module, and statement, is added to each original data. The details will be described in subsection 6.4.3.

(C) Data Analysis Unit

The Data Analysis Unit consists of four components, (CM7) through (CM10):

The (CM7) Data Expansion component restructures the data by inserting the compressed parts which were deleted in 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). Additionally, the comments and the blank statements are deleted.

The (CM9) Programmer Productivity Evaluation component calculates several values according to the algorithms or guidelines for measurement. The evaluations are assumed to be for purposes of programmer productivity (the programming efforts, the quality/quantity of resulting programs, and the program modifications discussed in Section 5.3). The details will be described in subsection 6.4.4.

The (CM10) Statistical Analysis component applies statistical analysis methods to collected data (CM8) and to the results of the evaluation (CM9). In an analysis, to compare results, the historical data of the programmer and the data on similar prior projects may be used extensively. Predictions for the current project based on such data are very important for managing the project (including the programmers). If the project turns out to be unsuccessful, the request for information about the results is passed to the Information Feedback Unit.

(D) Information Feedback Unit

The Information Feedback Unit consists of two components, (CM11) and (CM12):

The (CM11) Feedback Management component determines the timing of feedback and the details of information to be returned. Information feedback is also executed when the programmer requests it.

The (CM12) User Interface component provides an effective presentation of feedback information to the programmer.

6.3 Functionality

This section elaborates on the functionality of the GINGER system. The GINGER system provides ten functions which are classified into four groups; (A) kernel, (B) metric-oriented, (C) education-oriented and (D) project-oriented. First, the fundamental functions for the data collection and the analysis system are described.

(A) Kernel functions

A-1: Automatic data collection (CM3 and CM4 in Figure 6.1)

To assure the reliability of data from the software development process, the data should be collected entirely automatically. In the GINGER system, automatic data collection about the programmer's activity is achieved by separating data collection (CM3 and CM4) from data management (CM1 and CM2), as shown in Figure 6.1. Therefore, it is essential for the system that the programming environment allow the programmer to do as much of his or her task on the computer as possible.

A-2: Statistical analysis (CM10 in Figure 6.1)

To increase the usefulness of the analysis results, the system supports statistical analysis methods and provides interpretations. Predictions based on the statistical analysis are especially important to improvement-oriented systems.

A-3: Information feedback (CM11 and CM12 in Figure 6.1)

For an improvement-oriented system, collecting and analyzing data on the process and product are necessary, but they are not sufficient. The analyzed results should be fed back into the software development process (especially to the programmer). To realize effective improvement, the system should provide a good form of presentation which makes it easy for programmers to understand the feedback information.

A-4: Data base for experience (CM6 in Figure 6.1)

To get good feedback information for improving programmer productivity, the historical data (collected data and analysis results) on prior projects should be utilized extensively. In the GINGER system, collected data and the results of analysis can be stored in a data base and can be retrieved at any time.

(B) Metric-oriented functions

The following two items are required for evaluating programmer productivity since no definite measures exist to evaluate programmer efforts or the quality/quantity of the resulting program as mentioned in Section 5.3.

B-1: Management of large data (CM5, CM6 and CM7 in Figure 6.1)

The data on program modifications are collected by comparing successive versions of the program. To manage this large amount of data, the system uses Data Compression (CM5). Thus effective storage and retrieval of the analysis results are achieved.

B-2: Evaluation by multi-measures (CM8 and CM9 in Figure 6.1)

A careful analysis evaluates several measures at the same time and then selects the most appropriate one. The system provides such a mechanism by collecting substantial elementary data in Data Collection (CM3 and CM4) and by preparing all of the logical information needed for evaluation in Data Analysis.

(C) Education-oriented functions

In the educational and training environments in universities, programmers are novices. In addition, many trainees work on the same exercise concurrently. Thus, the following three things are especially necessary in such environments.

C-1: Unobtrusive data collection (CM3 and CM4 in Figure 6.1)

In order to educate and train novice programmers, it is undesirable to interrupt or restrict their programming activity. Therefore, the system (CM1 and CM2) is embedded in a software development environment and the data is collected from the programmers without their knowledge.

C-2: Advice to programmers (CM11 in Figure 6.1)

Novice programmers generally do not know what they should do in order to improve their productivity. The system should provide novice programmers with not only the analysis results but also with helpful advice.

(D) Project-oriented functions

In educational and training environments, small-scale projects are executed concurrently. Here, small-scale implies that the program size is small, the development period is short, and so on. The following two things are required to apply the GINGER system to small-scale projects.

D-1: Data collection with low overload (CM3 and CM4 in Figure 6.1)

For small-scale projects, the computer system to be used for development tends to be not so powerful. If data collection (and analysis) needs large amounts of computation, it will interfere with program development. Then, the project may not succeed and reliable data may not be collected. Therefore, the system is designed to make the load for data collection (and analysis) as small as possible.

D-2: Real-time information feedback (CM11 and CM12 in Figure 6.1)

In small-scale projects the development period may not be very long. Therefore, data collection and data analysis can be executed quickly in the system. Additionally, feedback information is returned to the programmers in real-time as the projects proceed.

6.4 Prototype system

6.4.1 Characteristics of the prototype

A prototype system is currently being developed on a UNIX environment [Torii 1990]. In the prototype system, functions A-1 through A-4, B-1, B-2, C-1, C-2, D-1, D-2 stated in Section 6.3 have been implemented. The main characteristics of the prototype, with respect to implementation, are summarized as follows:

(I1) The system has been implemented using C language 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 it. In particular, in implementing the Data Collection, Data Management and Information Feedback Units, 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 programmers' workstations, and all other units are implemented on manager's workstation, as shown in Figure 6.1.

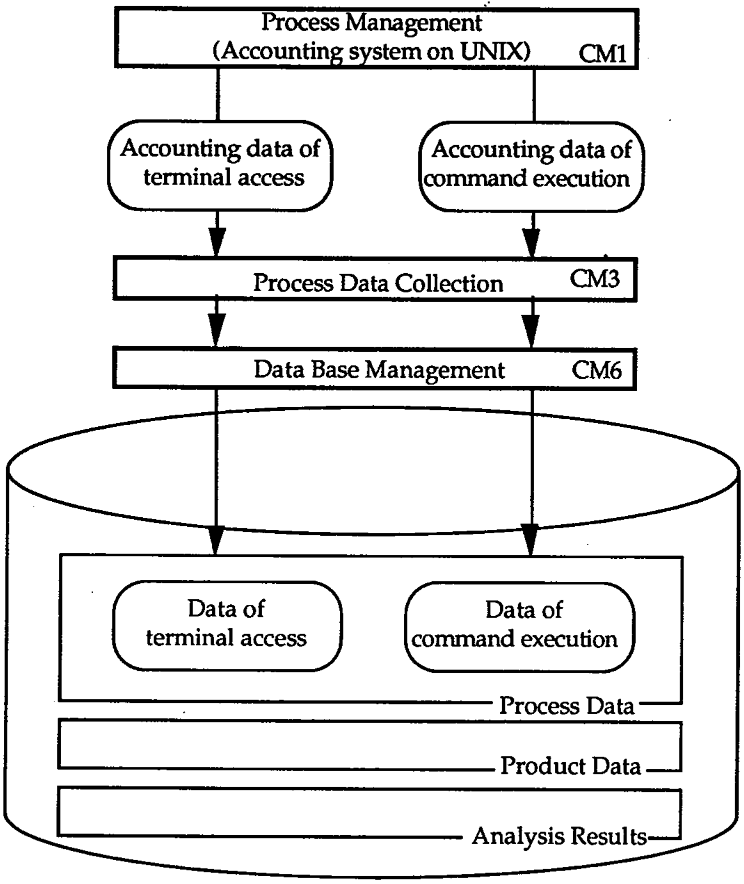

Process Management
(Accounting system on UNIX)
CM1
Accounting data of terminal access
Accounting data of command execution
Process Data Collection
CM3
Data Base Management
CM6
Data of terminal access
Data of command execution
Process Data
Product Data
Analysis Results
Process/Product Data Base

Figure 6.2 Computation of process data on UNIX

6.4.2 Collection of process/product data

(1) Process data (see Figures 6.2 and 6.3)

The process data in the Process/Product Data Base consists of two kinds of data: terminal access and command execution. They are accumulated by Process Data Collection (CM3). Figure 6.2 shows a data flow in the computation of process data. Accounting data (see Figure 6.3(a)) is transformed into the data of

