IEICE TRANS. INF. & SYST., VOL.E89–D, NO.4 APRIL 2006

PAPER

Effectiveness of an Integrated CASE Tool for Productivity and Quality of Software Developments

Michio TSUDA[†,††a)], ***Member***, **Sadahiro ISHIKAWA**[†††], ***Nonmember***, **Osamu OHNO**[†††], **Akira HARADA**[†,†††], ***Members***, **Mayumi TAKAHASHI**[††], ***Nonmember***, **Shinji KUSUMOTO**[†], ***and*** **Katsuro INOUE**[†], ***Members***

SUMMARY This is commonly thought that CASE tools reduce programming efforts and increase development productivity. However, no paper has provide quantitative data supporting the matter. This paper discusses productivity improvement through the use of an integrated CASE tool system named EAGLE (Effective Approach to Achieving High Level Software Productivity), as shown by various data collected in Hitachi from the 1980s to the 2000s. We have evaluated productivity by using three metrics, l) program generation rate using reusable program skeletons and components, 2) fault density at two test phase, and 3) learning curve for the education of inexperienced programmers. We will show that productivity has been improved by the various facilities of EAGLE.

key words: *CASE, productivity, quality, reuse*

1. Introduction

Software systems are becoming larger and more complex day by day. Demand to reduce the development cost and time is greatly increasing. To overcome this problem, it is essential to increase the productivity of individual engineers. Since the early the 1980s, it has been expected that the use of CASE (Computer Aided Software Engineering) will improve software productivity and quality. Many companies and researchers have been studying CASE tools extensively [3], [6], and have developed CASE tools with substantial effort. These tools are used for developing various software not only inside the companies, but also outside the companies, i.e. for outside customers. The following are some features of those CASE tools.

(1) They support standardized development methods provided by each company.
(2) They intend to integrate tools for system design, programming and testing.
(3) They have a program generation facility with software reuse techniques.

As mentioned above, increasing productivity is expected by introducing these CASE tools. However, to date, precise and quantitative analysis of the effects of the use of CASE tools has been scarcely reported [2]. In this paper, we will discuss the effects of the use of CASE tools on productivity, using various data which were collected in Hitachi through many projects for developing business application systems [9].

Hitachi, as other computer manufacturers have done, has been studying and developing CASE tools, especially integrated CASE tools.

Hitachi has a company-wide standardized system development method, named HIPACE (Hitachi High-Pace). Based on this method, an integrated CASE system named EAGLE [8] has been developed which is discussed in detail in Sect. 2. The EAGLE project started at the beginning of the 1980s with facilities still being enhanced.

Not only the EAGLE system has been developed, but they have started education on its use. Approximately 30 percent of the system engineers and programmers have completed the education and they develop business application systems using EAGLE.

The aim of this paper is to describe the improvement of productivity through the use of various metrics, i.e., program generation using reusable program skeletons and components, fault density at two test phases, and learning curve (which shows the degree of effort needed for an inexperienced developer based on the elapsed time for development). We will compare various data collected from the early the 1980s through the 2000s. An early version of EAGLE was used in the 1980s and the two enhanced versions of EAGLE was used in the 1990s and the 2000s. From these comparisons, we will show that enhancements of EAGLE facilities improve productivity. For example, generation rate has improved from 60% in the 1980s to 80% in the 1990s. Fault density has also improved in the sense that more faults have been detected at an earlier phase of program development. Furthermore, advance in the learning curve in the 1990s indicates the less effort for learning the enhanced system than the previous one. It also shows less elapsed time for development.

This paper consists of the following sections. Section 2 describes the features of EAGLE, comparing data collected during the 1980s, the 1990s and the 2000s. In Sect. 3, we analyze the program generation rate of standard program skeletons and components. Section 4 discusses the faults found at the test phase. In Sect. 5, the learning curve for programmers are defined, and two sets of project data are presented. We conclude our discussion and consider future research in Sect. 6.

Manuscript received June 17, 2005.
Manuscript revised October 7, 2005.
[†]The authors are with the Graduate School of Information Science and Technology, Osaka University.
[††]The authors are with the Hitachi Systems & Services, Ltd.
[†††]The authors are with the Hitachi, Ltd.
a) E-mail: m-tsuda@hitachi-system.co.jp
DOI: 10.1093/ietisy/e89–d.4.1470

Table 1 Summary of HIPACE and EAGLE for system development.

