Translation from Transactions of IPSJ

Experimental Evaluation of Team Performance in Program Development Based on a Model —Extension of a Programmer Performance Model—

KEN-ICHI MATSUMOTO*, SHINJI KUSUMOTO*, TOHRU KIKUNO* and KOJI TORII*†

This paper proposes and compares three models (*ST1*, *ST2*, and *ST3*) for evaluating team performance in software development. These three models are defined by extending the programmer performance model (*SI*), which is defined according to a concept of error life span and has a high correlation with the "aptitude" of a student programmer. The definitions are as follows: (1) $ST1 = \Sigma SI_j$, where SI_j is the performance of programmer j in the team, (2) $ST2 = \frac{1}{n}\Sigma SI_j$, where n is the number of programmers in the team, and (3) $ST3 = SI$, where SI is the programmer performance when the team is considered as an individual programmer.

The results of experimental evaluations show that *ST3* is the most appropriate model for evaluating team performance in software development. They also show that it is possible to devise an optimal team organization strategy based on the model, in which *ST3* reaches a maximum when programmer j develops program components with a total size of $N \times \frac{SI_j}{\Sigma SI_j}$, where N is the total size of the program developed by the team.

1. Introduction

Generally, large software systems are developed by teams that consist of many analysts, designers, programmers, and so forth. In such team development, software quality and productivity depend on not only the individual performances of the programmers but also the method of team organization, the load distribution in the team, and so on [14]. Several ideas for organizing software development teams have already been proposed [11]. The chief programmer team concept, originated by Mills, is one of the best-known ideas for organizing software development teams [1].

However, since there are few model-based approaches for evaluating programmers and team activities in software development, it has been believed that team and programmer activities could not be measured absolutely. Thus, very simple but insufficient measures have been widely used in 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, the number of years that a programmer has had experience constructing similar software, and managers' intuitive evaluations have all been used.

Previously, we have proposed a model, called the "programmer performance model," for evaluating programmers' activities in software development [7, 8]. The programmer performance model *SI* is based on a novel concept of error life span.[1] The life span of an error is defined as the duration from the time at which an error manifests itself in the software as a fault or faults to the time at which this fault or these faults are removed from the software. Results of experimental evaluations show the validity of the model as a measure of the programmer performance [7, 8].

This paper proposes three new models (*ST1*, *ST2*, and *ST3*) for evaluating the activities of teams in software development. The models are defined by extending the programmer performance model *SI*. *ST1* simply summarizes the value of *SI* for all programmers in the team. *ST2* takes an average of the value of *SI* for programmers in the team. *ST3* evaluates the sum of the error life-spans in the project, on the assumption that

This is a translation of the paper that appeared originally in Japanese in Transactions of IPSJ, Vol. 31, No. 12 (1990), pp. 1812–1821.

*Department of Information and Computer Sciences, Faculty of Engineering Science, Osaka University, 1-1 Machikaneyama, Toyonaka, Osaka 560, Japan.

†Presently, with Advanced Institute of Science and Technology, Nara.

[1]In the IEEE standard [16], an error is defined as a human action that results in software containing a fault. Examples include omission or misinterpretation of user requirements in a software specification and incorrect translation or omission of a requirement in a design specification. A fault is defined as a manifestation of an error in software. A fault, if encountered, may cause a failure (a fault is synonymous with a bug).

the team is regarded as a single virtual programmer. In order to compare the validity of these three models, this paper gives some results of experimental evaluation in an industrial environment. In addition, it discusses a strategy for organizing reliable teams (to maximize the team performance) based on the team performance model *ST3*.

Section 2 introduces the concept of error life-span and presents a programmer performance model *SI* for each individual programmer along with experimental data showing the validity of the model. Section 3 describes three models *ST1*, *ST2*, and *ST3* for evaluating team performance. Section 4 gives experimental data from a project in which eight teams (of three to five programmers) solved a typical business application problem by using COBOL. Section 5 analyzes the experimental data in order to compare the three models. Section 6 discusses an optimal team organization based on *ST3*. Section 7 presents a summary of the ideas discussed in this paper, draws some conclusions, and discusses future research work.