Accounting data of terminal access

```
kusumoto  console        Wed Nov 16 09:49 - 12:41   (02:52)
kusumoto  console        Tue Nov 15 12:40 - 14:37   (01:57)
```

```
kusumoto  console        Fri Oct 21 15:03 - 16:54   (03:51)
kusumoto  console        Fri Oct 21 11:05 - 12:34   (01:29)
kusumoto  console        Thr Oct 20 10:23 - 10:30   (00:07)
```

Accounting data of command execution

```
a.out          kusumoto console   0.39 secs Wed Nov 16 12:35
ls             kusumoto console   0.09 secs Wed Nov 16 12:35
hc             kusumoto console   0.06 secs Wed Nov 16 12:35
```

```
vi             kusumoto console   5.06 secs Wed Oct 20 10:29
ls             kusumoto console   0.06 secs Wed Oct 20 10:28
more           kusumoto console   2.34 secs Wed Oct 20 10:27
```

(a) Accounting data

Data of terminal access

```
1988-10-20 10:23 10:30
1988-10-21 11:05 12:34
1988-10-21 15:03 16:54
```

```
1988-11-15 12:40 14:37
1988-11-16 09:49 12:41
```

Access date (year-month-day) | Login time (hour:minute) | Logout time (hour:minute)

Data of command execution

```
1988-10-20 10:29 vi          5.06 secs
1988-10-21 11:25 vi          4.07 secs
1988-10-21 11:26 cpp         0.30 secs
```

```
1988-11-16 12:35 hc          0.06 secs
1988-11-16 12:35 a.out       0.39 secs
```

Execution date (year-month-day) | Execution time (hour:minute) | Command name | CPU time (1/100 seconds)

(b) Transformed data

Figure 6.3 Example of process data

terminal access and command execution (see Figure 6.3(b)) by removing some pieces of information and rearranging the order of the rest.

Figure 6.3(a) shows an example of accounting data given by the accounting system on UNIX. The accounting data of terminal access consists of (1) the name of programmer who has accessed the terminal, (2) the identifier of the terminal, (3) the access date, (4) the login time (in minutes) when the terminal session was begun by the login command, (5) the logout time (in minutes) when the terminal session was ended by the logout command, and (6) the time duration from login to logout. A set of these six data items is recorded for each terminal access.

On the other hand, the accounting data of command execution consists of (1) the command name, (2) the programmer 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. A set of these six data items is recorded for each command execution (see Figure 6.3(a)).

The details of the process data to be transformed are shown in Figure 6.3(b). The terminal access data consists of the access date, login time, and logout time for each terminal access. The command execution data consists of the execution date and time, the name of command, and the amount of CPU time.

The accounting data for each programmer are obtained by using the commands "last" and "lastcomm" prepared by the accounting system [UNIX 1986]. Usually, the accounting system is set to delete these data at specific intervals (i.e., at each weekend or the end of each month). Therefore, a C program was developed and inserted into Process Data Collection (CM3) to collect these data just before the accounting system deletes them.

The transformation from the accounting data (in Figure 6.3(a)) to the process data (in Figure 6.3(b)) consists of three successive steps (CM1, CM3 and CM6). These are briefly summarized as follows:

Process Data Computation Procedure (see Figure 6.2)

Step 1: Retrieve the accounting data by the commands "last" and "lastcomm" (Process Management).

Step 2: Sort these accounting data in chronological order, and extract the key information from them (Process Data Collection).

Step 3: Store the data (see Figure 6.3(b)) in the Process/Product Data Base (Data Base Management).

(2) Product data (see Figures 6.4 and 6.5)

The product data consists of the history of modifications and the latest version of the file. These data are accumulated for each file by Product Management (CM2) and Product Data Collection (CM4). Figure 6.4 shows the data flow of the computation of product data, where several functions provided by the UNIX file system [UNIX 1986] are utilized extensively. The program text developed by the programmer is managed as files. For each file, relevant data such as date, time and access rights are also maintained by the file system. The product data are collected by applying the following procedure at five minute intervals:

Product Data Computation Procedure (see Figure 6.4)

Step 1: Find the latest time t_u and a file F_u, such that the file F_u was updated at t_u and t_u is within the past five minutes (Product Management).

Step 2: Make a pair (F_u, t_u) for the file F_u and the time t_u obtained at Step 1 (Product Data Collection).

Step 3: Compute a difference ΔF between the new version of the file F_u and the latest version F_l in the Process/Product Data Base. This computation is realized by using the file comparator "diff" [UNIX 1986] in UNIX (Data Compression).

Step 4: Add the pair $(\Delta F, t_u)$ to the history of modifications and replace the latest version of file F_l by file F_u (Data Base Management).

The details of the product data are shown in Figure 6.5. The history of modifications includes a header and a series of pairs of $(t_u, \Delta S)$, where t_u is an updated time and ΔS is a set of scripts (see Figure 6.5(a)). The updated time t_u is recorded in seconds. The set of scripts ΔS, which corresponds to the difference ΔF in Step 3, is generated by the file comparator command "diff" [UNIX 1986]. (There are five types of scripts. The description of each script type is summarized in Table 6.1). Generally, more than one script is generated to represent one updating of a file. One can be assured that the original version of the file is reproduced by giving ΔS to the editor "ed"[UNIX 1986] in UNIX. The program modifications are calculated based on this script ΔS, as is discussed later.

6.4.3 Management of project data

The data structure of the Process/Product Data Base (see Figure 6.1) is briefly shown in Figure 6.6. The relevant data is managed for each project and all project data is arranged for each programmer. The programmer's data consists of