HIPACE (system development standards)		Facilities supported by EAGLE		
		Implemented in early 1980s **(EAGLE1)**	Enhancements in 1990s **(EAGLE2)**	Enhancements in 2000s **(EAGLE3)**
System Design Phase	Data Analysis - Data Definition	Data dictionary - Data Attribute (Data name/type /length)	Data dictionary - Constraint - Procedure	Data dictionary -Business Rule
	System Specification Design - System Analysis - Data Base Design - File/Record Design - Screen Design - Report Design - Performance Analysis	System Specification Libraries - File/Record - Screen - Report - Screen Transition	System Specification Libraries - Data Flow Diagram - System Flow - Data Base Schema - Code Table	System Specification Libraries - Business Model - Data Model
	Program Specification Design - Structured Design - Component Design -Object Oriented Design	Program Specification and Component Libraries - Program Skeleton (Batch processing) - Components (Function Process)	Program Specification and Component Libraries - Program Skeleton (On-line processing) - Components (Data Process)	Program Specification and Component Libraries - Program Skeleton (3-tier architecture processing) - Components (Business Logic)
Program Develop-ment Phase	Program Coding [Support Language]	Program Generation with Skeleton Library and Component Library [COBOL,PL/1]	Program Generation with Data Dictionary [COBOL,C]	Program Generation with Business Logic Dictionary [COBOL,Java,C,C++, .Net(C#,VB,COBOL)]
	Test Case Design	None (Documentation by hand)	Test Case Generation from Program Specification	Check List Generation from Program Specification
	Unit Test	Command Generation Source Tester	PAD Tester	Program Static Verification
Test Phase	Integration Test	None	None	None
	System Test	None	None	None

2. Overview of EAGLE

HIPACE is a company-wide standardized software development method in Hitachi. It was based on the waterfall model in the 1980s. In the 1990s, it was added on object oriented model. In the early the 1980s, we began to develop EAGLE as an integrated CASE tool system based on HIPACE.

As seen in Table 1, facilities supported by EAGLE have been enhanced. The enhanced facilities in the right column of Table 1 are additional ones developed in the 1990s, so that they can be explicitly distinguished. The early version of EAGLE in the 1980s is called EAGLE1 and the enhanced one is called EAGLE2 in the 1990s and we have developed EAGLE3 in the 2000s by adding new features to EAGLE2.

EAGLE1 and EAGLE2 target the large-scale business application systems on Hitachi mainframe computers. EAGLE3 targets client/server systems and Web applications [4].

Here we describe each phase of HIPACE and EAGLE in detail.

System Design:

The design phase consists of 3 sub-phases, data analysis, system specification design and program design. Each subphase is as follows.

- Data analysis: The data items used in the application systems are collected and defined in the data dictionary. A data item consists of the data attribute such as data name (e.g. English name, Japanese name), data type (e.g. nu-

meric, character, real), and data length. In EAGLE2, not only data attribute but also constraints and procedures are defined and are associated with each data item in the data dictionary. For example, data item classified as 'date type' has constraints of the values such as the range of year, month, and day. This data item is associated with a procedure which converts from Japanese date into the Christian Era date. The role of the data dictionary is to integrate and manage information resources for business application systems.

- System specification design: Data bases, screens, reports, files/records are designed and the specifications containing these designs are created, EAGLE manages these specifications in the specification libraries and controls access to these specifications. Designers can define these specifications interactively according to EAGLE's system design menus.
- Program design: Program structure and interfaces between programs are defined at this phase. The design is generally accomplished so that as many as possible program components in the library are reused. Also the new program components designed at this phase are stored in the library, so that these components can be reused for other projects in the future. Some program components were conventionally designed as functional procedures (e.g. file I/O procedures, error handling). In addition to these, program components that interact with associated data items have been added in EAGLE2, and business logic components have been added in EAGLE3. These program components are described in Sect. 3.
  EAGLE3 supports developing programs that have 3-tier architecture. A common segmentation of functionality within a 3-tier architecture is: (1) the Presentation Tier, which contains the user interface logic, (2) the Application Tier, which contains the business and system rules logic, and (3) The Database Tier, which contains the data and database logic. 3-tier architecture can ensure scalability and availability of the system.

