GINGER: A Quantitative Analysis Environment for Improving Programmer Performance

Koji TORII†, Ken-ichi MATSUMOTO†, and Shinji KUSUMOTO*

†Graduate School of Information Science
Nara Institute of Science and Technology
8916-5 Takayama, Ikoma, Nara 630-01, Japan

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

e-mail: `matumoto@is.aist-nara.ac.jp`

Abstract

This paper describes *GINGER*, a measurement-based programming support system with real-time feedback, and its use in educational settings. *GINGER* automatically collects software process and product data and performs analysis to provide feedback to programmers. The information provided by *GINGER* allows students to assess their progress in programming projects and determine how their work compares with that of other students. Instructors can also use this information to identify students who need extra attention. *GINGER* has been used in programming classes, both at Osaka University and at industrial organizations in Japan, for more than five years. An evaluation of the effectiveness of *GINGER* is presented. The use of *GINGER* to evaluate the programmer performance and the cost effectiveness of software reviews is also described.

Keywords: Software measurement, software metrics, real-time feedback, software engineering education

1 Introduction

The importance of measurement for monitoring and controlling software projects has been well established within the software engineering community [2] [16]. Metrics for software cost estimation and source code quality, for example, are now widely used in industry [13] and there are a number of research projects that focus on software metrics (see, for example, [12]).

Measurement can provide valuable information on the software process and its products. The principal benefits of measurement in software development include [6] [9]:

- *Characterizing the software organization, process and product*s – the ability to describe quantitatively the current state of software parameters, such as software quality, resources expended, and productivity.
- *Monitoring progress and providing feedback* – the ability to monitor progress, to anticipate problems, and to provide feedback to software personnel about potential problems.
- *Expressing requirements quantitatively* – the ability to express requirements quantitatively, both as goals and as acceptance criteria.
- *Analyzing costs and benefits* – the ability to quantify trade-offs that can be used by management in allocating resources.
- *Predicting project parameters* – the ability to predict software parameters, such as cost, delivery time, and reliability.

Because these benefits are primarily management-oriented, measurement is often conceived of solely as a management tool. Measurement can, however, also be used to provide feedback to developers during a project. This information can help them to assess their progress and modify their approach if needed.

Measurement is also fundamental to the educational process. Graded homework and examinations, for example, are routinely used to assess student progress. Ideally, such measurements are used to provide feedback to students to help them improve their skills and knowledge. This paper describes the use of some typical software metrics to provide feedback to students in programming classes in both academic and industrial settings. This information allows students to assess their progress in programming projects and determine how their work compares with that of other students. They can then modify their behavior or seek additional help as necessary. Instructors can also use this information to identify students who need extra attention.

The impact of the information provided by measurement is most valuable if the information is provided in a timely fashion and presented in an easy-to-understand format. To provide support for timely feedback of measurement to developers, we have developed a measurement-based programming support system with real-time feedback known as *GINGER*. *GINGER* automatically collects software process and product data and performs analysis to provide feedback to developers. Results of the analysis of data characterizing the software development process and its products are presented in graphical form whenever requested. *GINGER* has been used in programming classes, both at Osaka University and at industrial organizations in Japan, for more than five years.

This paper begins with a discussion of related work. This is followed, in Section 3, by an overview of *GINGER*. Section 4, discusses the current implementation of *GINGER* and Section 5 describes the evaluation of *GINGER*'s usefulness. Section 6 discusses two applications of *GINGER* that illustrate how the system has been used in courses at Osaka University and in industrial software organizations. Finally, Section 7 presents some conclusions on the applicability of the *GINGER* concept to software development in non-educational settings.

2 Related Work

Several systems for automatically collecting and analyzing quantitative data about the software process and its products have been described. Of these, three are closely related to *GINGER* in purpose and function.

Basili and Rombach's *TAME* (Tailoring A Measurement Environment) project at the University of Maryland is aimed at collecting data, based on the Goal/Question/Metric (*GQM*) paradigm, in order to construct models of the software process and its products [2]. Its goal is long-term improvement of the software development process. The *TAME* system will ultimately run in a distributed environment consisting of at least one mainframe computer and a number of workstations. *TAME* is primarily intended for use on large projects, as evidenced by the large storage capacity of its experience base.

Selby, et al., have specified a measurement and empirical analysis system, known as *Amadeus*, which enables a software developer to integrate measurement and empirical feedback mechanisms into development and maintenance processes [17], [21]. These *empirically guided processes* can be used, for example, to focus tool and technique application on high-payoff areas. The underlying principle is to *make measurement active* by integrating measurement into the software process. This work contrasts sharply with the primarily passive use of measurement in the past.

The *Software Management Environment* (*SME*) is a management tool used to display models of the software development process based on comparing data from previous projects [8]. *SME* was developed at the Software Engineering Laboratory (*SEL*). It provides the project manager with the ability to observe, compare, predict, analyze, and control key software development parameters such as effort, reliability, and resource utilization.

These three systems are primarily applicable to monitoring and controlling large-scale software development projects. In addition, each is aimed primarily at collecting product metrics which will be used to provide management feedback and models. In contrast, GINGER is primarily intended for use in an educational environment. GINGER:

- allows instructors to monitor the progress of students in groups of more than 100.
- allows students to control their own activities based on feedback from the environment,
- is capable of collecting all of the required data automatically without disrupting the programmer's activities,
- is capable of storing and managing large amounts of data, and
- is able to provide personalized analysis to students in a timely fashion using data collected automatically.

Basili and Reiter [1] have described the use of automated data collection and analysis capabilities similar to those provided by *GINGER*. Their environment was used to support a quantitative comparison of software development approaches. *GINGER* extends the capabilities described by Basili and Reiter by adding the capacity for statistical analysis and feedback as described in Section 4. In addition, the environment described by Basili and Reiter was used only to support research. *GINGER* has been used to support both research and education aimed at improving programmer performance.

3 Overview of GINGER

GINGER was originally constructed to make it possible to precisely track student progress in programming classes. The initial focus of *GINGER* was on evaluating and testing programs submitted as part of class assignments. Capabilities for quantitatively evaluating both the programs and the students' programming processes were added later.

3.1 Goals

As we have noted, the information provided by measurement is most effective in improving the software process if it is provided to managers and developers in the form of feedback on the progress of the project. While off-line feedback, such as progress reports, is useful, we believe that on-line feedback is more valuable. Thus, our goal is to provide an environment that analyzes the data in real-time and provides the results to students and instructors in a form that will help them assess and improve the students' capabilities.

3.2 Requirements for a Quantitative Improvement Environment

This section describes the requirements for a quantitative improvement environment that motivated the construction of *GINGER*.

3.2.1 Data Collection

Measurement can provide data that can be used to improve the software process, both at the organization level and the personal level. However, collecting this data can be difficult. Developers often resist the additional work required to keep the necessary records and are sometimes reluctant to provide detailed information on their activities. In part, this stems from the fact that it is difficult for them to manually provide accurate information about their activities. When the required information is obtained, it is often historical, having been constructed from recollection. Because of this, the quality of the data may be suspect. Thus, a measurement environment should collect data automatically and unobtrusively. Note that, in order for this to be possible, it must be possible for the developer to perform most of his or her tasks within the measurement environment.