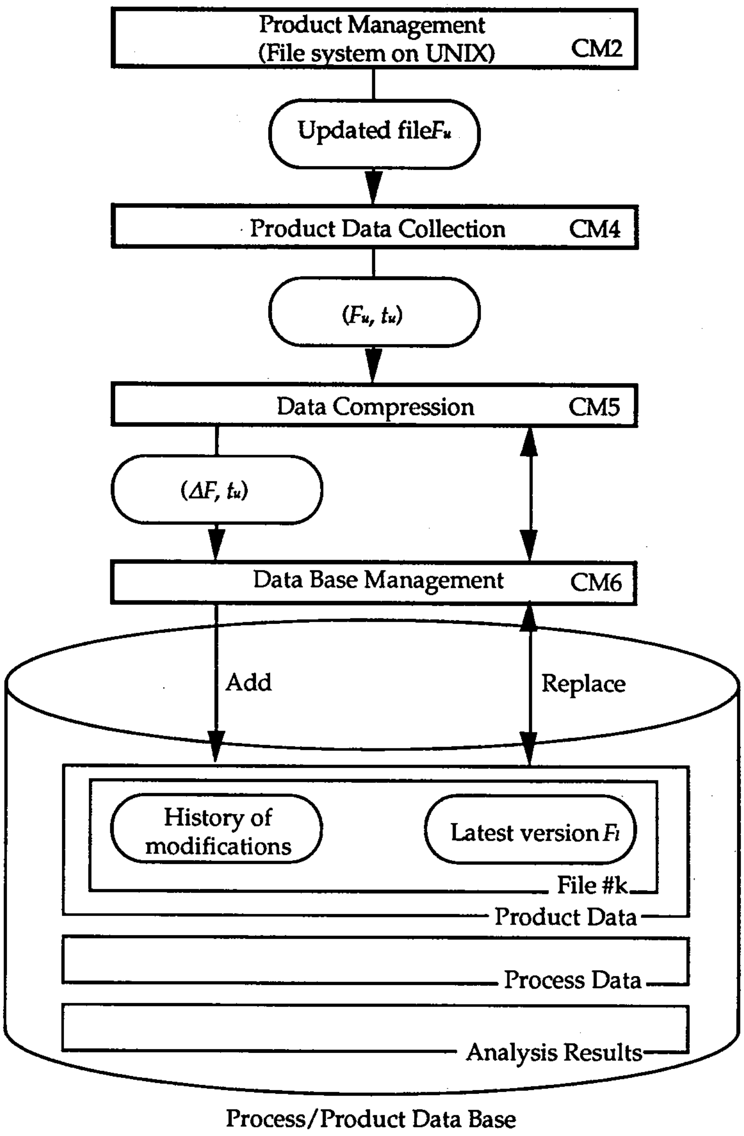
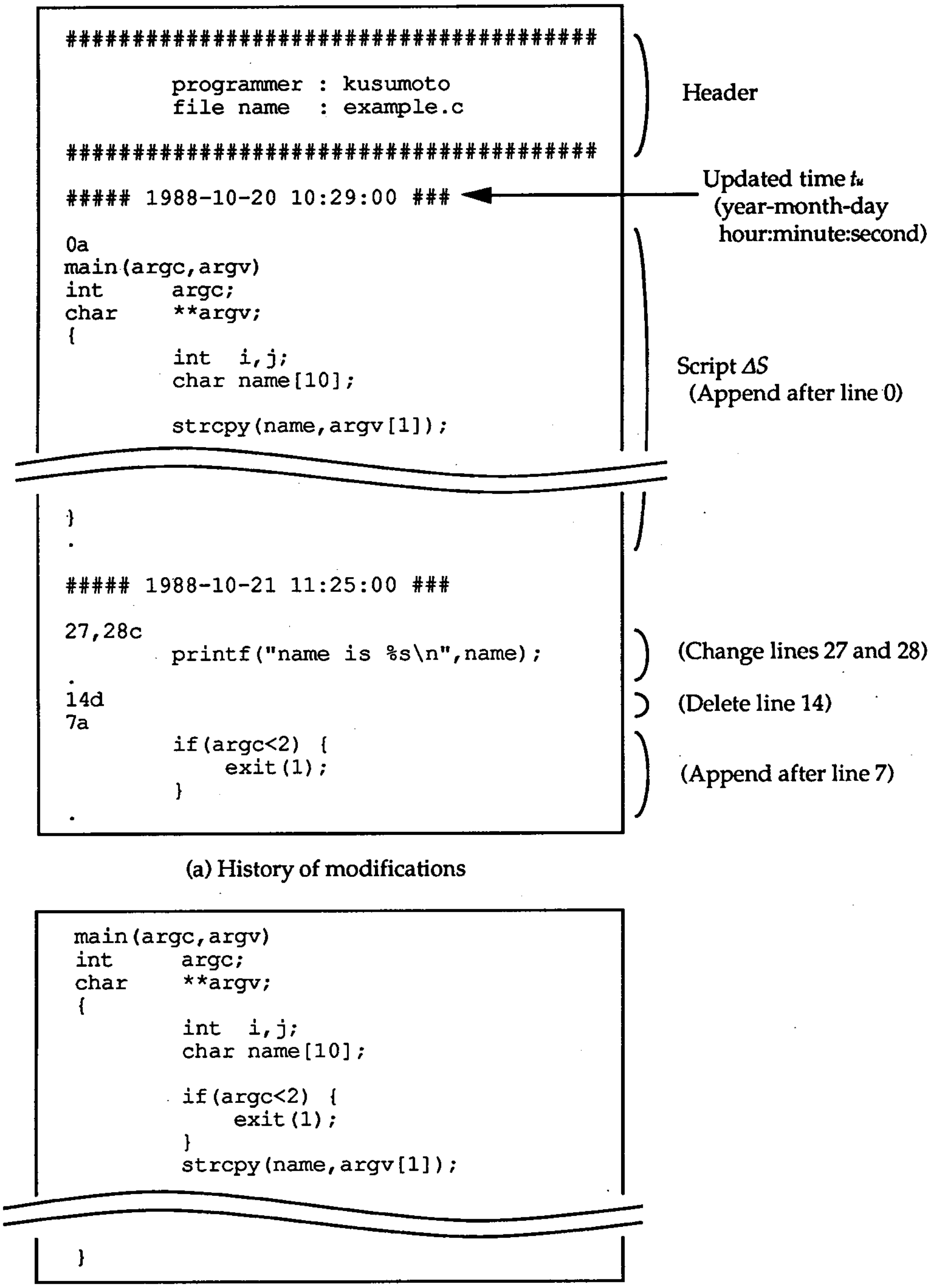


Figure 6.4 Computation of product data on UNIX

the process data and the product data from the Data Collection Unit and the analysis from the Data Analysis Unit.



```
########################################

        programmer : kusumoto
        file name  : example.c

########################################

##### 1988-10-20 10:29:00 ###

0a
main(argc,argv)
int     argc;
char    **argv;
{
        int  i,j;
        char name[10];

        strcpy(name,argv[1]);
```

```
}
.

##### 1988-10-21 11:25:00 ###

27,28c
        printf("name is %s\n",name);
.
14d
7a
        if(argc<2) {
            exit(1);
        }
.
```

(a) History of modifications

```
main(argc,argv)
int     argc;
char    **argv;
{
        int  i,j;
        char name[10];

        if(argc<2) {
            exit(1);
        }
        strcpy(name,argv[1]);
```

```
}
```

(b) Latest version

Figure 6.5 Example of product data

Table 6.1 Calculation of program modifications

Script types	**Values of measure**
n a <text> .	L_a = the number of lines in <text>
n c <text> .	$L_c = 1$ $L_{c'}$ = the number of lines in <text>
m,n c <text> .	$L_c = n - m + 1$ $L_{c'}$ = the number of lines in <text>
n d .	$L_d = 1$
m,n d .	$L_d = n - m + 1$