Program Development:

The program development phase consists of 3 sub-phases, program coding, test case design, unit test.

- Program coding: EAGLE generates most parts of the programs using the data dictionary, the system specifications, and the program components. The remaining parts are hand-coded at this phase. EAGLE provides effective diagrams, named problem analysis diagrams (PAD) [5]. PAD is a tree structure chart for programming and testing.
- Test case design: Test conditions, test data, and expected results are prepared in advance. In the early the 1980s, test cases were designed by hand using standard forms (work-sheets) defined in HIPACE. In the 1990s, some test cases have been generated automatically by EAGLE2 based on standard program components used for the program generation. Details of test case generation techniques are described in Sect. 4.
- Unit test: Program modules are executed according to the test cases. If faults are detected in the program, they are corrected at this phase. The quality of the program can be validated through this testing process. EAGLE analyzes source programs generated by itself and provides command sequences through the unit test tool. Following these command sequences, programmers have to add test data/ conditions to the test commands, and proceed with the unit test interactively. The test coverage rate is calculated by EAGLE automatically.

Test:

The test phase is divided into the integration test and the system test phases. Limited facilities are supported for this test phase by EAGLE2.

3. Reuse Analysis

3.1 Program Skeleton and Component

It has been reported that software reuse is one of the effective strategies to improve program productivity [1]. We have measured the automatic generation rate of software to evaluate productivity. EAGLE offers facilities for software reuse at the programming phase.

A partial code collection representing the overall program structure is called a **skeleton**. A unit of a complete and reusable procedure is called a **program component** or simply **component**. Various skeletons and components are standardized and stored in the library and are provided by EAGLE for reuse. Figure 1 shows the flow of generating source code by EAGLE's. EAGLE1, EAGLE2 and EAGLE3 deal with phases 1, 2 and 3 in Fig. 1, respectively.

Skeletons are not complete code by themselves. Programmers have to insert appropriate code into the skeletons. For example, a file update skeleton consists of file open code, record search code, record update code and file close code. This program structure can be widely applicable to most file update programs. Programmers only insert record search/update code into this skeleton to obtain complete programs. Those skeletons are mostly 500–1000 lines long. We classify these skeletons into batch program type and on-line program type as shown in Table 2.

The structure of the batch program skeletons have not changed basically. But the structure of the on-line program skeletons have changed by the change in the architecture. We developed the 3-tier architecture skeletons in EAGLE3.

Function process components are procedures commonly used in various programs. They are classified into those for general purpose and those for specific application domains. In the 1990s, in addition to function process components to execute common procedures, data process components to access commonly used data items were added: For example, there is a program component associated with