3.2.2 Data Storage and Retrieval

Providing good feedback for improving programmer productivity requires historical information (both raw data and the results of analysis) from previous projects. A quantitative improvement environment should, therefore, store this data for convenient retrieval at any time. The environment should also be able to manage the large amounts of data (e.g., data on program modification) that can be generated. Thus, some mechanism for program modification is required.

3.2.3 Data Analysis and Feedback

Several different analyses may be required to meet a specific objective. The environment should, therefore, be capable of collecting all of the necessary raw data and extracting the information required to perform the various analyses. To increase the usefulness of the results, the environment should also support statistical analysis of the resulting data. Such statistical analysis is especially important in an improvement-oriented environment.

Data collection and analysis are necessary for an improvement-oriented environment, but they are not sufficient. The results of the analysis should be provided to the programmer as feedback. In order to be useful, this information should be presented in a format that is easy to understand.

Finally, since the programmers in educational environments (both university and industrial) are novices, the environment should provide not only the results of analysis, but also advice on how to make improvements. This advice should be provided in a timely manner, so the environment should be capable of executing the analysis and returning the information to the programmers in real-time.

4 Current Implementation

The current version of *GINGER* is implemented on the *UNIX* environment [22]. The principal characteristics of the current implementation are:

- *GINGER* has been implemented using the *C* language to provide for portability across UNIX platforms. The current implementation consists of approximately 5,000 lines of C code. (*GINGER* has also been ported to other environments by rewriting some of its low-level functions to accommodate the new operating system.)
- Much of *GINGER*'s current functionality, particularly that in the *Data Collection*, *Data Management*, and *Information Feedback* units, has been implemented using standard *UNIX* tools and file system functions.
- *GINGER* currently runs on a network of workstations linked by an *Ethernet*. Copies of the *Data Collection* unit execute on each programmer's workstation; all other units execute on the manager's workstation.

4.1 Architecture

GINGER is composed of four logical units: the *Data Collection Unit*, the *Data Management Unit*, the *Data Analysis Unit*, and the *Information Feedback Unit*. Figure 1 illustrates the architecture of *GINGER* and shows its interaction with the programmer. In the figure, solid arrows indicate data flow and dashed arrows indicate control flow.

4.2 Data Collection

The raw data collected by *GINGER* includes both process data and product data. Process data is used to trace a programmer’s progress during the project. Product data is used to evaluate a program’s size and quality. Combining these two types of data makes it possible to evaluate a developer's capability and productivity. Process data includes both terminal access data (access date, login time, and logout time) and command execution data (execution date, execution time, command name, and CPU time). Product data consists of the program itself as well as an history of modifications. The history of modifications includes, for each update: update date, update time, and difference between successive versions of the program.

Because *GINGER* uses standard *UNIX* tools and file system functions to collect this data, its interpretation involves some assumptions. The principal assumptions are:

- All of a programmer's files are stored in a single directory under their home directory. This requirement is easily achieved by adopting appropriate conventions.
- All work is performed on the workstation. While it is possible that some work is performed off-line, this work is not counted. In practice, we have observed that students appear to perform their work on-line.

4.2.1 Process Data

Terminal access and command execution data is extracted from the standard *UNIX* accounting data by removing unnecessary information and rearranging the rest. Accounting data for each programmer is obtained by invoking the *UNIX* accounting system commands "last" and "lastcomm"[1] [23]. Since the accounting system is typically set to delete this data at specific intervals (e.g., each weekend or at the end of the month), a C program within the *Data Collection Unit* collects the data just before it is deleted by the accounting system.

1 The UNIX "last" command prints information about the sessions of specified users and terminals. This information includes: the time at which the session began, the duration of the session, and the terminal used for the session. The "lastcomm" command provides information on previously executed commands. It can be used to list command executions by command, user, and/or terminal.

Accounting data used to compute terminal access consists of: 1) the username of the programmer who has accessed the terminal, 2) the terminal identifier, 3) the access date, 4) the login time, 5) the logout time, and 6) the elapsed time between login and logout. Accounting data used to compute command execution consists of: 1) the command name, 2) the username, 3) the terminal identifier, 4) the amount of CPU time (in hundredths of a second) used to execute the command, 5) the date, and 6) the time at which the command was executed. Figure 2(a) illustrates a typical set of accounting data.

The transformation from accounting data to process data is performed in three steps. They are:

Step 1: Retrieve the accounting data using the UNIX "last" and "lastcomm" commands.

Step 2: Sort the accounting data into chronological order and extract the key information. This step is performed by the *Data Collection Unit*.

Step 3: Store the transformed data in the *Process/Product Data Base*. This step is performed by the *Data Management Unit*.

The transformed data is illustrated in Figure 2(b).

4.2.2 Product Data

Since the program text produced by each programmer is maintained as *UNIX* files, relevant data such as date and time of modification, and access rights[2] are maintained automatically by the file system. Product data, therefore, can be obtained using *UNIX* file system functions [23]. Product data is collected by applying the following procedure at five minute intervals:

[2] Information about access rights is used occasionally to determine the type of a file. The value of the “executable” attribute can be used to determine whether a file is text or object code.

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. This step is performed using *UNIX* file system commands.

Step 2: Construct a pair (F_u, t_u) for the file F_u and the time t_u obtained in Step 1. This step is performed by the *Data Collection Unit*.

Step 3: Compute the difference, ΔF_i, between the new version of the file, F_u, and the last version, F_l, in the *Process/Product Data Base*. This computation is performed by the *Data Management Unit* using the *UNIX* file comparator "diff" [23].

Step 4: Add the pair (ΔF_u, t_u) to the history of modifications and replace the latest version of file, F_l, by F_u. This function is also performed by the *Data Management Unit*.

Details of the product data are illustrated in Figure 3. The history of modifications includes a header and a series of pairs of (t_i, ΔS_i, $1 \leq i \leq n$), where t_i is an update time and ΔS_i is a set of scripts (see Figure 3(a)). The set of scripts, ΔS_i, corresponds to the difference ΔF_i in Step 3 above. (There are five types of scripts, which correspond to those defined for the UNIX editor "ed" [23]. The description of each script type is summarized in Table 1). The version of the file at time t_i can be generated by passing ΔS_1, ΔS_2, ..., ΔS_i to the "ed" editor. The latest version of the file (Figure 3(b)) can be produced by passing the complete series of ΔS_u to "ed" [23]. The program modifications are also computed from this script, as described in Section 4.4.

4.3 Data Storage and Retrieval

Data is managed by product and, within a project, is arranged by programmer. The data stored for a given programmer includes both the process data (Figure 2) and product data (Figure 3) collected by the Data Collection Unit and the results of the analysis performed by the Data Analysis Unit.

4.4 Data Analysis

An example of the type of analysis provided by the current implementation of *GINGER* appears in Figure 4. This analysis consists of information on programming effort, the size and quality of the resulting program, and program modifications, which are primitive measures of programmer productivity.

Four different measures of programming effort are computed: 1) calendar days, 2) total terminal access time (in minutes), 3) total number of command executions, and 4) total CPU time (in hundredths of a second) for command execution. Command executions and their associated times are further classified into four categories based on the type of command: program editing, compilation, linking, and execution. These measures are derived from the terminal access and command execution data described in Section 4.2.