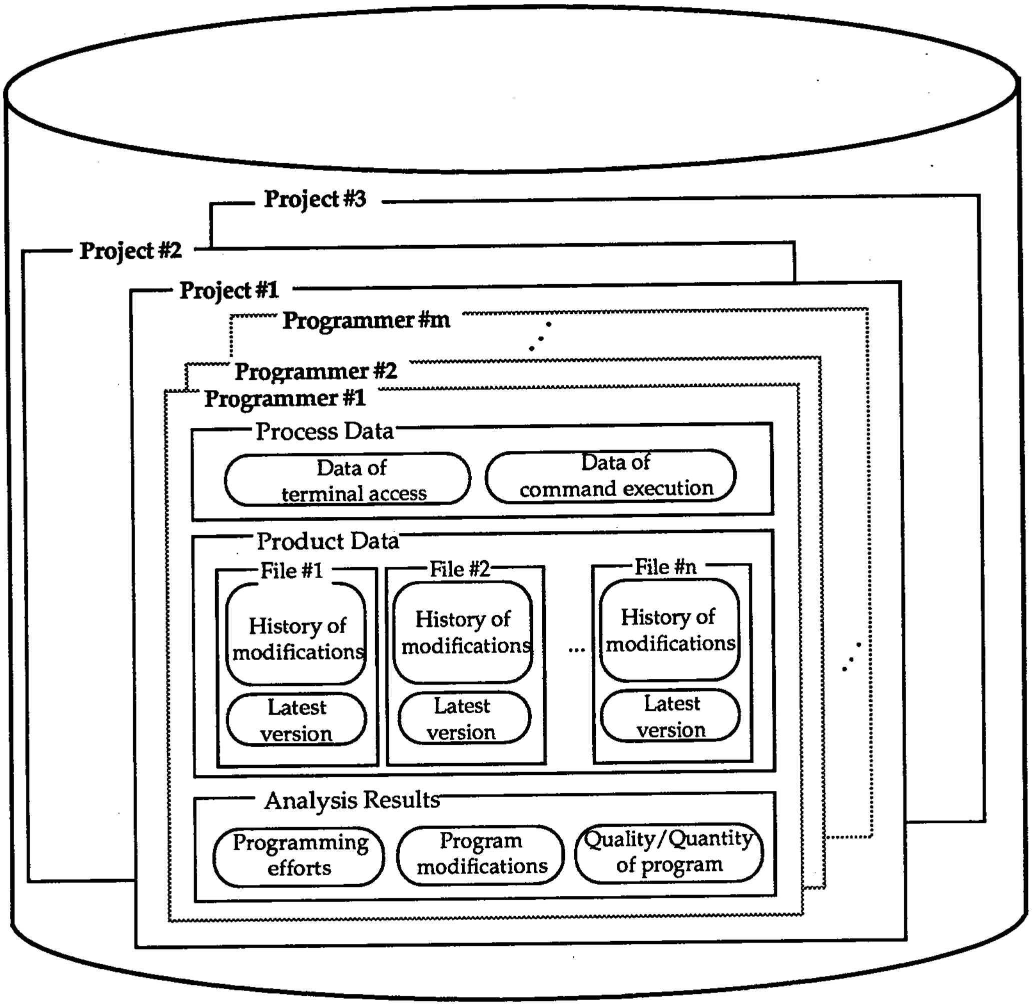


Figure 6.6 Process/Product DataBase

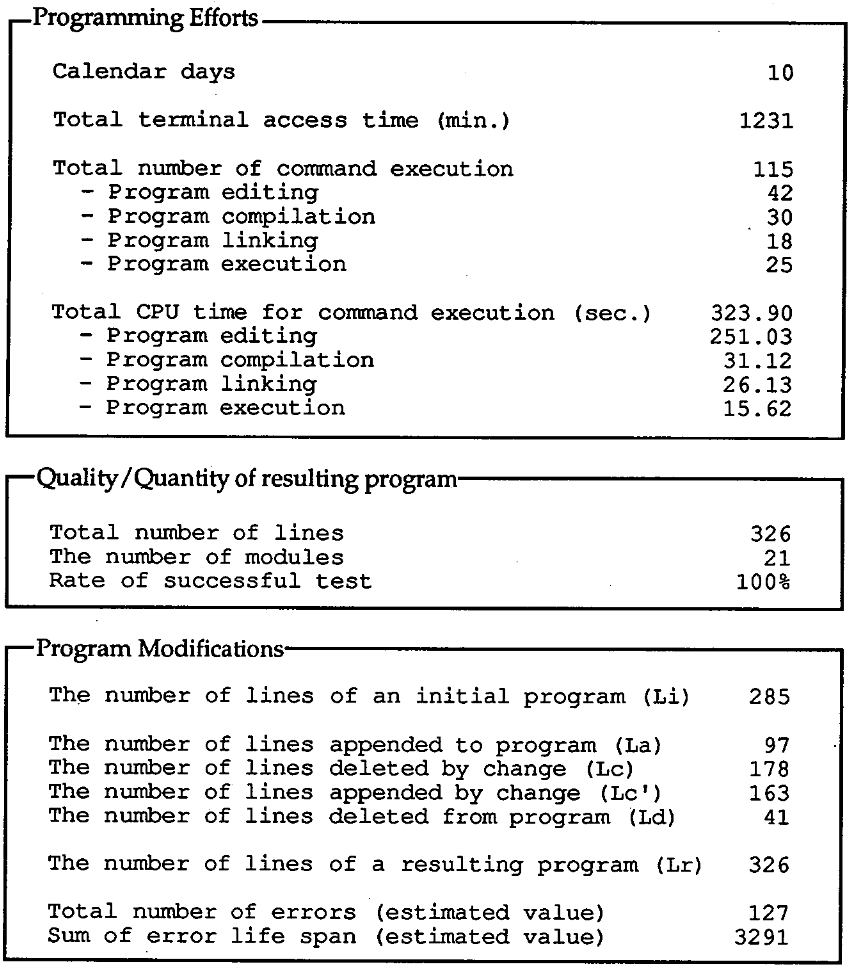
6.4.4 Analysis of programmer productivity

The analysis results in information on programming efforts, the quality/quantity of the resulting program, and program modifications. An example analysis is shown in Figure 6.7.

For programming efforts, the following four kinds of measures are currently calculated: (1) calendar days, (2) total terminal access time (in minutes), (3) total number of command executions, and (4) total CPU time (in 1/100 seconds) for command execution. The latter two are further classified four ways, depending on the type of command: program editing, compilation, linking, and execution. These values are calculated based on the data for terminal access and command execution as taken from the process data (see Figure 6.2).

The quality/quantity of the resulting program is calculated from the latest version F_l of the file in product data (see Figure 6.5 (b)). Currently, for measuring quantity, two concrete measures are adopted: the total number of lines and the total number of modules (see Figure 6.7). As for quality, only the ratio of successful tests to the total number of test cases is evaluated.

An outline of an automatic test for evaluating quality is shown in Figure 6.8. In Figure 6.8, the program to be tested is taken from the product data in the Process/Product Data Base. For each test case, a pair, consisting of an input to the program and the correct output, must be prepared beforehand by the data analyst. The program output, obtained by program execution, is compared with the correct output. If there are differences between them, the test case is considered to be unsuccessful. The results of the comparison are summarized as a test report. Figure 6.9 shows an example of a test report, in which test1 is unsuccessful and



Programming Efforts

```
Calendar days                                        10

Total terminal access time (min.)                  1231

Total number of command execution                   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 an 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 a resulting program (Lr)     326

Total number of errors (estimated value)            127
Sum of error life span (estimated value)           3291
```

Figure 6.7 Example analysis

test2 is successful. In order to increase the value of the ratio, the test report is reported back to the programmer through Feedback Management (CM11), as shown in Figure 6.8.

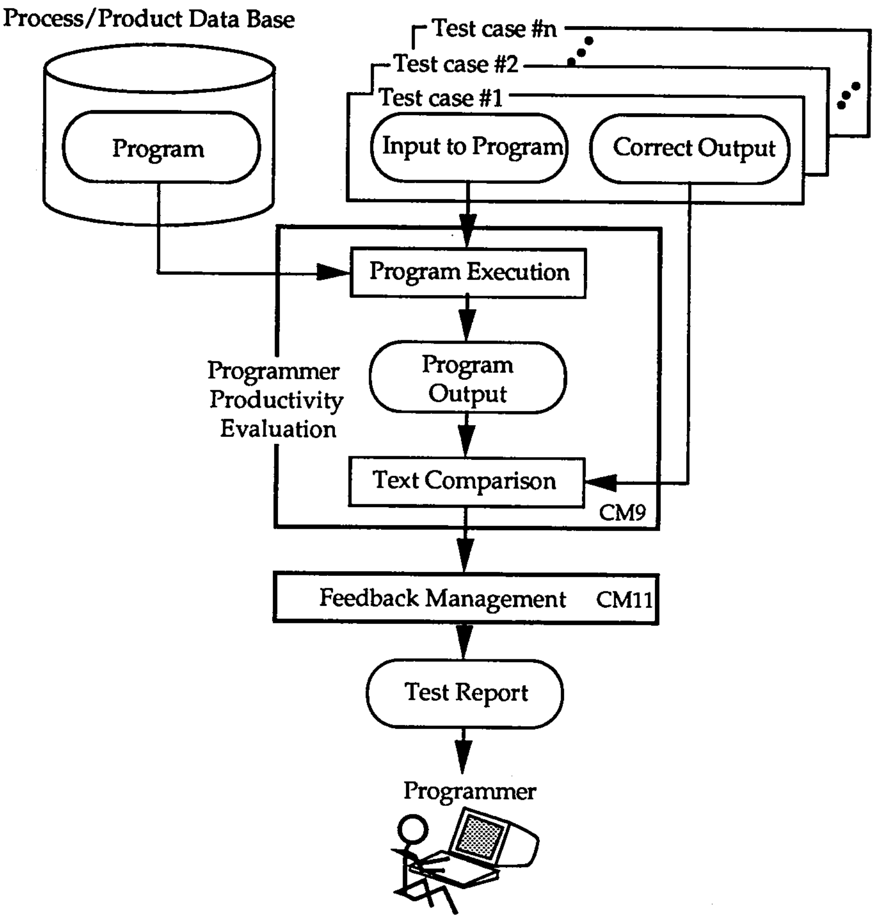
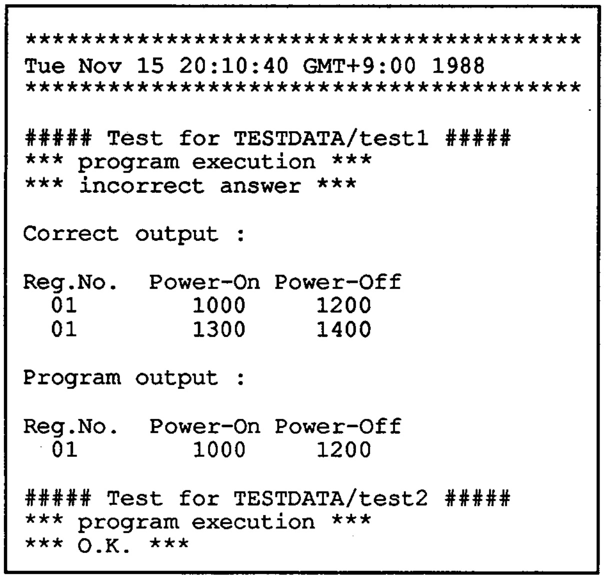


Figure 6.8 Automatic testing and reporting

Finally, program modifications are calculated based on the history of modifications in the product data. For basic measures of program modifications, the following six *L*'s are currently adopted according to the type of modification (see Figure 6.10):

L_i : the number of lines of an initial program

L_a : the number of lines which are appended to the program



```
************************************
Tue Nov 15 20:10:40 GMT+9:00 1988
************************************

##### Test for TESTDATA/test1 #####
*** program execution ***
*** incorrect answer ***

Correct output :

Reg.No.  Power-On Power-Off
  01        1000      1200
  01        1300      1400

Program output :

Reg.No.  Power-On Power-Off
  01        1000      1200

##### Test for TESTDATA/test2 #####
*** program execution ***
*** O.K. ***
```

Figure 6.9 Example of test report

L_c : the number of lines which are the subject of change (that is, lines deleted by change)

$L_{c'}$: the number of new lines which are inserted by change (that is, lines appended by change)

L_d : the number of lines which are deleted from the program

L_r : the number of lines of a resulting program

Figure 6.10 shows a typical case of program modifications. The following relation is clearly derived.

$$L_r = L_i + (L_a + L_{c'}) - (L_c + L_d)$$

The value of L_i is obtained from the history of modifications, and L_r is also easily calculated from the latest version of the file in the product data. The values of L_a, L_c, $L_{c'}$ and L_d can be calculated based on the script in the history of modifications in the product data. (The relation between the types of scripts and the values of L_a, L_c, $L_{c'}$ and L_d is shown in Table 6.2.)