2. Programmer Performance Model

2.1 Error Life-Span T_e

An error life-span T_e for an error e has been defined as the duration from the time at which the error e manifests itself in the software as a fault or faults to the time at which the fault or faults caused by the error e are removed from the software [9, 17]. Figure 1 shows examples of error life-span T_e. In this figure, × and ○ represent the times at which errors manifest themselves and faults are removed, respectively.

Concepts similar to the error life-span can be seen in earlier papers [10, 15]. Mills introduced the concept of "error days" for estimating the quality of an acceptable system [10]. For each fault removed, the number of "error days" is defined as the sum of the days from its creation to its detection. He mentioned that this measure is an indication of probable future errors and of the effectiveness of the design and testing processes. Weiss and Basili have used "changes" as a way of

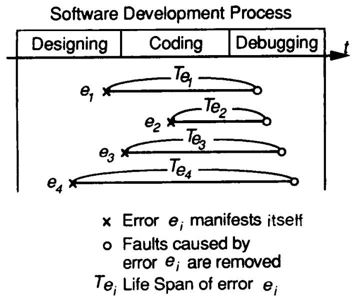
Software Development Process
Designing
Coding
Debugging
t
e_1
Te_1
e_2
Te_2
e_3
Te_3
e_4
Te_4
× Error e_i manifests itself
o Faults caused by error e_i are removed
Te_i Life Span of error e_i

Fig. 1 Error life span.

evaluating software development processes [15]. They mentioned that "the length of time each fault remained in the system" would be useful information for evaluating the software development processes. However, no collection or analysis of such data was carried out.

2.2 Programmer Performance

There have been many studies of programmer performance [3, 4, 13]. Sackman et al. showed that, for most performance variables, there are very large individual differences in programming performance [13]. In the COCOMO model, programmer capability is one of the driver attributes, and has a range of 2.03 for productivity [3]. There are, however, no model-based approaches for evaluating programmer performance.

We believe that the error life-span indicates not only the product quality, as suggested by Mills [10], but also some aspects of programmer performance. The error life-span closely is related to the performance of the programmer in the following two respects:

1. The number of errors made in the software development processes
2. The rate of detection and removal of faults caused by these errors.

For example, we consider a case in which an error causes some faults in a program text. If the life-span of the error is long, that is, if the faults have remained for a long period of time in the program text, then the programmer will have a hard time removing them. One cause of this difficulty is that the programmer will forget the details of the old code relating to the faults. The erroneous code will also affect other code appended to the program text later. Hence, we naturally think that an error with a long life-span has a strong (negative) effect on the project's progress and the reliability of the program.

2.3 Definition of *SI*

Now, the value of *SI* is defined to indicate some aspects of programmer performance. The *SI* for each individual programmer is formally defined by formula (1).

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

where f: Normalizing function
p: Complexity of given problem

This definition of *SI* is derived from the fact that a programmer who makes fewer errors and removes faults caused by these errors in a shorter time achieves a better performance. In this definition of *SI*, the following two assumptions are made:

1. Specification (of a problem) is not modified during software development.
2. Design, coding, and debugging are done by the same programmer.

2.4 Experiments to Determine Programmer Performance

In order to show the validity of the error life-span T_e and *SI*, we conducted several experiments in academic environments [7-9]. In these experiments, the square of the final program size, namely L^2, was used as a normalizing function $f(p)$. The reason we chose L^2 may be explained as follows. ΣT_e is rewritten as

$$\sum T_e = avg \times N \tag{2}$$

where *avg*: Average error life-span
N: Total number of 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

$$\frac{avg}{p} \times \frac{N}{p} \tag{3}$$

where *p*: Complexity of problem

In these experiments, there were only small differences among the specifications. We therefore thought that the complexity *p* of the problem could be estimated from the final program size *L* (the number of the lines in the final program text). As a result, we used L^2 as the normalizing function. The *SI* for each individual programmer is rewritten according to the formula (4).

$$SI = \left(\frac{\sum T_e}{L^2}\right)^{-1} \tag{4}$$

where T_e: Life span of error *e*
L: Final program size

In previous publications [7-9], we show the following three major results of these experiments with respect to the validity of the *SI* defined by formula (4).