Measures of program size and quality are obtained from the latest version, F_l, of the files which make up the program (see Figure 3(b)). Two measures of size are currently used: the total number of lines and the total number of modules (see Figure 4). The quality measure is the ratio of the number of successful tests to the total number of test cases applied when the program is evaluated. The computation of this measure is described in Section 6.1.

Program modifications are computed based on the history of modifications and the latest version of the file. Figure 5 provides a typical example of program modification. The following six measures are used to characterize program modifications:

L_i : the number of lines in the initial program (i.e., the number of lines in the program the first time it is saved)

L_a : the number of lines which are added to the program in an editing session

L_c : the original length of a block of lines that is changed during an editing session

$L_{c'}$: the new length of a block of lines that is changed during an editing session

L_d : the number of lines which are deleted from the program in an editing session

L_r : the number of lines in the final program

The following relationship between the measures is easily 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 computed 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 computed based on the script of the history of modifications in the product data. (The relationship between the types of scripts and the values of L_a, L_c, $L_{c'}$ and L_d is shown in Table 2.)

The number of errors removed during the program development process is also estimated (see Figure 4) as is the sum of the error life span [22], which is used as a measure of programmer productivity. The number of errors removed and the error life span are computed automatically based on program modification data [16], as described in Section 6.1.

4.5 Feedback

In programming courses, in both academia and industry, assessment of program quality is an essential part of evaluating student progress. One commonly used approach is to test the programs submitted for an assignment using a common set of test data. This approach, however, can be difficult to use when a large number of programs must be evaluated in a limited amount of time.

GINGER provides a facility for automatically testing student programs. The outline of this facility is illustrated in Figure 6. A set of test cases, each consisting of an input together with the corresponding correct output, is prepared for each program by the instructor(s). Each program to be tested is retrieved from the *Process/Product Data Base* and the test cases are applied to it. The program output is compared to the expected output and, if they differ, the test case is considered to be unsuccessful. The results of the comparison are summarized in a test report. Figure 7 provides an example of a test report in which *test1* is unsuccessful and *test2* is successful. The test report is forwarded to the student via electronic mail by the *Information Feedback Unit*, as shown in Figure 6 and he or she is given an opportunity to correct any errors that were identified.

4.6 Process Visualization Tool

As noted above, effective use of data on the progress of software development activities requires that this data be provided as feedback to the developer. This data is most useful if on-line data, as well as off-line date such as progress reports, is available. In addition, it is necessary that this data be presented in a form that is easily understood. This is especially true for novice developers who are not familiar with this type of information.

We have constructed a tool to help developers visualize the software development process. The *Process Visualization Tool* is embedded in *GINGER*'s *Data Analysis Unit* and *Information Feedback Unit*. It provides the developer with: 1) the prescribed development plan, 2) the current status of the project along with a history of his or her activities, 3) a projection of the future status of the project based on the historical data, and 4) the difference between the prescribed plan and the projected one. This information is presented graphically whenever requested by the developer.

Figure 8 illustrates the basic components of the Process Visualization Tool and the data that flows between them. The tool consists of five components: the *Past Process Evaluator*, the *Future Process Predictor*, the *Programmer Productivity Comparator*, the *Feedback Manager*, and the *Image Constructor*. The *Past Process Evaluator*, the *Future Process Predictor*, and the *Programmer Productivity Comparator* are part of *GINGER*'s *Data Analysis Unit*. The *Feedback Manager* and the *Image Constructor* are contained in the *Information Feedback Unit*.

The *Past Process Evaluator*, *Future Process Predictor*, and *Programmer Productivity Comparator* are part of *GINGER*'s *Data Analysis Unit*. The *Past Process Evaluator* uses current programmer productivity data obtained from the *Programmer Productivity Evaluator* to update the developer's productivity history. The data obtained from the *Programmer Productivity Evaluator* consists of programming effort, program size and quality, and program modifications, as described in Section 4.4 and illustrated in Figure 4. The updated productivity history is stored in the *Product/Process Data Base* by the *Data Management Unit*.

The *Future Process Predictor* computes a projection of the development process from the current time to the prescribed date of completion. This projection is based on trends derived from the historical data computed by the *Past Process Evaluator*.

Several different models may be used to compute the projection. Currently, we assume that the deviation from ideal behavior is constant with respect to (calendar) time and equal to the current deviation. The future deviation is then extrapolated using a linear least-squares fit.

The *Programmer Productivity Comparator* predicts the value of programmer productivity for the process completion time projected by the *Future Process Predictor*. The difference between the target productivity and the projected value is then calculated.

The *Feedback Manager* and the *Image Constructor* are included in *GINGER*'s *Information Feedback Unit*. The *Feedback Manager* requests analysis of programmer productivity from the *Data Analysis Unit* at a specified time interval. This interval can be selected by the project manager. When a developer requests information about his or her productivity, the *Data Analysis Unit* sends the data to the *Image Constructor* which provides the developer with a graphical view of the development process. This graphical data is displayed using the *X-Window* system.

Figure 9 illustrates the feedback information provided by GINGER through the *Process Visualization Tool*. Figure 10 illustrates a graphical terminal display.

5 Assessment of *GINGER*

To evaluate the effectiveness of the information feedback provided by *GINGER*, we performed an experiment using a programming project in an undergraduate course in the Department of Information and Computer Sciences at Osaka University.

In this experiment, sixty programmers were divided into two teams (designated *Group I* and *Group II*) of equal size. Each programmer developed a compiler for translating Pascal source code into assembly language. The programs were all developed to the same specification using the C programming language. Students were given three months to complete their programs. The resulting programs were evaluated using *GINGER*'s testing capability (see Section 4.5). Thirty test cases were used and the criterion for successful completion was passage of 100% of the tests.

Students in *Group I* were instructed to control the size of their programs. A target range of 1476 - 1563 lines of code was established based on the size of programs from similar projects over the previous three years.[3] Students in *Group II* were not given a target program size. Initially, none of the students were provided with information on program size. During the final month of the project, however, students in *Group I* were allowed to use *GINGER*'s visualization tool (Section 4.6) to obtain feedback on the size of their programs. Approximately half of the students in *Group I* used *GINGER* for feedback. Those that did not use *GINGER* for feedback were designated as a third experimental group, *Group III*.

Comparisons were made only among completed programs, that is, those that passed the acceptance tests. The hypothesis was that, if the information feedback was effective in helping programmers control program size, the variation in size among *Group I* programmers would be smaller than that among *Group II* and *Group III*

---

3 The previous projects used the same specification and the results were measured using *GINGER*. These projects did not make use of *GINGER*'s feedback capabilities, however.

programmers. The results of the comparison are shown in Table 3. The number of students in each row indicates the number who successfully completed the assignment from each group.

Evaluation of the data in Table 3 indicates that there were differences between the two groups with respect to both the mean and variance of program size. The mean program size of 1843 lines of code for *Group I* is smaller and closer to the target range than the means of either Group *II* (2152) or *Group III* (1955). The significance of this difference was confirmed using a t-test. The variance in program size within *Group I* (406^2) was also smaller than that for *Group II* (858^2) and Group III (1101^2). The significance of the difference in the variances was confirmed using the F-test. The differences were significant at the 0.05 level in both cases.