Additionally, the number of errors removed during the program development process is estimated (see Figure 6.7). The sum of the error life span [Matsumoto et al. 1987], which has been introduced as a measure of programmer productivity, is also estimated. As mentioned in Section 4.4, the number of errors and the error life span can be calculated automatically based on program modifications.

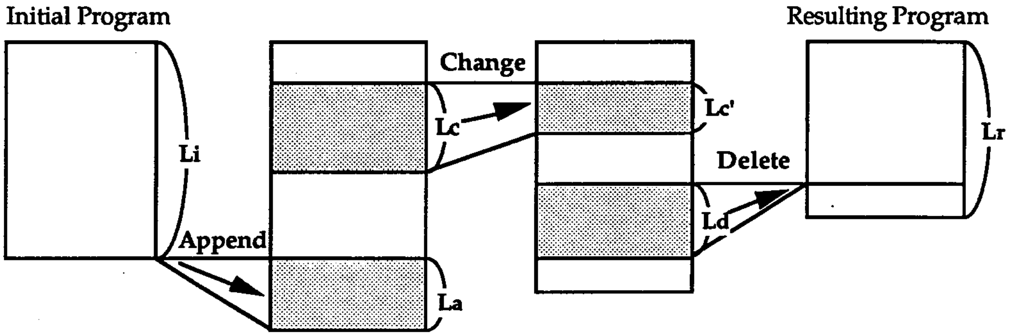


Figure 6.10 Basic measures for program modifications

Table 6.2 Illustration of script

Script types	Function in editor "ed"
n a <text> .	"a" stands for append command, and "*n*" is a non negative integer. This script implies to read <text> and append it after the *n*-th line.
n c <text> .	"c" stands for change command, and "*n*" is a nonnegative integer. This script implies to delete the *n*-th line, then accept <text> which replaces the line.
m,n c <text> .	"c" stands for change command, and "*m*", "*n*" , are nonnegative integers. This script implies to delete the lines between the *m*-th line and the *n*-th line, then accept <text> which replaces these lines.
n d .	"d" stands for delete command, and "*n*" is a nonnegative integer. This script implies to delete the *n*-th line.
m,n d .	"d" stands for delete command, "*m*", "*n*" are nonnegative integers. This script implies to delete the lines between the *m*-th line and the *n*-th line.

Chapter 7: Experimental Evaluation of the Environment

7.1 Objective of the experiments

As mentioned in Section 5.1, data collection and information feedback are essential activities to control the software development project. Therefore, in order to show the usefulness of the measurement environment in the prototype system GINGER, we have to observe and evaluate the activities needed for both data collection and information feedback.

During the experiments reported in Chapter 4, we observed that the automatic data collection was unobtrusive. In addition, it is also observed that the data collection activities of the prototype system may increase the load of the computer system slightly. However, as for the information feedback of the measurement environment, the usefulness has not been shown yet. Therefore, to show the usefulness of information feedback in the prototype system, an experimental project described below has been carried out. The experiments are for undergraduate students in the Department of Information and Computer Sciences of the Osaka University [Matsumoto et al. 1988b].

7.2 Outline of the experiments

The experimental project includes two kinds of experiments: Experiment 4, in which no feedback is given during software development and Experiment 5, in which information feedback is given by the prototype system. The results of

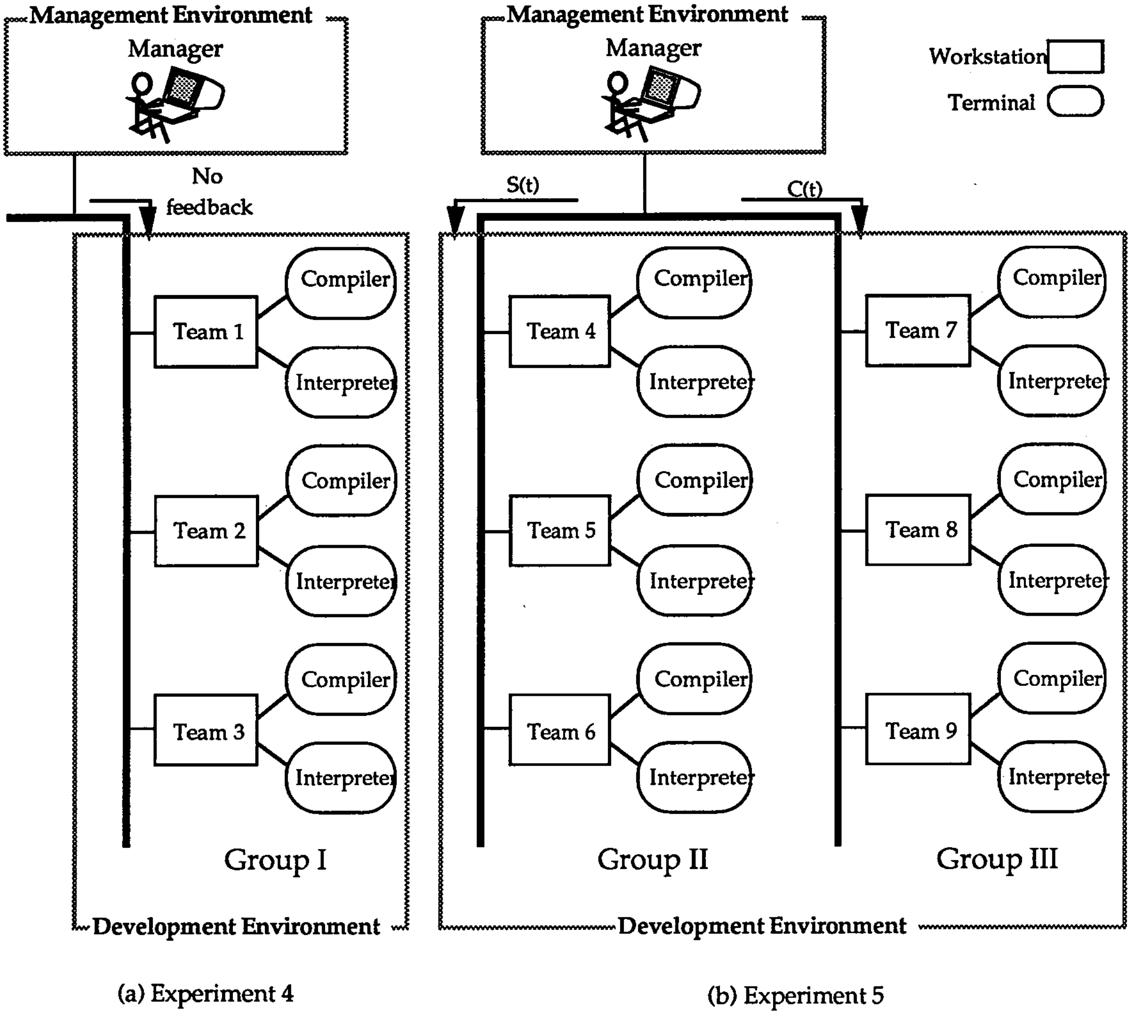


Figure 7.1 Outline of the experiments

these two experiments are compared to show the effectiveness of information feedback.