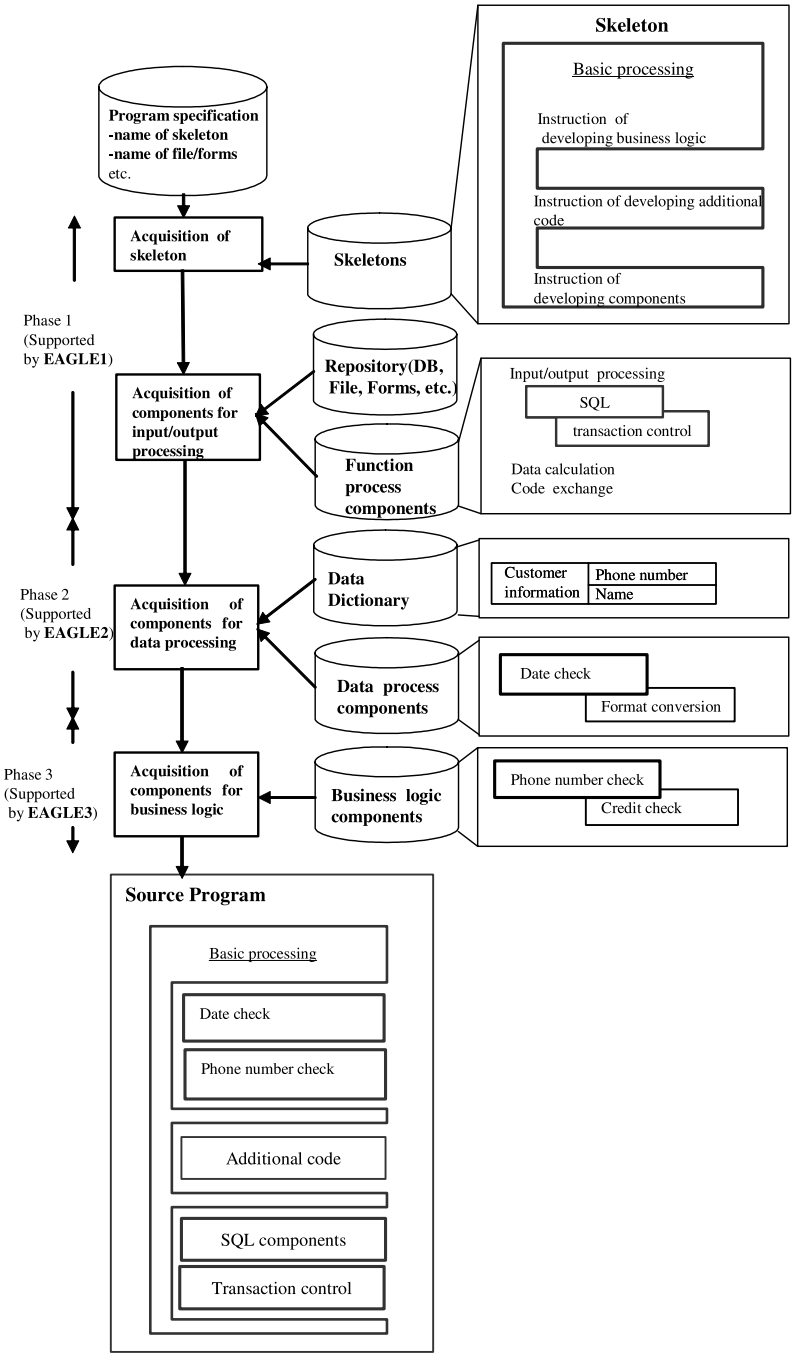


Fig. 1 Flow of generating source code by EAGLE's.

Table 2 Types of standard skeletons.

Type	Number of skeletons		Examples of standard skeletons
Batch Program	EAGLE1 and EAGLE2	31	File processing (conversion, update, merge) Report generation, Summing up
	EAGLE3	37	DB extraction, DB update
On-line Program	EAGLE1 and EAGLE2	45	Inquiry, Data entry, DB update
	EAGLE3	21	3-tier architecture skeleton

data item "birthday", which decomposes "birthday" into "year", "month" and "day" and displays them on a screen. This program component is very useful to the systems having access to data item "birthday". Table 3 shows the types of standard components. We prepared the 200 standard data process components. The name of data process components is specified in a data dictionary. Table 4 shows the structure of a data item in the data dictionary. In the 1980s only the attribute of the data was defined. In the 1990s, the program component names were added to them.

Program components for input/output data conversion define how to process the data items. In the case of Table 4, the input for the data item "BIRTHDAY" is converted from Japanese date to Christian Era, and also the output is converted vice versa.

The generation program example using "LX06" is shown below.

```
MOVE BIRTHDAY TO D06-I-YMDWA
CALL 'LX06' USING D06-TBL
MOVE D06-YMD TO BIRTHDAY
```

The data check processing parts and the input/output processing parts occupy on average 40% of the data processing parts in the data input program. We think that the generation of these processing parts is very important and effective. A high program generation rate can be attained by using the input and output processing components and skeletons.

In the 2000s, in addition to data process components, business logic components were added in EAGLE3. That is, several business logics are registered to business logic dictionary as components and reuse the component according to the requirements. Here, the business logic means common calculation, counting, checking formulas and so on. "Calculation of price" and "Checking the existence of customer phone number" are actual examples of the business logic.

3.2 Program Generation with Standard Skeletons and Components

To promote reuse of program skeletons and components, the facility to generate programs with skeletons and components was enhanced. The developer needs only select skeletons, components and screen/report/file specifications, so that they can generate fairly large parts of entire programs. In addition, EAGLE2 automatically generates check procedures and conversion procedures for data items which are defined in the screen/report/file specifications.