Figure 11 shows the growth in program size during the project and provides a dramatic illustration of the differences among the three groups. As Figure 11 shows, the range of sizes for the groups were nearly identical before the feedback was introduced. In the latter stage of the project, the range of program sizes for *Group I* remained relatively small while those for the other two groups increased dramatically.

The similarity among the groups before the introduction of feedback indicates that, without this information, students in *Group I* were no more (or less) effective than the others in controlling the size of their programs. When provided with feedback, however, the students in this group were able to effectively control the size of their program, as indicated by the decrease in the variance in program size. From this we conclude that the feedback provided by *GINGER* was effective in helping programmers achieve their goals.

In their experimental evaluation of the effects of goals on programmer performance, Weinberg and Schulman [24] concluded:

> "If given specific objectives, programmers can make programming choices in such a way that they will meet those objectives – provided that they do not conflict with other objectives."

Our experience, supported by the data in Table 3 and Figure 11, indicates that simply providing programmers with objectives is not enough. Feedback on performance is necessary if programmers are to achieve those objectives.

6 Applications of GINGER

This section describes two applications of the *GINGER* system and the two tools described in Section 4.

6.1 Process Evaluation

Human factors are one of the most important determinants of software quality and productivity [7]. Programmer capability is, for example, the largest among the fourteen *COCOMO* cost drivers, with a range of 2.03 [3]. It is also known that, for most performance variables, there are large differences in individual performance among programmers [20]. Programmer performance, therefore, is one of the most important attributes of the software process to be measured.

Measurement of programmer performance is important in both academic and industrial educational environments. In the industrial environment, in particular, it is important to reduce variability among individuals.[4] In industrial environments, it is also important to evaluate team activities since teamwork is one of the most important things to be learned in the industrial environment (see Section 6.1).

We have developed a programmer performance model which is based on the concept of error life span. The life span of an error, T_e, is defined as the time elapsed from when the error manifests itself as a fault in the software to when the fault is removed. This concept was first proposed by Mills [19], who used "error days" as a measure of software quality. Basili and Weiss [25] also used program modifications as a means of evaluating the software process. Results of experimental evaluations have shown a high correlation between this model and "aptitude" among student programmers [16].

---

4 In Japan, reduction of variability among programmers is aimed at improving the abilities of those at the lower end of the distribution while preserving the abilities of the more capable individuals. Thus, the average ability of the group is improved. This reduction in variability means that the software development process is more predictable.

In these experiments, obtaining error life spans was very expensive. It was necessary to trace and analyze all files used in the experiments and manage large amounts of data by hand. The amount of effort required for this means that obtaining error life spans in industrial training courses or actual software development projects would be prohibitive. Similarly, it would be difficult, if not impossible, to provide this information as feedback to the programmer during the course of the project.

To reduce the amount of effort required to obtain this information, we devised a method to automatically estimate error life spans. This method is based on the observation by Dunsmore and Gannon that program changes (i.e., textual changes between successive versions of a program) are correlated with the total number of errors in the program [10]. Thus, each line modified during an edit session corresponds to a single error. The error life span, therefore, can be estimated from the number of times each line is created and deleted. This information is easily obtained from the program modification histories collected automatically by *GINGER*.

Based on the results of Dunsmore and Gannon [10] and our own experience in tracing and analyzing program files in the experiments mentioned above [16] [18], we make the following two assumptions:

1) The purpose of program modifications[5] during each edit session is to remove faults.

2) A set of lines created during one session and modified during another corresponds to one error.

[5] Here, only modifications to program content are considered. Other modifications, such as indentation, or changes to comments, are not counted.

Figure 12 illustrates the estimation of error life span. In Figure 12, t_i ($0 \le i \le 3$) represents the time when the i-th edit session has terminated. P_i represents the program text created during the i-th edit session. l_j ($1 \le j \le 5$) represents lines modified during the third edit session. From assumption (1), the purpose of modifications of program text at the 3rd edit session is to remove faults. From assumption (2), the modified lines l_j can be divided into two subsets: $\{l_1, l_2, l_3\}$ and $\{l_4, l_5\}$. Thus, for the third edit session, two errors were removed. The first corresponds to $\{l_1, l_2, l_3\}$ and has a life span of $t_3 - t_1$, while the second corresponds to $\{l_4, l_5\}$ and has a life span of $t_2 - t_1$.

To validate this approach to the estimation of error life span, we computed the estimated sum of error life spans for the experiment mentioned above. The estimated value was then compared with the actual sum of error life spans. The coefficient of correlation between the two values is 0.90, indicating that the estimation technique is sufficiently reliable.

We have also applied this estimation method to a programming course in a Japanese computer company in order to evaluate team activities. A team performance model was defined by regarding a team as a virtual programmer. The team's performance was represented by the sum of estimated values of error life spans for all members of a team. The results of this experiment indicates that this team performance model provides a single value that has a high correlation with the amount of time that the team spends debugging. The coefficient of correlation is -0.83. The negative coefficient of correlation is due to the fact that the amount of time spent debugging is large for a team with low performance (i.e., a large sum of error life spans).

6.2 Cost Effectiveness of Software Reviews

We have developed a metric for evaluating the effectiveness of software reviews [15]. This metric was validated, using *GINGER*, in an experiment performed as part of a programming course at a Japanese computer company.

This metric, which combines features of metrics proposed by Fagan [11] and Colofello and Woodfield [5], computes the ratio of the reduction of the total costs to detect all faults in design documents and code by review to the virtual testing cost (the cost to detect all errors by testing) for the code. It is defined by the equation:

$$M = \frac{\Delta C_t - C_r}{C_t + \Delta C_t} = \frac{E_{dr} + E_{cr}}{e_d + e_c} \bullet \frac{c_t - c_r}{c_t}$$

where:

C_t = the cost (in minutes) for testing,

C_r = the cost (in minutes) for reviews,

ΔC_t = the reduction in testing cost due to reviews

e_d = the number of errors introduced during design

e_c = the number of errors introduced during coding

c_r = the average cost to detect an error during reviews

c_t = the average cost to detect an error during testing

E_{dr} = the number of errors detected during design reviews

E_{cr} = the number of errors detected during code reviews

Details of the derivation of this metric may be found in [15].

To validate this metric, we conducted an experiment as part of a course for novice programmers at a Japanese computer company. The course was for new hires who had no previous experience in programming or in conducting reviews. In addition to basic programming, students were taught the company's standard process for performing design and code reviews. Other important characteristics of the experiment were:

- There were seven teams of four or five developers each. The teams were organized by the instructor to minimize the differences in their capabilities.
- Each team developed a batch processing system using the same specification. Each team designed the system using *JSP* (Jackson Structured Programming) and coded it using *COBOL*. The final program size was about 2,000 lines of code.
- The programs developed by each team were tested by the instructor. Programs were required to pass all test cases.

The total cost of testing was computed as the total terminal access time while testing. This data was collected automatically using *GINGER*. Data required for the other parameters was taken from information reported off-line by each programmer.