The main characteristics of both Experiments 4 and 5 are summarized as follows (see Figure 7.1):

(1) Eighteen programmers, who were undergraduate students in the Department of Information and Computer Sciences of the Osaka University, were divided into nine teams, each consisting of exactly two programmers. The nine teams were further divided into three groups (called Groups I, II and III), each consisting of three teams.

(2) Each team developed programs for the same system using the C language. The system consisted of two programs: a compiler, which translates a Pascal program into an intermediate language, and an interpreter, which translates and executes each statement of the intermediate language. The resulting systems (compiler and interpreter) contained about 2000 lines.

(3) One programmer of each team developed a program for the compiler; the other developed a program for the interpreter. Each team had for their use one workstation with two terminals.

(4) All programs were tested by automatic testing, as shown in Figure 6.8, where 30 test cases were provided. The successful test rate had to be 100% for each program.

7.3 Information feedback

In Experiment 4, the three teams in Group I developed programs for the compiler and the interpreter. For Group I, the prototype system GINGER monitored the activities of the programmers and collected some data, but no feedback was given to the programmers.

The following two kinds of data were collected in Experiment 4 (to be used as the feedback information in Experiment 5):

(1) Information on the quantity of the resulting program

S(t): the number of lines in the resulting program at time *t*.

S: the number of lines in the resulting program when the final program has been developed.

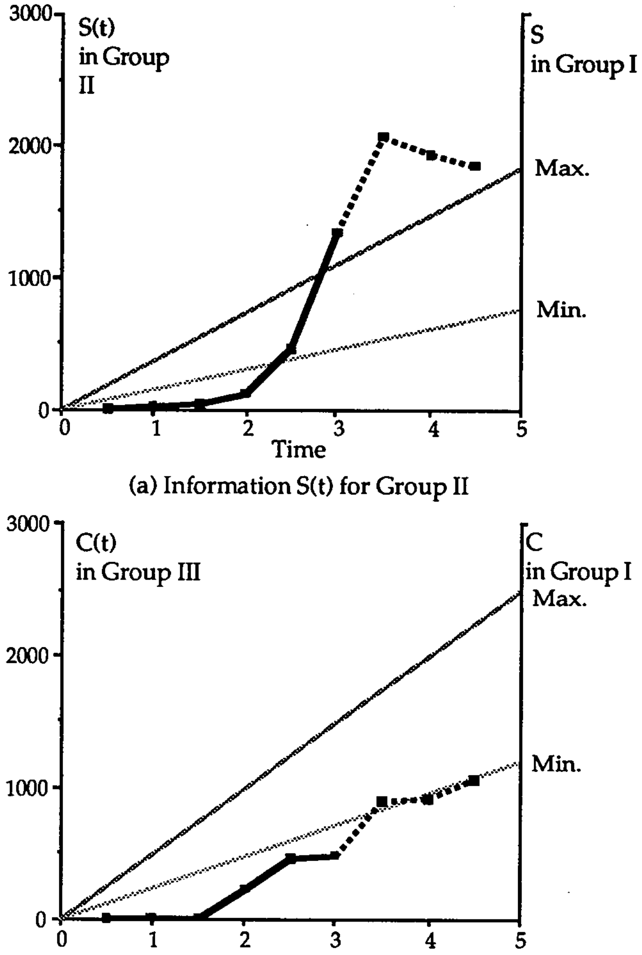
(2) Information on program modifications

C(t): the number of lines which have been changed or deleted up to time *t*.

C: the number of lines which have been changed or deleted when the final program has been developed.

Experiment 5 was executed about 2 months after Experiment 4 was over. In Experiment 5, as mentioned before, the prototype system GINGER monitored each team in Group II and Group III and collected data *S(t)*, *S*, *C(t)* and *C*. Additionally, the system gave feedback information to each team as follows. Twice a week, the programmers of Group II were provided with their own *S(t)* and the maximum and minimum values of *S* for Group I (see Figure 7.2(a)). The programmers of Group III were provided their own *C(t)* and the maximum and minimum values of C in Group I (see Figure 7.2(b)).

In addition, the programmers in Group II were instructed to develop programs as fast as possible with reference to the maximum and minimum values of *S* for Group I. In other words, the value *S(t)* should exceed the estimated value (according to the minimum value of *S*) for Group I. Similarly, the programmers in Group III were instructed to develop programs cautiously, with reference to the maximum and minimum values of C for Group I. That is, the value *C(t)* should not go beyond the estimated value (according to the maximum value of C) for Group I.



(a) Information S(t) for Group II

(b) Information C(t) for GROUP III

Figure 7.2 Example of feedback information

(The current time is 3 (weeks), and the dotted line represen the result obtained at 4.5 (weeks). The lines between origi and Max. (Min.) represent an estimation.)

7.4 Result and interpretation

The experimental data are summarized in Table 7.1, which shows total terminal access time *T*, total *S*, and total *C* for each team. In these experiments, Groups I, II and III were organized by the instructor to minimize the differences among group performances. It is assumed that the productivity of a programmer (and team) can be evaluated by total terminal access time *T*, which is one of the measures of programming effort. The justification for this assumption is explained as follows. (1) Each team developed the same system. (Each team was given the same specification of the system to be developed.) Thus, the size of the resulting program did not affect the productivity. (2) All programs were tested, and the successful test rate was 100% for the common test cases. Thus, the qualities of all program are the same.

Under these conditions, we find that the greater the programming effort, the lower the productivity. Thus, the reciprocal value of total terminal access time *T* (1/*T*) is considered to represent the productivity of the team (and programmers) in these experiments.

To compare among the three groups, the averages of *T*, *S* and *C* for each group are calculated and summarized in Table 7.2. From Table 7.2, the following relation (7.1) is derived with respect to the averages of productivity:

$$\text{Group I} < \text{Group II} \cong \text{Group III} \qquad \ldots (7.1)$$

Namely, the productivity of Group II and Group III, in which each programmer had been given as feedback the information on his or her own program development and the results of prior project, are higher than the productivity in

Table 7.1 Experimental data

Group	Team	Total terminal access time T (min.)	Total number of lines in resulting program S (LOC)	Total number of lines changed or deleted C (LOC)
I	1	5907	1796	3052
	2	8955	1900	4407
	3	7930	2826	1933
II	4	6760	2551	1808
	5	6054	2771	4652
	6	1696	2300	3431
III	7	8343	1943	2294
	8	3879	1957	2176
	9	2555	1538	1573

Group I, in which no information had been given as feedback. The following conclusions may be reached:

First, it is very important for programmers to have a clear goal (or target) during program development. The programmers in Group II could develop the programs fast by comparing their own activities to the prior project, since they had been given the information on the prior project as feedback.

Second, in preliminary experiments [Matsumoto et al. 1988b], it was found that there is a high correlation between the value of C and programmer productivity, which is the ratio of the number of lines in the resulting program to the total amount of terminal access time. The programmers in Group III

Table 7.2 Comparative results among three groups

Group	Average of T (min.)	Average of S (LOC)	Average of C (LOC)
I	7598	2174	3131
II	4837	2541	3297
III	4926	1813	2014

could develop the programs cautiously and efficiently by comparing their own activities to the prior project.

There is no difference between Group II and III with respect to productivity, from the relation (7.1). But, the following relation (7.2) is derived from Table 7.2, with respect to the number of lines in the final program.

Group III < Group I < Group II ... (7.2)

In addition, the following relation (7.3) is derived from Table 7.2, with respect to the number of lines which have been changed or deleted.

Group III < Group I ≡ Group II ... (7.3)

As a result, it is observed that the programmers who are given information on the quantity of the resulting program tend to develop larger programs than the programmers who are not given information. It is also observed that the programmers who are given information on program modifications tend to develop smaller programs, more cautiously than the programmers who are given no information.

7.5 Possible applications

The general goal of a data collection and analysis system for improving the productivity of software development is to provide feedback of important information, not only to programmers but to analysts/designers as well. Unfortunately, the data in the system analysis and designing phases cannot be automatically collected since the documents in these two phases are not computerized yet. Thus in the current state of GINGER, only some information can be given as feedback to programmers during the implementation phase (see Section 6.4).

The effective feedback of information to designers is clearly an interesting and important task. The values of strength/cohesiveness and coupling of modules are typical examples of such information. But it is also well known that they are not measurable at the design phase since each value depends on the source code. In the future, if certain data can be collected from the design process and stored in the database, the architecture of GINGER can be extended to measure and analyze this phase as well.

The current version of the GINGER system has been applied to improve software development activities in the academic field according to two different views.