The program generation rate is defined as:

$$\textit{Program generation rate} = \frac{L_g}{L_t}$$

L_g: Lines of skeleton and component code generated by EAGLE1 or EAGLE2.

L_t: Total lines of program code including programmer's hand code.

Table 3 Types of standard components.

Type	Number of components	Example of standard components
Function Process Component	235	Data calculation, Code exchange, Error handling, Procedure for data base/data Communication access
Data Process Component	200	Data Check, Format conversion from Japanese date to Christian Era, Edition an amount of money into Japanese print format

Table 4 Example of data dictionary, birthday.

Data Name		BIRTHDAY
Attribute	data type	PIC 9
	length	6
	alias	birth
Data Process Component	data check	LX03(data-validation)
	input data conversion	LX06 (Japanese-date-to-Christian-Era)
	output data conversion	LX07 (Christian-Era-to-Japanese-date)

3.2.1 Program Generation in EAGLE1 and EAGLE2

We collected two sets of data from 43 and 44 programs generated by EAGLE1 and EAGLE2, respectively. In both cases, the generated programs were integrated systems for local city offices. There is no significant distinction between their feature. The average generation rate of 43 programs by EAGLE1 was 60% and that of other 44 programs by EAGLE2, 81%. Furthermore, we classified the programs according to program types such as batch programs and on-line programs. We further classified them, and averaged the generation rate for each type. Table 5 shows the difference of program generation rate between EAGLE1 and EAGLE2.

In most cases, the generation rate has improved approximately 20% by EAGLE2. This improvement is mostly due to the enhancement for generating program components associated with a data item. The generation rate for the data checking programs is low (73%), compared with other programs of batch types (report generation: 98%, file processing: 97%), since not enough program components for them are provided even in EAGLE2.

In the past, program components were developed and designed by extracting reusable blocks from each program. These were performed from the view point of individual designers. In the 1990s, the components for data access to each data item were developed and stored in the data dictionary. These components are automatically reusable to every project.

Also we can see that the generation rate for on-line programs is less than that for batch programs. Each batch processing system mostly uses one of the 31 types of skeletons without making any modifications. For on-line systems, however, it is difficult to reduce programmer's hand code since each screen operation is different for each project.

Table 5 Program generation rate (%).

Type		EAGLE1	EAGLE2
Batch	Report generation	69	98
	File processing	78	97
	Data check	50	73
On-line	Collating and update	74	77
	Inquiry data entry	50	72

Table 6 Program generation rate by EAGLE3.

Project		A	B
Batch		86	72
Online	P-Tier	0	46
	F-Tier	44	43
	D-Tier	98	99

3.2.2 Program Generation in EAGLE3

Programs in the Presentation and Application tiers are generated by using skeletons and components provided by the functionalities of EAGLE2. Programs in the Database tier are automatically generated from system specification.

We collected some data from two projects *A* and *B*. Both projects are large system developments for public systems (Each project includes more than 2000 programs) but the business logics are completely different. Table 6 shows program generation rate in the projects. The average generation rate of Project A was 74% and that of Project B, 75%. The generation rate in EAGLE3 for batch program is less than one in EAGLE2. This is because the configuration of batch jobs has changed in 2000's. Batch jobs in 1990's are mainly consisted of file processing. For example, in the update processing of index sequential file, the followings are processed: Input of transaction file, checking the data, sorting, matching of master file and error processing. Program for each processing is constructed. However, in 2000's, in addition to the conventional batch processing, processing for data base have increased. Though redundancy of job flows decreased and performance of processing mass data is improved, it is difficult to pattern the program control. To cope with the problem, we have developed two skeletons: (1) data base checking up skeleton and (2) data base free skeleton. Data base free skeleton provides with processing of calling modules in D-Tier and re-run and entire control should be described by developers.

Also, program generation rate in P-Tier and F-Tier becomes low. Currently, there are a lot of implementation methods of browsing and transition of screen for programs in P-Tier. It is necessary to standardize the implementation of P-Tier. With respect to programs in F-Tier, utilization of business logics are insufficient in the projects. We are going to educate the utilization of business logics in EAGLE3.