Analysis of the experimental data reveals that the order of teams with respect to the values of this metrics corresponds with our intuitive impression of their performance on the review. This metric also provides higher stability in evaluating the effectiveness of the reviews than either Fagan's metric or that of Colofello and Woodfield.

We also found a high correlation (r = 0.98) between the total number of faults introduced in the design and coding phases and the reciprocal value of the average cost to detect a fault in the design review and code review. The reciprocal of the average cost to detect a fault in these reviews also has a very high correlation (r = 0.92) with the reciprocal value of the average cost to detect a fault in testing.

Based on these observations, we were able to devise a metric to estimate the cost effectiveness of software reviews using only parameters that can be computed during the review process. This metric, M_k, is defined by the following equation:

$$M_k = \frac{F(0.9152 - 0.00114c_r)c_r}{1090 + 4.88c_r}$$

where F is the total number of faults detected in reviews and c_r is the average cost to detect a fault through design and code reviews. We have used the metric to evaluate the cost-effectiveness of three review techniques that we proposed [14].

7 Summary and Conclusions

This paper has described the architecture and functions of *GINGER*, a measurement-based programming support system. *GINGER* automatically collects software process and product data and performs analysis to provide feedback to programmers. The information provided by *GINGER* allows students to assess their progress in programming projects and determine how their work compares with that of other students. Instructors can also use this information to identify students who need extra attention.

GINGER has been evaluated experimentally in programming classes at Osaka University. The results of this evaluation indicate that the feedback provided by *GINGER* is effective in helping programmers achieve their goals.

Two applications of *GINGER* in industrial education settings were also presented. The first involved the use of error life span to evaluate team performance on a programming task. The second was concerned with evaluating the cost-effectiveness of software reviews.

To date, *GINGER* has been used primarily in educational contexts, both academic and industrial. However, we believe that such an environment can also be valuable in providing assessment and feedback in actual software development projects. For example, project managers could use *GINGER*'s measurement capabilities and *Future Process Predictor* to obtain early indicators of problems. *GINGER* is currently capable of providing several of the management indicators described in [4]. Individual developers or programming teams could also use feedback on their own performance to modify or re-direct their efforts in a manner analogous to that used by students in the educational settings described here. As noted in Section 5, such feedback is essential for helping developers to meet assigned goals.

Finally, the discussion in this paper has focused on the use of *GINGER* to obtain information about code-level software products. However, with the advent of modern CASE tools, a considerable amount of design-level information is now available in machine-readable form. As a result, it should be possible to use *GINGER* to collect and analyze information about an emerging software design. We are currently investigating this possibility.

Acknowledgments

The authors wish to thank Dr. Lloyd G. Williams of Software Engineering Research for his technical advice on the experiments described here and for his valuable comments on an earlier draft of this paper. We also wish to thank Professor Victor R. Basili of the University of Maryland for his invaluable suggestions and stimulating discussions and Mr. Kouichi Kishida of Software Research Associates, Inc. for his valuable comments and support.

Mr. Yukio Mohri of Nihon Unisys, Ltd. assisted in performing the industrial experiment described in Section 6.2.

The authors also wish to thank Prof. Tohru Kikuno of Osaka University for his valuable comments and helpful criticism of this paper. Finally, special thanks go to the students of Osaka University who assisted in developing the *GINGER* system and who participated in the experiments described here.

References

[1] V. R. Basili and R. Reiter Jr., "A Controlled Experiment Quantitatively Comparing Software Development Approaches," *IEEE Transactions on Software Engineering*, pp. 299-320, 1981.

[2] V. R. Basili and H. D. Rombach, "The TAME Project: Towards Improvement-Oriented Software Environents," *IEEE Transactions on Software Engineering*, vol. 14, no. 6, pp. 758-773, 1988.

[3] B. W. Boehm, *Software Engineering Economics*, Englewood Cliffs, NJ, Prentice-Hall, 1981.

[4] J. A. Clapp, "Management Indicators," in *Encyclopedia of Software Engineering*, J. J. Marciniak, ed., New York, John Wiley and Sons, 1994, pp. 637-644.

[5] J. S. Colofello and S. N. Woodfield, "Evaluating the Effectiveness of Reliability-Assurance Techniques," *Journal of Systems and Software*, vol. 9, no. 3, pp. 191-195, 1989.

[6] S. D. Conte, H. E. Dunsmore, and V. Y. Shen, *Software Engineering Metrics and Models*, Benjamin Cummings, 1986.

[7] B. Curtis, ed., *Human Factors in Software Development*, IEEE Computer Society Press, 1985.

[8] W. Decker and J. Valett, "Software Management Environment (SME): Concepts and Architecture," SEL-89-003, Software Engineering Laboratory, 1989.

[9] J. R. Dunham and E. Kruesi, "The Measurement Task Area," *IEEE Computer*, vol. 16, no. 11, pp. 47-54, 1983.

[10] H. E. Dunsmore and J. D. Gannon, "Analysis of the Effects of Programming Factors on Programming Effort," *Journal of Systems and Software*, vol. 1, pp. 141-153, 1980.

[11] M. Fagan, "Design and Code Inspections to Reduce Errors in Program Development," *IBM Systems Journal*, vol. 15, no. 3, pp. 182-211, 1976.

[12] N. E. Fenton, *Software Metrics: A Rigorous Approach*, London, Chapman and Hall, 1991.

[13] R. B. Grady, *Practical Software Metrics for Project Management and Process Improvement*, Englewood Cliffs, NJ, Prentice-Hall, 1992.

[14] S. Kusumoto, K. Matsumoto, T. Kikuno, and K. Torii, "Approaches to Improving Effectiveness of Review Activities in the Technical Review Process," *Proceedings of the International Software Quality Exchange '92*, March, 1992, pp. 7B-1-7B-16.

[15] S. Kusumoto, K. Matsumoto, T. Kikuno, and K. Torii, "A New Metric for Cost Effectiveness of Software Reviews," *IEICE Transactions on Information and Sytems*, vol. E75-D, no. 5, pp. 674-680, 1992.

[16] K. Matsumoto, "A Programmer Performance Model and Its Measurement Environment," Ph.D. Thesis, Osaka University, Japan, 1990.

[17] K. Matsumoto, K. Inoue, H. Kudo, Y. Sugiyama, and K. Torii, "Error Life Span and Programmer Performance," *Proceedings of the 11th International Computer Software Applications Conference*, October, 1987, pp. 259-265.

[18] K. Matsumoto, S. Kusumoto, T. Kikuno, and K. Torii, "An Experimental Evaluation of Team Performance in Program Development Based on Model,"

[19] H. Mills, "Software Development," *IEEE Transactions on Software Engineering*, vol. SE-2, no. 4, pp. 265-273, 1976.

[20] H. Sackman, W. J. Erikson, and E. E. Grant, "Exploratory Experimental Studies Comparing Online and Offline Programming Performance," *Communications of the ACM*, vol. 11, no. 1, pp. 3-11, 1968.

[21] R. W. Selby, A. A. Porter, D. C. Schmidt, and J. Berney, "Metric-Driven Analysis and Feedback Systems for Enabling Empirically Guided Software Development," *Proceedings of the Thirteenth International Conference on Software Engineering*, Austin, TX, May, 1991, pp. 288-298.