1. ΣT_e has a high correlation with the total terminal access time.

We found that the total terminal access time, which seems to directly correspond to programmer performance, has a higher correlation with the sum of the error life-spans than with the total number of errors. The coefficient of correlation between the total terminal access time and the sum of the error life-spans was 0.82. On the other hand, the coefficient of correlation between the total terminal access time and the total number of errors was 0.45.

2. The *SI* has a high correlation with the grade point average.

Moher and Schneider have found that "experience" (as measured by the number of computer science programming courses taken) and "aptitude" (as measured by the grade point averages in computer science courses) are the major predictors of performance for student programmers [12]. We found that there was a high correlation between the *SI* and the grade point average in computer science courses for each programmer in the experiment. The coefficient of correlation between them was 0.75.

3. The *SI* was stable in two different projects.

The *SI* ranking of the programmers was almost the same in two different projects. The coefficient of correlation between the *SI*s in two projects is 0.63. Thus, the value of *SI* is relatively stable for the same programmer in different projects.

We conclude from these three observations that *SI* can be used as a metric for programmer performance in software development projects.

3. Team Performance Models

In this section, we extend the programmer performance model *SI* to three team performance models (*ST1*, *S2*, and *ST3*) that evaluate the activities of programmers in a team. These three models make different assumptions about the relationship between programmer performance and team performance. Let us consider a case in which software is developed by a team consisting of *n* programmers. In the following, let SI_j denote the *SI* of programmer $j (1 \leq j \leq n)$. For the sake of simplicity, the following notation

$$SI_j = \left(\frac{E_j}{L_j^2}\right)^{-1} \tag{5}$$

where $E_j = \Sigma T_e$,
is used instead of the notation in formula (4).

1. Team performance model *ST1*

If we could disregard both the advantages and disadvantages of team development, then team performance could be considered as the sum of the performance of the programmers in the team. *ST1* is defined as follows:

$$ST1 = \sum_{j=1}^{n} SI_j. \tag{6}$$

2. Team performance model *ST2*

If it is assumed that the performance of a programmer has a strong effect on the other programmers' performance in a team development, then team performance could be considered as the average of the performance of programmers in the team. *ST2* is defined as follows:

$$ST2 = \frac{1}{n} \sum_{j=1}^{n} SI_j. \tag{7}$$

3. Team performance model *ST3*

If it is assumed that a team functions as a virtual programmer aggregately developing software, then team performance could be evaluated by using formula (4). *ST3* is defined as follows:

$$ST3 = \left(\frac{\sum E_j}{(\sum L_j)^2}\right)^{-1}. \tag{8}$$

4. Experimental Evaluation of Team Performance Models

4.1 Outline of Experiment

In this section, we describe an experiment conducted at a training course, from April to August 1988, for new employees of a certain computer company. This experiment compared the validity and usefulness of the three team performance models proposed in Section 3. The main characteristics of the experiment may be summarized as follows:

1. Eight teams of programmers developed the same system (a file-processing program in a business application), using COBOL.
2. The system consisted of 18 program modules. (This number of program modules was specified for each team; however, the distribution of modules to members of a team was freely determined by the leader of that team.) The final program sizes were all about 2000 lines.
3. Each team consisted of three to five programmers. The teams were organized by the instructor so that the differences among their performances, in an intuitive sense, would be low.
4. Each team was assigned two terminals. Thus, the ability to access terminals was relatively limited in comparison with the experiments mentioned in subsection 2.4 [8].

4.2 Automatic Estimation of Error Life-Span

In the experiments mentioned in subsection 2.4, obtaining the error life-spans was very expensive. We traced and analyzed all the files used in the experiments by hand and kept a large amount of data concerning the processes. However, the cost of obtaining the error life-span information in a software project involving many programmers is prohibitive in practice.