4. Fault Density Analysis

Fault density at a phase of development is defined as the number of faults detected at the phase normalized (divided) by the program size. We use here the fault density for 1000 lines at the unit test phase and the integration test phase. As described in Sect. 3, EAGLE does not generate complete programs so that the developers have to add the code by hand. Compared with the generated parts of the code which were sufficiently tested before, these hand-made code could easily contain many faults. These faults are generally found by exhaustive testing.

4.1 Test Case Generation

It is important to detect faults in the early phases of a development so as to minimize the troubles at the following phases. From this viewpoint, it is strongly suggested that the programmers perform sufficient unit test at the program development phase. (In HIPACE, the unit test is involved in the program development phase, not in the test phase.) In the 1980s, programmers had to design a large number of test cases for unit test and write them on work-sheets. There was no supporting tool for this. To increase the efficiency of the testing and to improve the productivity, we have introduced a facility of automatic generation of the test cases into EAGLE2. Now, standard test case skeletons are automatically generated by EAGLE2, according to the standard program skeletons. Programmers can complete the test cases only by adding some values to the test case skeletons.

As for unit test phase, EAGLE automatically generates test commands needed for the test tool by analyzing the source programs. For example, file I/O simulation, break points setting, and results displaying are examples of the test commands. The programmers have only to input data interactively according to generated test commands. Test coverage CO (the percentage of exercised statements) and C1 (the percentage of exercised logical segments) are automatically calculated. By using these facilities of EAGLE, the effort needed for unit test have been reduced greatly.

4.2 Comparison of Fault Density

Table 7 shows the difference of the fault density between two projects, A and B. These two projects were proceeded under the EAGLE2 environment; however, project A did not use the standardized test case. In project A, each programmer designed test cases by hand, and in project B, EAGLE2 provided standardized test cases (When these projects were carried out, the generation facility of EAGLE2 had not been completed, so that the programmer had to perform some customization and data input by hand.). Both project A and B are the development of integration systems for local city offices. The same development group developed both. Both systems are almost 10 K lines long and the generation rate of the code are approximate 50% in both projects.

We see here the distinction of the fault rates at the unit test phase and the integration test phase.

From Table 7 followings could be stated

(1) In Project B, more faults are found at the unit test phase (4.8 faults/klines), compared with those in Project A (3.7 faults/klines).
(2) At the integration test phase, less faults are detected in Project B (2.2 faults/klines in Project A and 2.9 faults/klines in Project B).
(3) Project B has a few more faults detected in total (6.6 faults/klines in Project A and 7.0 faults/klines in Project B).

Table 7 Comparison of fault density.

Phase	Project A		Project B	
	number of test cases (cases/klines)	faults density (faults/klines)	number of test cases (cases/klines)	faults density (faults/klines)
Unit test	36.8	3.7	52.6	4.8
Integratin test	15.3	2.9	14.1	2.2
Total	52.1	6.6	66.7	7

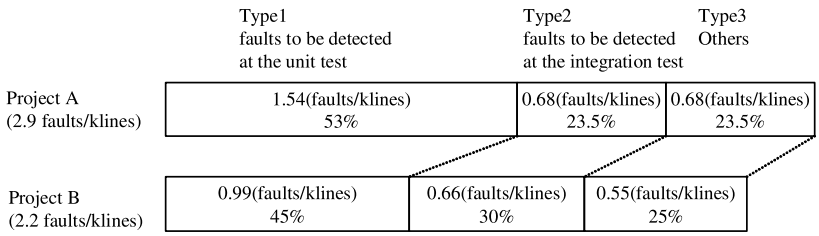


Fig. 2 Faults detected at the integration test.

From these, we could think that providing many standardized test cases will shift fault detection from the integration test phase to the unit test phase.

To confirm this hypothesis, we further analyzed the faults detected at the integration test. We classified them into 3 types. Type 1 is the fault which had to be detected at the unit test. For example, faults relating to input/output procedures, loop and branching procedures, and procedure specification are examples of this type. Type 2 is the fault which has to be essentially detected at the integration test. Faults of program interface and program specification are those examples. Other faults are classified into the type 3.

Figure 2 shows the classification of fault types for both projects A and B at the integration test.

From Fig. 2, we observe the following.

(1) In Project B, the number of Type 1 faults decreased. (From 53% in Project A to 45% in Project B).
(2) The numbers of Type 2 faults are almost the same. (0.68 faults/klines in Project A and 0.66 faults/klines in Project B).