[22] K. Torii, T. Kikuno, K. Matsumoto, and S. Kusumoto, "A Measurement Environment and Some Results at Class Experiments," *Proceedings of the 2nd International Workshop on Software Quality Improvement*, 1990.

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

[24] G. M. Weinberg and E. L. Schulman, "Goals and Performance in Computer Programming," *Human Factors*, vol. 16, no. 1, pp. 70-77, 1974.

[25] 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, 1985.

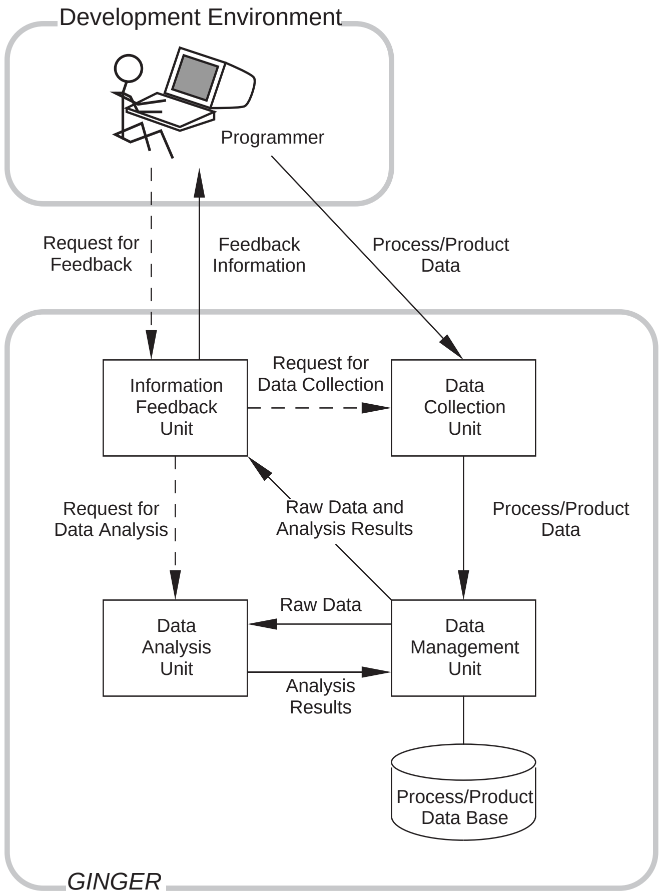

Figure 1. Architecture of GINGER

Terminal Access Data

```
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)
```

Command Execution Data

```
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

Terminal Access

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

Command Execution

Execution date (year-month-day)	Execution time (hour:minute)	Command name	CPU time (1/100 seconds)
1988-10-20	10:29	vi	5.06
1988-10-21	11:25	vi	4.07
1988-10-21	11:26	cpp	0.30
.			
.			
.			
1988-11-16	12:35	hc	0.06
1988-11-16	12:35	a.out	0.39

(b) Extracted Data

Figure 2. Process Data

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

        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);
        }
.
```

Header

Update time t_u
(year-month-day
hour:minute:second)

Script éS_u
(Append after line 0)

(Change lines 27 and 28)

(Delete line 14)

(Append after line 7)

(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 3. Product Data

Table 1. Editor Scripts

Script Type	“ed” Editor Function
`na` `  <text>` `.`	append `<text>` after line `n`
`nc` `  <text>` `.`	delete line `n` and replace it with `<text>`
`m,nc` `  <text>` `.`	delete the lines between line `m` and line `n` and replace them with `<text>`
`nd` `.`	delete line `n`
`m,nd` `.`	delete the lines between line `m` and line `n`

Programming Effort

Calendar days	10
Total terminal access time (min.)	1231
Total number of command executions	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

Program Size/Quality

Total number of lines	326
The number of modules	21
Percent of successful test cases	100

Program Modifications

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

Figure 4. Analysis Example

Table 2. Editor Scripts and Program Modification Measures

Script Type	Measure
na <text> .	L_a = number of lines in <text>
nc <text> .	$L_c = 1$ $L_{c'}$ = number of lines in <text>
m,nc <text> .	$L_c = n - m + 1$ $L_{c'}$ = number of lines in <text>
nd .	$L_d = 1$
m,nd .	$L_d = n - m + 1$