(1) Teacher/Instructor

The teacher/instructor would/should monitor the programming progress of students directly from the coded source program rather than from off-line student reports. With respect to coded source programs, analyzed data such as the size and the number of modifications in the source code can be obtained [Matsumoto et al. 1988b].

(2) Students

Students can also monitor not only his or her own data, but also the statistical data, such as the average number of modifications, in certain specific experiments [Matsumoto et al. 1988b], of classmates and the previous students from different years.

From these two views, the following concrete applications can be considered;

- Teachers can use monitored data to instruct students in the technology/skill of coding/testing [Matsumoto et al. 1988b].
- Off-line reports submitted by students can be validated [Matsumoto et al. 1988b].
- Causes of bugs may be analyzed statistically based on the debugging/testing progress [Matsumoto et al. 1988d].
- The programming ability of each student or of a team may be evaluated, assuming ability is strongly related to the number of modifications made during programming [Kusumoto et al. 1989].
- If the programming ability of individual students is evaluated, the teams or groups can be organized as desired [Kusumoto et al. 1989].
- Residual bug counts [Matsumoto et al. 1988a] can be estimated according to the number of modifications.

Wider applications outside of the classroom are expected in the future, since there may be many potential users. Feedback characterizes process/product problems so that they can be improved. Thus we can apply the GINGER system from different views to other roles: (1) managers, (2) developers, (3) quality assurance personnel, (4) customers and (5) researchers. Most of the issues may turn out to be similar to those of the classroom, with some exceptions, such as the estimation of the cost of the product.

Chapter 8: Conclusion

8.1 Summary of major results

In this thesis, a new programmer performance model and team performance model are proposed for evaluating the activities of the individual programmer and the programmers of team in a quantitative and objective way. These two models are defined based on the novel concept of error life span.

The life span of an error is defined as the time duration from when the error manifests itself in the software to when the error is removed from the software. It can be considered to represent the degree of effect of the error on the productivity and quality of the software development. Results of experimental evaluation show that the sum of the error life span has higher correlation with the total terminal access time than does the total number of errors. Thus, error life span is concluded to be one good metric for evaluating the activities of programmers. In addition, in order to reduce the effort of getting the values of the metrics, an automatic estimation method for the sum of error life span is devised. The method has been already used extensively in several experimental projects and the validity and effectiveness of the proposed method has been shown.

Based on the concept of error life span, the programmer performance model is defined. In defining of the model, the sum of the error life span is used as a good indicator that closely links the performance of the programmer with (1) the number of errors made in the software development process, and (2) the rate of detection and removal of these errors. Additionally, in order to extend the application fields of the programmer performance model, the programmer performance model is defined to include a normalizing function for the complexity of given problem. Results of experimental evaluations show that the programmer performance model has high correlation with the "aptitude" of the student programmer. Of course, we think that the programmer performance model does not indicate the complete extent of the performance of the programmer, but it shows and evaluates, in a quantitative and objective way, the effect of one aspect of a programmer's activities on software development.

The programmer performance model is extended to three kinds of team performance models (*M1, M2,* and *M3*). Results of experimental evaluation suggest that the model *M1* (the total of performance of programmer in team) and the model *M2* (the average of performance of programmer in team) are not good indicators for evaluating the performance of the team. The model *M3* (defined by regarding a team as a virtual programmer) has the highest correlation with the team debugging effort time (that is, the time spent for the most important team activities on software development). In addition, we show that the team performance becomes maximum only if each programmer on the team develops program modules with a size that is proportional to their performance. Thus, the team performance model *M3* can be used for not only the evaluation of the activities of the team, but also the construction of team organization before the project starts.

This thesis also describes a measurement environment GINGER, which automatically collects and analyzes the data from the activities of programmers during software development and shows the obtained and analyzed data to the programmers as feedback information to control the software development project in an objective way. In the proposed environment, special emphasis is given to programmer productivity in evaluating the activities of programmers. Additionally, to analyze the activities of programmers in detail and to improve programmer productivity by using the results of analysis, the concept of program modification is introduced as a metric to estimate the activities of programmers.

GINGER consists of a software development environment and a management environment and it presents a mechanism for supporting interactive information processing between these two environments. Data on programmer activities are collected from the development environment and sent to a manager in the management environment. In the management environment, the collected data is analyzed with respect to the productivity and quality of the software and then given as feedback to the developers in the development environment. This two-way information flow is essential in controlling a software development project.

A prototype of GINGER has been developed in a UNIX environment and has already been used in several empirical studies. An experimental evaluation of information feedback from manager to programmers is described in Chapter 7. Results of the experiments show that information feedback can give the programmers a clear goal or target with respect to productivity and quality of software during its development process and thus help programmers accomplish their goal effectively. The proposed measurement environment provides some primitive functions to measure and control the software development process and product, as well as the evaluation of the programmer productivity.

8.2 Future work

Currently in the proposed environment, the manager takes a leading role in data collection and information feedback. Thus, these processes involving are done from a manager's viewpoint. As a result, we conclude that such environments can control the software development project, but possibly cannot improve significantly the productivity and quality of software.

There exist two problems in the current measurement environment for improving the productivity and quality of software, especially in a large-scale project. The essential difficulties in these problems is that the main objective and concern of the manager is the success of the ongoing project.

The first problem is that data, which may contribute to the improvement of the productivity and quality of software in a future project, are not collected and analyzed by a manager. For example, consider an error analysis (one of the most well-known approaches to improving programmer performance). If we have the data on the error distribution, tendencies of errors made by programmers, and so on, we can suggest to each programmer the weak points in their coding activity and thus improve his or her programmer performance.

However, in general, error analysis is done at the end of the coding phase, after the project completed. Thus it is hard to effectively utilize the results of error analysis for an ongoing project. Therefore, a manager will neither do error analysis with interest nor collect data, even if it may yield higher productivity and software quality for a future project.

The second problem is that almost all data and analysis results are discarded by a manager. However, these data could be useful to future projects. For example, in our experiments described in Chapter 7, the data of program size and program modifications collected in Experiment 4 were used to present programmers a clear goal in Experiment 5. If we should have the data collected from both the successful and unsuccessful projects, projects similar to an ongoing project, then we could present a more comprehensible goal to developers of the ongoing project.

Since collecting and storing the data on a project does not always contribute to the success of the project, most managers do not expend the effort to do it even after the end of the project. Of course, a manager may gain experience in managing a software development project, but it is very difficult for other managers to reuse this experience. Thus arises the need to establish a systematic methodology for managing software development.

The difficulties in the problems described above come from having data collection and information feedback done only by the manager, especially considering that the only major objective of the manager is the success of the ongoing project. In order to resolve these difficulties, we should introduce a new "analyst" the measurement environment.

The analyst's objective is to improve the productivity and quality of the software in ongoing and future projects. In other words, the analyst tries to achieve high productivity and quality of software at an organization level rather