This analysis shows that in the case of Project B, where the standardized test cases are provided, the number of faults removed at the unit test before the integration test has increased.

An adequate number of the test cases is essential to removing faults detectable at the unit test phase. By using the facility of EAGLE2, we can generate those test cases very efficiently.

Since the cost of preparing experimental setting like Project A and B was very high, we were unable to increase the number of projects for the comparison. Therefore, we did not get the statistically meaningful conclusions. However, the company thinks that EAGLE2's test case generation is a practically useful and important facility, so this facility is widely used in the company day by day.

5. Learning Curve Analysis

In addition to the facility enhancement of CASE tools, education of effective use was considered very important to the productivity and quality. We have started an educational course of EAGLE system, and we measured the elapsed time of program development by students, with EAGLE 1 and EAGLE2 environment.

5.1 Learning Curve and Improvement Rate

It is naturally considered that the elapsed times for developing similar programs will decrease with the experiences of a developer. A model to express the decrease of the effort has been proposed [7]. We use this model to describe the learning curve of the program development. Eq. (1) defines the learning curve.

$$Y = aX^{-n} \tag{1}$$

X: number of experience times to develop an application program,
Y: time required to develop the (X)th program (hours/1 klines) (normalized by the program size),
n: improvement coefficiency.

To see the decreasing rate intuitively, we also introduce improvement rate defined in Eq. (2) [7]. The improvement rate expresses the ratio of two required times Y_X and Y_{2X}, where $Y_X = aX^{-n}$ and $Y_{2X} = a(2X)^{-n}$.

$$\textit{Improvement rate} = \frac{Y_{2X}}{Y_X} \tag{2}$$

Y_{2X}: time required to develop the $(2X)$th program (hours/1 klines),
Y_X: time required to develop the (X)th program (hours/1 klines),

For example, if improvement rate with $X = 3$ is 80%, then this means that a programmer has developed the 6th program in 80% of the time of the 3rd program.

5.2 Data Collection Method

To calculate the learning curve and improvement rate, we

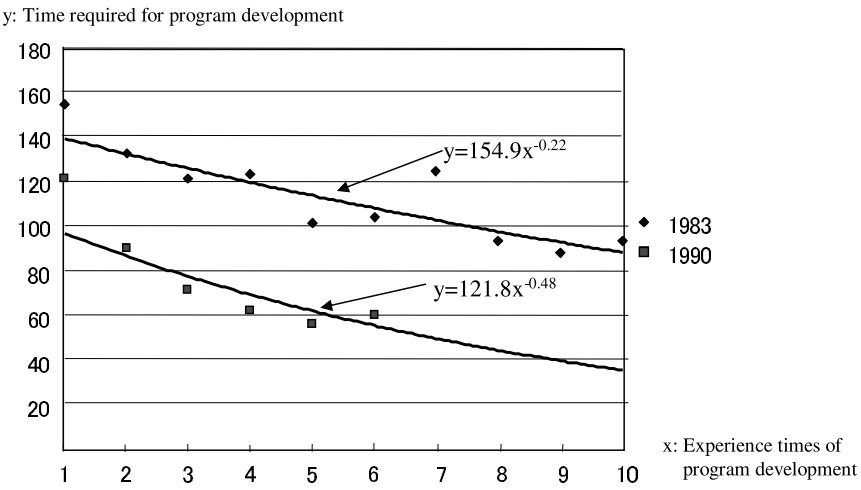


Fig. 3 Learning curve of program development.

use data collected from the educational course on EAGLE. Students are inexperienced, newly hired employees. Although most of them were graduated from universities, they are mostly not computer science major. We compared the data of 1983 where the EAGLE1 environment was used and the data of 1990 where the EAGLE2 environment was used. We do not see any significant difference programming capability in these two groups of the students. During a 4 week period, the students were required to develop business application programs of 400–600 lines in COBOL, using EAGLE1 and EAGLE2. The programs developed by students were batch programs and the programs developed in 1990 are similar to those in 1983. The following lists the feature of the educational course.

- 1983 data (Jul. 1982–Jul. 1983)
  36 students in the EAGLE1 education course
- 1990 data (Sep. 1990–Mar. 1991)
  90 students in the EAGLE2 education course

The following data were collected.