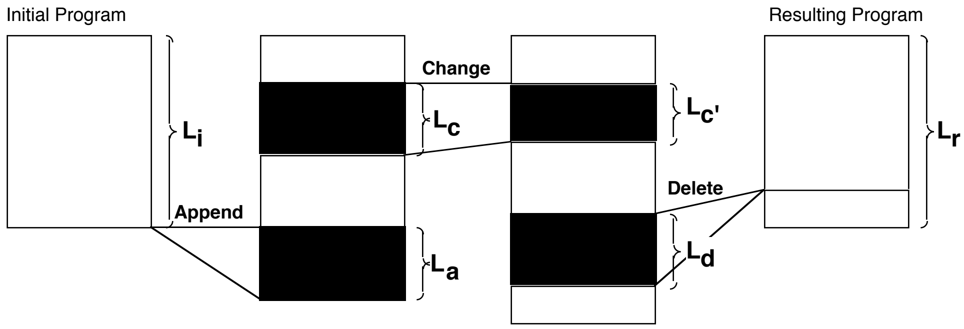

Figure 5. Basic Measures for Program Modifications

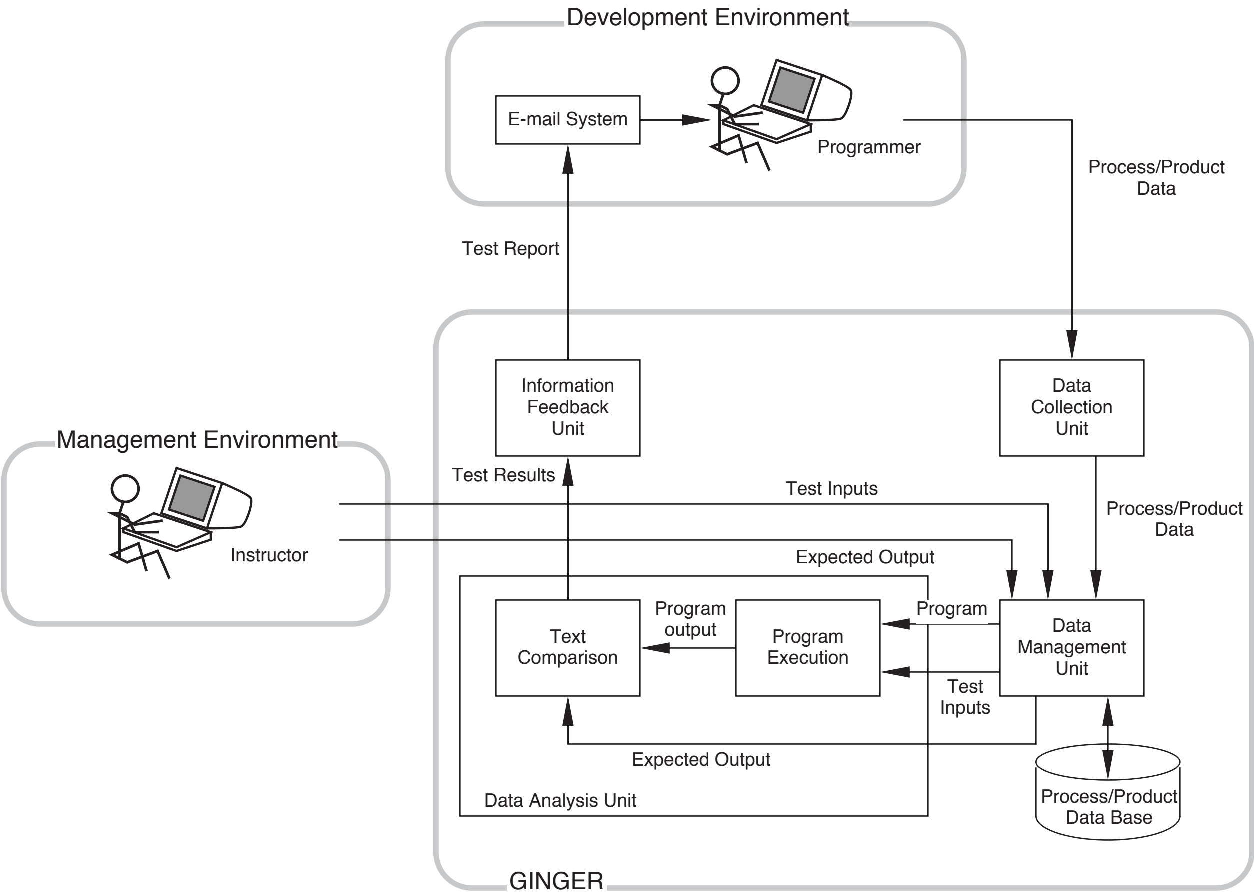

Figure 6. Automatic Testing and Reporting

```
****************************************
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 7. Example Test Report

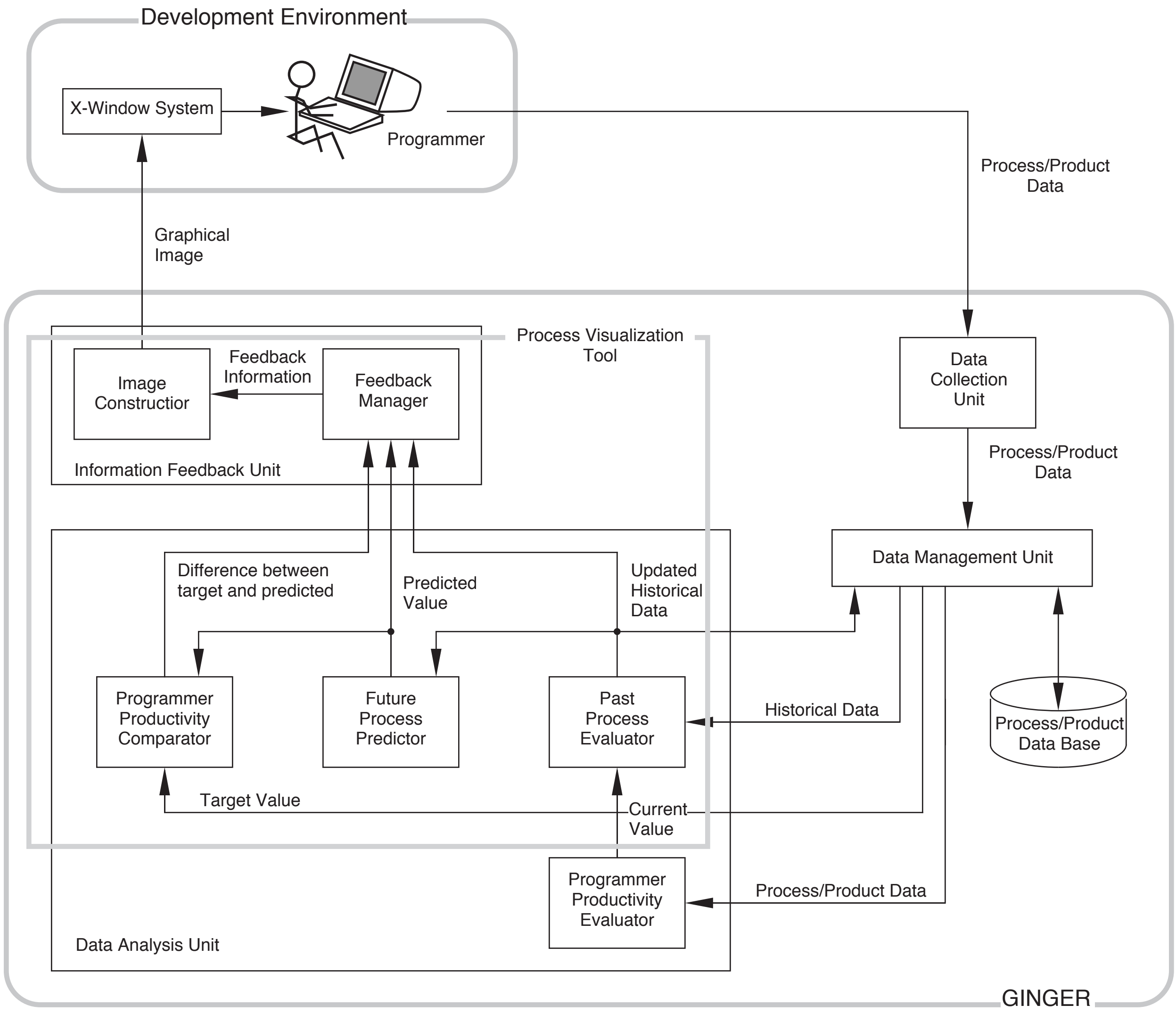

Figure 8. Schematic Representation of Process Visualization Tool

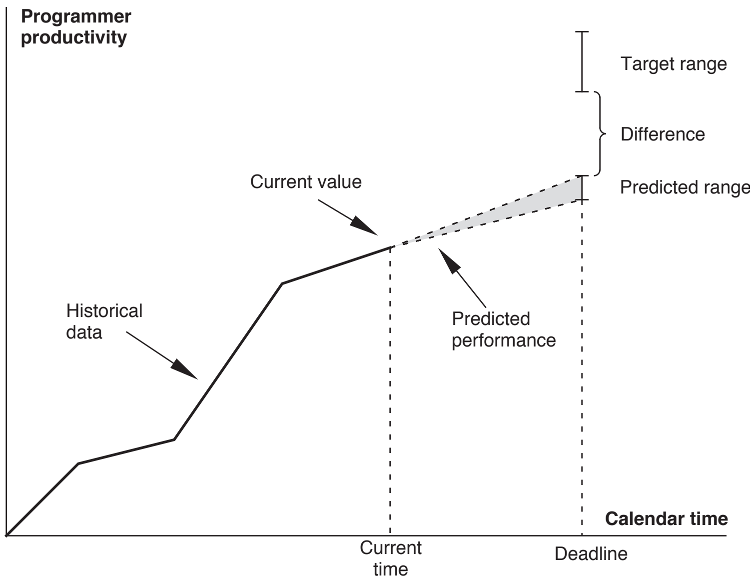

Figure 9. Feedback Information