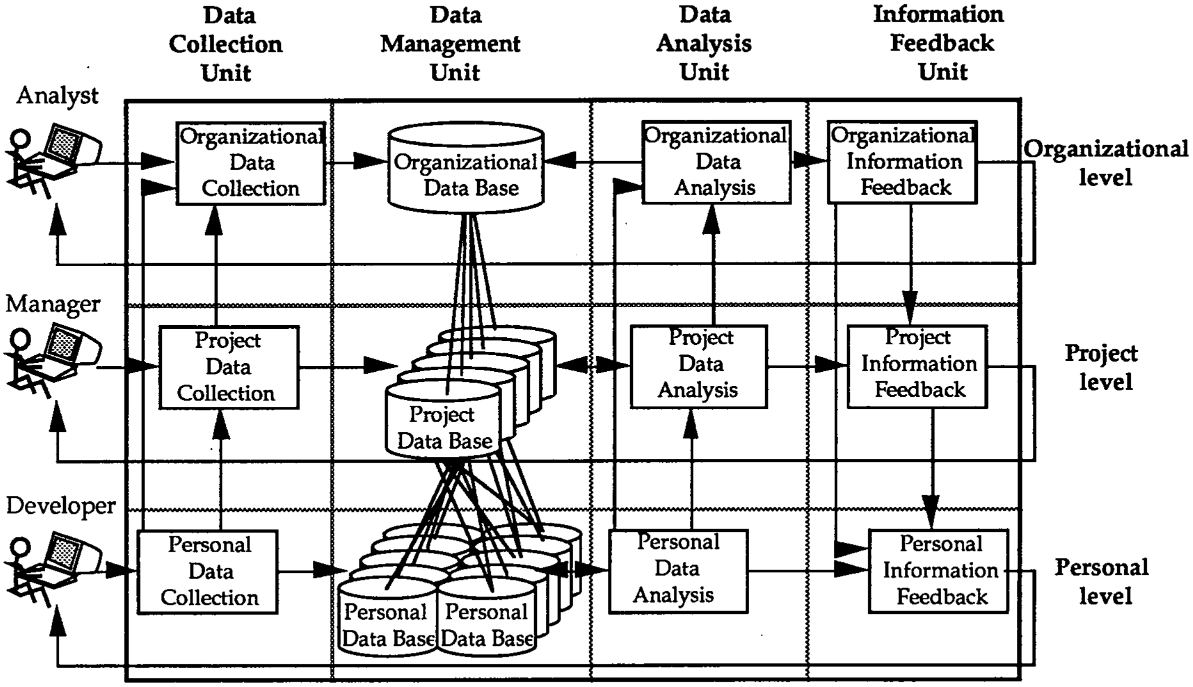


Figure 8.1 Conceptional drawing of a future measurement environment

than at an individual project level. Figure 8.1 shows a conceptional drawing of a future measurement environment. In this environment, the developer, manager, and analyst each bear the responsibility for improving the productivity and quality of the software at three different levels: personal, project, and organizational levels, respectively. Four logical units: Data Collection, Data Management, Data Analysis, and Information Feedback are also partitioned into three parts and three feedback loops are implemented according to these three levels.

In the environment in Figure 8.1, the manager does not have to collect and manage a large amount of data from an ongoing project. Further, the manager can use the data from previous, similar projects (even if he or she did not directly manage these projects). Finally, the amount of feedback provided to the developer increases and is expanded by analysis of several other projects as well as by information from the manager.

References

[Brooks 1975]
F. P. Brooks, Jr., "The Mythical Man-Month: Essays on Software Engineering," Addison-Wesley Pub., 1975.

[Baker 1972]
F. T. Baker, "Chief programmer team management of production programming," *IBM Systems Journal,* Vol.11, No.1, pp.56-73, 1972.

[Basili 1980]
V. R. Basili (Ed.), "Tutorial on models and metrics for software management and engineering," IEEE Computer Society Press, 1980.

[Basili & Reiter 1979]
V. R. Basili and R. W. Reiter, Jr., "An investigation of human factors in software development," *IEEE Computer,* Vol.12, No.12, pp.21-38, Dec. 1979.

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

[Basili & Rombach 1987]
V. R. Basili and H. D. Rombach, "Tailoring the Software Process to Project Goals and Environments," *Proceedings of the 9th International Conference of Software Engineering,* pp.345-357, March 1987.

[Boehm 1981]
B. W. Boehm, "Software Engineering Economics," Prentice-Hall, 1981.

[Chen 1978]

E. T. Chen, "Program complexity and programmer productivity," *IEEE Transactions on Software Engineering*, Vol.SE-4, No.3, pp.187-194, May 1978.

[Conte et al. 1986]

S. D. Conte, H.E. Dunsmore and V.Y. Shen, "Software Engineering Metrics and Models," The Benjamin/Cummings Pub., 1986.

[Curtis 1985]

B. Curtis (Ed.), "Tutorial : Human factors in software development," IEEE Computer Society Press, 1985.

[DeMarco 1982]

T. DeMarco, "Controlling Projects: Management, and Estimation," Yourdon Press, 1982.

[Druffel et al. 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, Nov. 1983.

[Dunham & Krusei 1983]

J. R. Dunham and E. Krusei, "The measurement task area", *IEEE Computer*, Vol.16, No.11, pp.47-54, 1983.

[Dunsmore & Gannon 1980]

H. E. Dunsmore and J. D. Gannon, "Analysis of the effects of programming factors on programming effort," *Journal of Systems and Software*, Vol.1, No.2, pp.141-153, 1980.

[IEEE 1983]

"IEEE Standard Glossary of Software Engineering Terminology", IEEE, Rep. IEEE-Std-729-1983, 1983.

[Kikuno et al. 1990]

T. Kikuno, K. Matsumoto and K. Torii, "Advanced Technologies for software reliability —Survey and future trends—," *Journal of IEICE Japan,* Vol.73, No.5, pp.454-460, May 1990 (in Japanese).

[Kusumoto et al. 1989]
S. Kusumoto, K. Matsumoto, T. Kikuno and K. Torii, "An experimental evaluation of relationship between faults in functional design and number of module modifications," *National Convention Record of IEICE Japan,* D-154, Sep. 1989 (in Japanese).

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

[Lamb 1988]
D. A. Lamb, "Software Engineering: Planning for Change," Prentice Hall, 1988.

[Mackenzie 1969]
R. A. Mackenzie, "The management process in 3-D," *Harvard Business Review,* Vol.47, No.6, pp.80-87, Nov.-Dec. 1969.

[Matsumoto et al. 1988a]
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, June 1988.

[Matsumoto et al. 1988b]
K. Matsumoto, K. Inoue, T. Kikuno and K. Torii, "Analysis of programming efforts based on program changes," *National Convention Record of IEICE Japan,* D-176, Sept. 1988 (in Japanese).

[Matsumoto et al. 1988c]

K. Matsumoto, K. Inoue, T. Kikuno and K. Torii, "An experimental evaluation of programmer performance based on error life span —For program development in academic environment—," *Transactions of IEICE Japan,* Vol.J71-D, No.10, pp.1959-1965, Oct. 1988 (in Japanese).

[Matsumoto et al. 1987]
K. Matsumoto, K. Inoue, H. Kudo, Y. Sugiyama and K. Torii, "Error life span and programmer performance," *Proceedings of the 11th International Computer Software and Applications Conference,* pp.259-265, Oct. 1987.

[Matsumoto et al. 1990]
K. Matsumoto, T. Kikuno and K. Torii, "An experimental evaluation of S-shaped software reliability growth models in academic environment —Comparison between models and determination of inflection rate—," *Transactions of IEICE Japan,* Vol.J73-D-I, No.2, pp.175-182, Feb. 1990 (in Japanese).

[Matsumoto et al. 1986]
K. Matsumoto, S. Onishi, K. Inoue, H. Kudo, Y. Sugiyama and K.Torii, "A Note on a Programmer's Capability and Its Collecting Tools," *Papers of Technical Group of IPS of Japan,* No.SE50-6, 1986 (in Japanese).

[Mills 1976]
H. Mills, "Software development," *IEEE Transactions on Software Engineering,* Vol.SE-2, No.4, pp.265-273, Dec. 1976.

[Moher & Schneider 1981]
T. Moher and G. M. Schneider, "Methods for improving controlled experimentation in software engineering", *Proceedings of the 5th International Conference of Software Engineering,* pp.224-233, March 1981.

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

[Myers 1976]
G. J. Myers, "Software Reliability —Principles and practices," John Wisley & Sons, Inc., 1976.

[Onishi et al. 1986]
S. Onishi, K. Matsumoto, Y. Sugiyama and K. Torii, "Metrics environment on the software development", *33rd National Convention Record of IPS of Japan*, 1G-1, pp.693-694, 1986 (in Japanese).

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

[Scott & Simmons 1975]
R. F. Scott and D. B. Simmons, "Predicting programming group productivity - A communications model", *IEEE Transactions on Software Engineering*, Vol.SE-1, No.4, pp.411-414, 1975.

[Thayer 1988]
R. H. Thayer (Ed.), "Tutorial : Software engineering project management," IEEE Computer Society Press, 1988.

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

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

[Walston & Felix 1977]

C. E. Walston and C. P. Felix, "A method of programming measurement and estimation," *IBM Systems Journal,* Vol.16, No.1, pp.54-73, 1977.

[Weinberg 1971]
G. M. Weinberg, "The Psychology of Computer Programming," Van Nostrand Reinhold, 1971.

[Weiss 1979]
D. Weiss, "Evaluating software development by error analysis: The data from the architecture research facility," *Journal of Systems and Software,* Vol.1, No.1, pp.57-70, 1979.

[Weiss & Basili 1985]
D. M. Weiss and V. R. Basili, "Evaluating software development by analysis of changes: Some data from the Software Engineering Laboratory," *IEEE Transactions on Software Engineering,* Vol.SE-11, No.2, pp.157-168, Feb. 1985.