To decrease the effort required to obtain error life-spans, we devised a method for automatically obtaining the estimated values of error life-spans. This method is based on the demonstration by Dunsmore and Gannon that program changes (that is, textual changes between successive versions of a program) are correlated with the total number of errors in the program [5]. Thus, each line modified during each editing session corresponds to one error to be counted. If the number of times each line has been created and deleted is known, the estimated error life-spans can be found easily.

On the basis of the results of experiments by Dunsmore and Gannon [5] and the experiments mentioned in subsection 2.4 [7, 8], the following two assumptions are made:

1. The purpose of modifying a program text during each editing session is to remove faults.
2. A set of lines created during one editing session and modified during another session corresponds to one error.

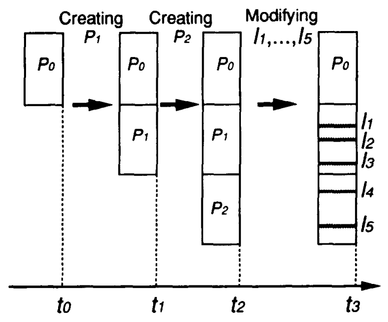

Fig. 2 Estimation of error life span.

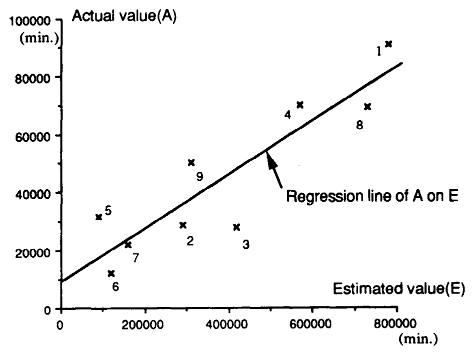

Fig. 3 Actual values vs. estimated values of ΣT_e.

Figure 2 shows an example of estimation of an error life-span. In Figure 2, $t_i(0 \leq i \leq 3)$ represents the time at which the i-th editing session terminated. P_i represents the program text created during the i-th editing session. $l_j(1 \leq j \leq 5)$ represents a line modified during the third editing session. According to the first assumption, the purpose of modifying the program text during the third editing session is to remove faults. From the second assumption, the modified lines l_j can be divided into two subsets: $\{l_1, l_2, l_3\}$ and $\{l_4, l_5\}$. Consequently, we can estimate that faults caused by two errors are removed during the third editing session, and that the life span of one error corresponding to $\{l_1, l_2, l_3\}$ is t_3-t_1, and the life span of another error corresponding to $\{l_4, l_5\}$ is t_2-t_1.

The estimated value of the sum of error life-spans is computed in this manner, using data from these experiments. Figure 3 compares the sum of the estimated error life-span with the sum of the actual error life-spans. The coefficient of correlation between them is 0.90. Thus, we can conclude that the estimated value is sufficiently equivalent to the sum of actual error life-spans.

In this experiment, the number of successive times each programmer accessed a terminal was counted and used as the time unit for evaluating T_e. In addition, each programmer had to fill out a form declaring the in-

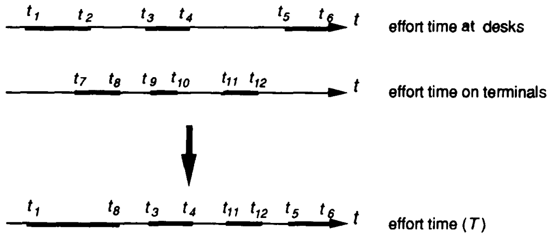
t_1 t_2 t_3 t_4 t_5 t_6 t
effort time at desks
t_7 t_8 t_9 t_{10} t_{11} t_{12} t
effort time on terminals
t_1 t_8 t_3 t_4 t_{11} t_{12} t_5 t_6 t
effort time (T)

Fig. 4 Explanation of effort time.

dividual effort time spent on designing, coding and unit debugging. Further, each team leader also had to fill out a form declaring the team effort time spent (primarily) on integration testing.