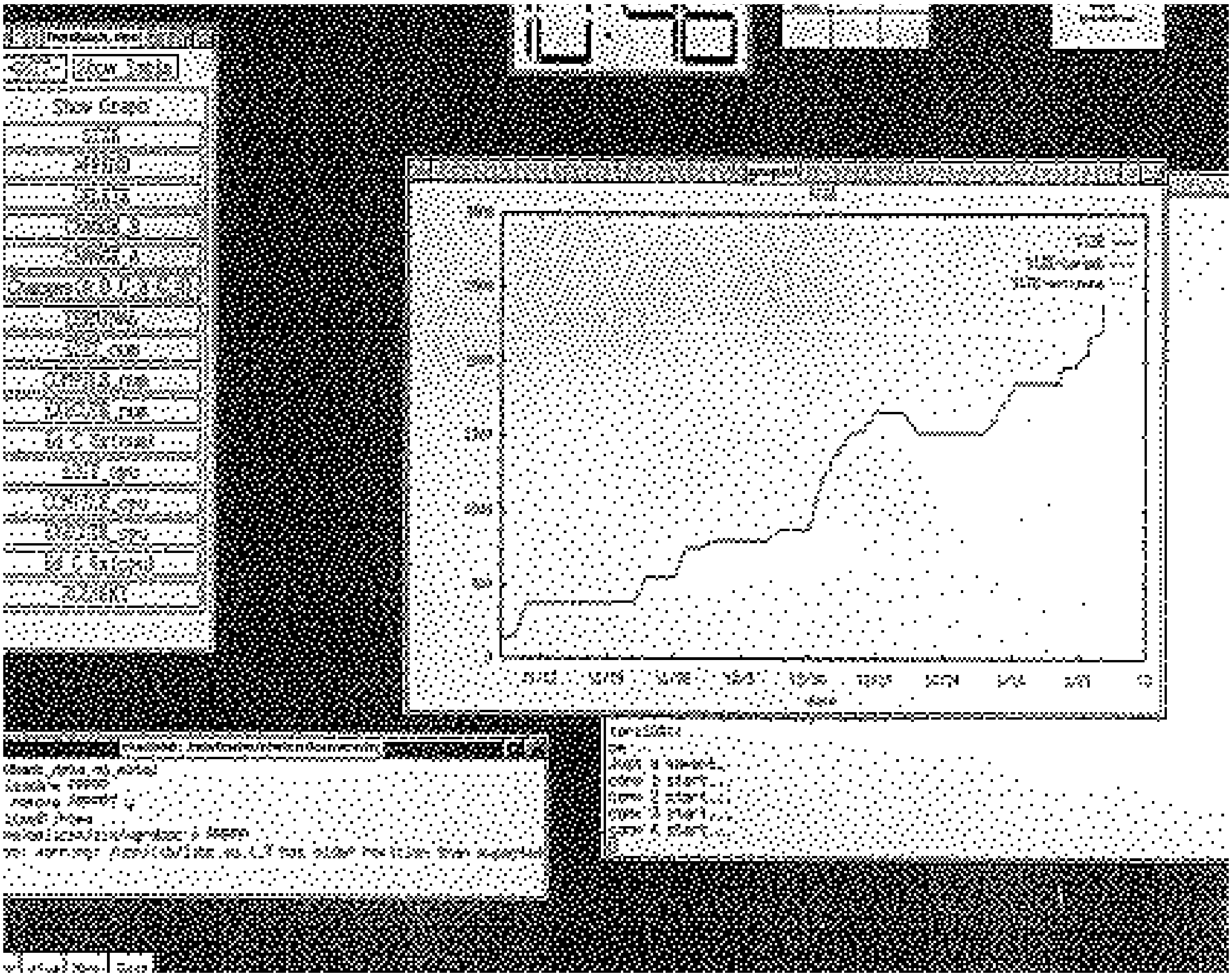

Figure 10. Sample Display

Table 3. Summary of Experimental Data

		Program Size (LOC)			
		Introduction of Feedback		Project Completion	
Group	**Students**	Mean ($\bar{x}$)	Variance (σ^2)	Mean ($\bar{x}$)	Variance (σ^2)
Group I	8	777	332^2	1843	406^2
Group II	13	760	357^2	2152	852^2
Group III	4	657	291^2	1955	1101^2

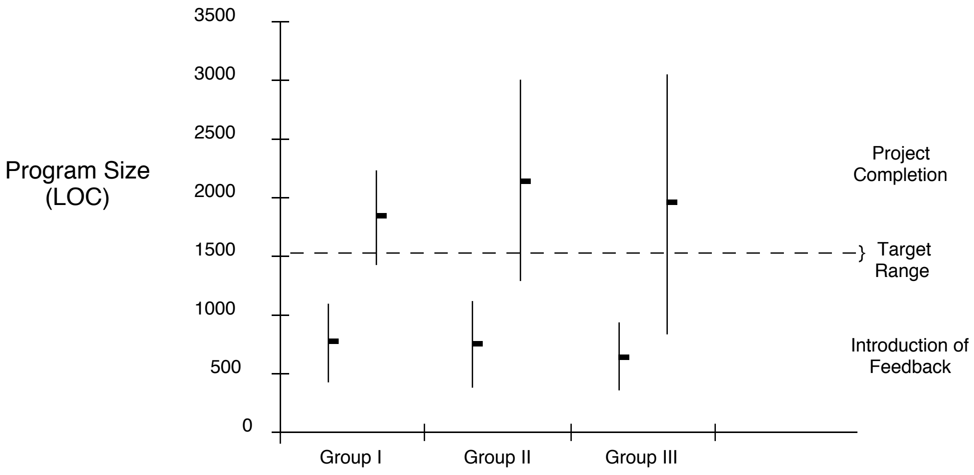

Figure 11. Program Size

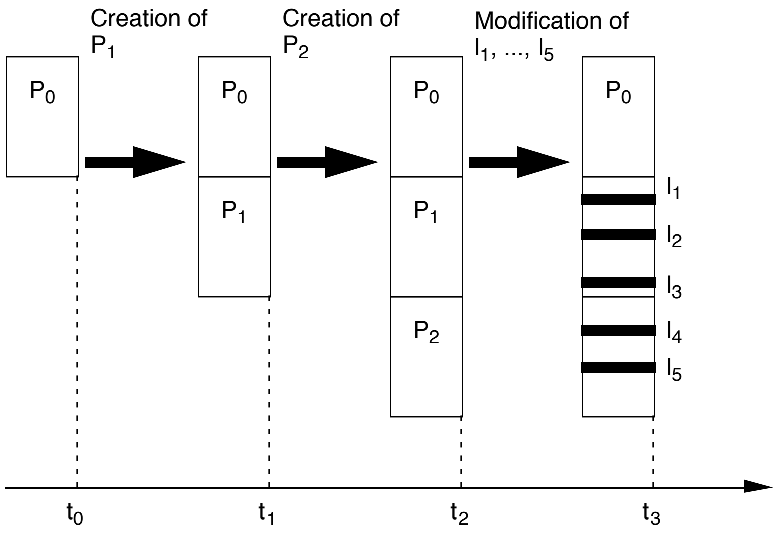

Figure 12. Estimation of Error Life Span