The effort time on terminals is the period during which a programmer or team works on terminals. Similarly, the effort time at desks is the period during which a programmer or team works at desks (not on terminals). The effort time at desks is reported by filling in forms. We can not rely on individual effort times, since programmers may forget to fill out forms. Thus, it appears better to use a new effort time formed by combining individual and team effort times , as shown in Fig. 4.

4.3 Experimental Data

Out of 18 modules, only 9 modules that satisfy the following two qualification conditions are selected for evaluation.

1. The average of the module size is more than 100 lines. (Modules that are too small are excluded from evaluation.)
2. The average ratio of the data division size to the module size is less than 0.5. (Programs that consists mainly of data definitions are also excluded.)

The resultant evaluation data are summarized in Table 1. The values of the sum of error life-spans are calculated in the manner described in subsection 4.2. Table 1 shows the program size (which is the number of lines in the final program text), the sum of the error life-spans, the total effort time (estimated, as shown in Fig. 4, from the effort time reported by the programmer and the terminal access time traced automatically), and *SI*. We used effort time as the time unit for the error life-span.

Table 1 Experimental data.

Team	Member	Program size	Sum of error life spans	Total effort time(min.)	*SI*
#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

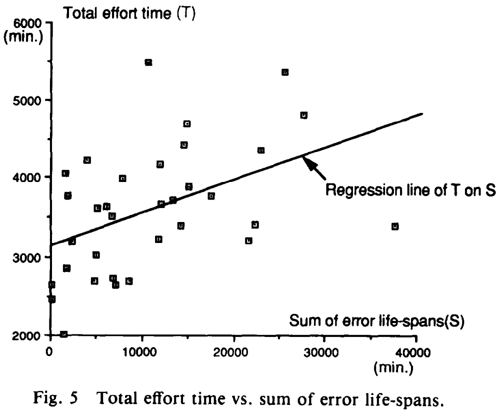

Fig. 5 Total effort time vs. sum of error life-spans.

5. Evaluation

5.1 Error Life-Span vs. Effort Time of a Programmer

Figure 5 shows the scatter plots of ΣT_e in relation to the total effort time of each programmer. (In subsection 2.4, we compared ΣT_e with the total terminal access time.) The coefficient of correlation between them is 0.46. This is not very high in comparison with the result in subsection 2.4, where we report a coefficient of correlation between them of 0.82.

The reason for this is that various factors affect the value of *SI* in the team development. For example, code review, one of the most common activities in team development, shortens the life-span of some errors. A further example involves module interfaces. Such portions of code are error-prone and difficult to detect faults in. If an error introduces faults into the interface among modules that are not assigned to the same programmer, the life-span of the error may become very long.

Cooperative work in team development also affects the effort time of each programmer. If a novice programmer can be helped by a skilled programmer in the same team, his effort time is shortened. Conversely, a team leader, who is generally is a skilled programmer, needs additional effort time to coordinate his team members or to help them to communicate.

We can conclude that it is meaningless to evaluate programmer performance in team development by using the programmer performance model *SI* in the manner described in subsection 2.4. We have to use programmer performance models and team performance models properly.

5.2 Comparison of Team Performance Models

Table 2 shows the values of *ST1*, *ST2*, and *ST3* and the team debugging effort time of eight teams (#1, #2, . . . , #8). The team debugging effort time is the total effort time for debugging after unit testing for each module has been completed. From Table 2, the following relation can be observed with respect to the values of three models:

$$ST2 < ST3 < ST1. \tag{9}$$

We have evaluated the correlations between *ST1*, *ST2*, and *ST3* (see Table 3). There are high correlations between *ST1*, *ST2*, and *ST3*, and especially between *ST1* and *ST2* (the coefficient of correlation is 0.99).

Table 3 also shows the coefficients of correlation between *ST1*, *ST2*, and *ST3* and the team debugging effort time. *ST3* has the highest correlation with the team debugging effort time (the coefficient of correlation is 0.83). Thus, it may be said that *ST3* is the most appropriate model for evaluating team performance in software development.

Table 2 Team performance and team debugging effort time.

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

Table 3 Coefficients of correlation between team performance 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

6. Team Organization Strategy Based on *ST3*

6.1 Relation between Programmer Activities and Team Performance

In this section, we discuss the strategy for team organization that maximizes the team performance based on the model *ST3*. From the definition of *ST3*, the following relation can be derived:

$$ST3 = \left(\frac{\sum_{j=1}^{n} E_j}{\left(\sum_{j=1}^{n} L_j \right)^2} \right)^{-1} \leq \sum_{j=1}^{n} \left(\frac{E_j}{L_j^2} \right)^{-1}. \tag{10}$$

If the relation

$$\frac{L_1}{SI_1} = \frac{L_2}{SI_2} = \cdots = \frac{L_n}{SI_n}. \tag{11}$$

is satisfied, then the value of *ST3* reaches a maximum. At that time, the relation

$$ST3 = \sum_{j=1}^{n} SI_j \tag{12}$$

Table 4 Relations between programmer activities and team performance.

Team	K_1	K_2	K_3	K_4	K_5	K_{opt}	$\frac{\sum_{i=1}^{n} \lvert K_i - K_{opt} \rvert}{K_{opt}}$	$\frac{ST1-ST3}{ST1}$ (%)
#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

holds. Thus, to maximize team performance in the new project P, each programmer $j(j=1, 2, \ldots, n)$ should develop program modules with a size L_j proportional to SI_j.

Table 4 shows the relation between programmer activities and team performance. K_j is the ratio of the final program size to SI_j. K_{opt} is the optimal value of K_j in the team. It is defined as follows:

$$K_{opt}=\frac{\sum_{j=1}^{n} L_j}{\sum_{j=1}^{n} SI_j} \qquad (13)$$

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} tends to increase as the number of programmers becomes smaller.

To evaluate the difference between K_j and K_{opt}, we calculate the value of the following formula:

$$D_K=\frac{\sum_{j=1}^{n} |K_j-K_{opt}|}{K_{opt}} \qquad (14)$$

In addition, to evaluate the difference between the maximum performance (*ST1*) and the actual performance (*ST3*) of each team, we calcualte the value of the following formula:

$$D_{ST}=\frac{ST1\text{-}ST3}{ST1}\times 100(\%) \qquad (15)$$

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

Consider teams #2 and #3, which give respectively the highest and the lowest values of *ST3* in Table 2. Table 4 shows that for team #2 there are large differences among K_j for each programmer and K_{opt}. As a result, the value of D_K for team #2 is relatively large. On the other hand, Table 4 shows that K_j for each programmer in team #3 is almost equal to K_{opt}. (The value of D_K for team #3 is the lowest for any of the eight teams.) Therefore, it is concluded that team #2 is superior to team #3 with respect to total team performance. On the other hand, with respect to the 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: team #2 is 67% below its optimal performance, whereas team #3 is only 5% below its optimal (maximum) performance.

6.2 Strategy for Maximizing the Value of *ST3*

As mentioned in subsection 6.1, *ST3* reaches a maximum when K_j for each programmer in the team is equal to K_{opt} defined by formula (13). Let us consider a case in which the following three assumptions are made:

1. For each programmer $j(j=1, 2, \ldots, n)$, the value of SI_j is known (for example, from SI'_j in a past project P' similar to project P).
2. The performance of a programmer may not be constant in different projects. The value of *SI* may also vary over different projects, but the variation in *SI* is proportional to its value, and the rate of variation is constant for all programmers in the team. That is, the relation $SI_j=\alpha SI'_j$ holds between the past project P' and project P.
3. For project P, there is a predictive (transformation) function F relating the problem specification S to the size of program derived from it.

In this case, we can maximize the value of *ST3* by assigning the specification $S_j(j=1, 2, \ldots, m)$ to programmer j on condition that the following relation holds:

$$\frac{F(S_1)}{SI'_1}=\frac{F(S_2)}{SI'_2}=\cdots=\frac{F(S_n)}{SI'_n}. \qquad (16)$$

7. Conclusion

In this paper, we proposed three team performance models (*ST1*, *ST2*, and *ST3*) based on the concept of error life-span and the programmer performance model. The results of experimental evaluation suggest that models *ST1* (the total performance of the programmers in a team) and *ST2* (the average performance of programmers in a team) are not good indicators for evaluating the performance of a team. Model *ST3* (defined by regarding a team as a virtual programmer) has the highest correlation with the team debugging effort time (that is, the time spent on the most important team activity in software development). In addition, we show that the team performance is maximized only if each programmer on the team develops program modules with a size that is proportional to his or her performance. Thus, the team performance model *ST3* can be used to evaluate not only the activities of the team but also the way the team is organized before the project starts.

We are currently investigating a system that automatically collects and analyzes data from the activities of programmers during software development, and that shows these data to the programmers as feedback information [7], so that the programmers can analyze and improve their activities in the manner shown by Basili and Rombach [2]. We expect that the overall productivity of development and the reliability of the resulting products will be increased by using this system. We consider that programmer and team performance models are appropriate candidates for feedback information.

References

1. BAKER, F. T. Chief programmer team management of production programming, *IBM Syst. J.*, **11**, 1 (1972), 56–73.
2. BASILI, V. R. and ROMBACH, H. D. The TAME project: Towards improvement-oriented software environments, *IEEE Trans. Softw.*

Eng., SE-14, 6 (1988), 758–773.

3. Boehm, B. W. *Software Engineering Economics*, Prentice-Hall (1981).

4. Chen, E. T. Program complexity and programmer productivity, *IEEE Trans. Softw. Eng.*, **SE-4**, 3 (1978), 187–194.

5. Dunsmore, H. E. and Gannon, J. D. Analysis of the effects of programming factors on programming effort, *Journal of Systems and Software*, **1** (1980), 141–153.

6. Gugerty, L. and Olson, G. M. Debugging by skilled and novice programmers, *Proc. of Computer and Human Interaction '86* (1986), 171–174.

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

8. Matsumoto, K., Inoue, K., Kikuno, T. and Torii, K. An experimental evaluation of programmer performance based on error life-span—For program development in an academic environment, *Trans. IEICE*, **J71-D**, 10 (1988) (in Japanese), 1959–1965.

9. Matsumoto, K., Inoue, K., Kudo, H., Sugiyama, Y. and Torii, K. Error life-span and programmer performance, *Proc. 11th International Computer Software and Applications Conference* (1987), 259–265.

10. Mills, H. Software development, *IEEE Trans. Softw. Eng.*, **SE-2**, 4 (1976), 265–273.

11. Myers, G. J. *Software Reliability—Principles and Practices*, John Wiley & Sons, Inc. (1976).

12. Moher, T. and Schneider, G. M. Methods for improving controlled experimentation in software engineering, *Proc. 5th International Conference of Software Engineering* (1981), 224–233.

13. Sackman, H., Erikson, W. J. and Grant, E. E. Exploratory experimental studies comparing online and offline programming performance, *Comm. ACM*, **11**, 1 (1968), 3–11.

14. Scott, R. F. and Simmons, D. B. Predicting programming group productivity—A communications model, *IEEE Trans. Softw. Eng.*, **SE-1**, 4 (1975), 411–414.

15. Weiss, D. M. and Basili, V. R. Evaluating software development by analysis of changes: some data from the Software Engineering Laboratory, *IEEE Trans. Softw. Eng.*, **SE-11**, 2 (1985), 157–168.

16. IEEE Standard Glossary of Software Engineering Terminology, IEEE, Rep. IEEE-Std-729-1983 (1983).

17. IEEE Standard Dictionary of Measures to Produce Reliable Software, IEEE, Rep-IEEE-Std-982.1-1988 (1988).