(1) Times elapsed for the various phases in the program development such as, from program specification editing to program compilation, test case design and testing, and documentation.
(2) Lines of program code.
(3) Number of test cases
(4) Number of experiences of program development with EAGLE1 or EAGLE2.

5.3 Improvement of Learning Curve

Figure 3 shows the learning curve based on the total of the program development phase in 1983 and 1990. The elapsed time to develop the first program in 1990 was reduced to 77% of the time required in 1983. The difference of the two curves suggests the effectiveness of EAGLE2 related to the elapsed time even for inexperienced developers. From this, we would say that in 1990, the productivity for the inexperienced developer, in the sense of lines of code produced in a certain time period, has improved by 1.3 times (77%) from that in 1983. For the 10th program, the difference of the elapsed times, (i.e., productivity in the sense mentioned above) was expected to increase to 2.3 times more (44%).

Table 8 Improvement rates Y_6/Y_3 of each phase.

Work type \ Year	1983	1990
Total	86	72
Program coding	86	69
Test case design & unit test	83	74
Unit test only	–	83
Documentation	86	72

Table 8 shows the improvement rate of $X = 6$ compared with $X = 3$ for each phase of the program development in 1983 and 1990. The improvement rate is defined as Y_6/Y_3. The improvement rate of the total of all phases improved from 86% in 1983 to 72% in 1990. In each phase of the development, the improvement rate improved approximately 15% between 1983 and 1990.

It is easily observed that in the case of 1990 the curve drops sharply and it becomes stable at the lower level.

From Fig. 3 we can also say that elapsed time does not reduce greatly after 4th-5th program developments in 1990. But elapsed time still reduce after 8th-9th program developments in 1983. It reveals that most programmers can master program development with less experience under EAGLE2 environment than under EAGLE 1 environment. It also shows that the programmers under EAGLE2 require only several experiences of the development with CASE tools before the elapsed time becomes stable. We would consider that this improvement is caused by the following.

- The elapsed time to develop the program was reduced by improving the program generation rate as shown in Table 8. Moreover, the elapsed time to test was reduced by improving the program quality. Students do not have to

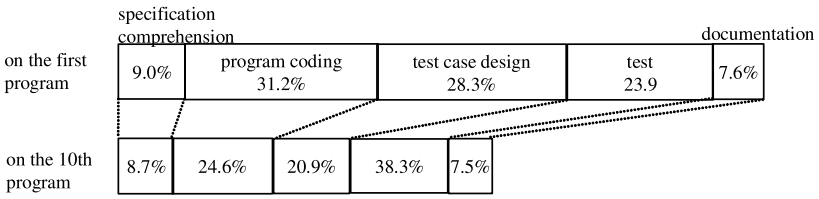


Fig. 4 Time required to develop a program.

know the detail about database/file scheme. Therefore, the remaining program to be coded by hand is rather simplified

- The quality level of the program was maintained by program generation using data process components. The purpose of the education is not to develop the program according to individual technology, but the purpose is to develop the constant quality program in the organization. We think that this purpose was achieved.
- The test tool and the test case generation are effective in the study of the unit test. Because student's learning outcome can be measured by the test coverage, effective feedback is possible.
- User interface has been improved in EAGLE2. For example, operations can be given by menu selection. Thus the students do not have to memorize them, and they can learn quickly.
- There were minor refinements of the course curriculum and text book in 1990. For example, sample programs are shown in the text book in 1990. Educational section personnels working full-time are taking charge of the education. The know-how of a past education has been accumulated.

6. Conclusions

In this paper, we analyzed the program generation rate of three versions of EAGLE (EAGLE1, EAGLE2 and EAGLE3), fault density, and learning curve of two versions of EAGLE (EAGLE1 and EAGLE2). From the comparison analysis of them, we found that the productivity has improved greatly through EAGLE enhancements.

(1) The enhancement of standard components and the skeletons in EAGLE2 improved the program generation rate by almost 20%.
The generation rate has been improved from 60% in EAGLE1 to 75% in EAGLE3.
From the above results, we can say that improvement of the productivity is expected greatly through EAGLE enhancements.

(2) More program faults were detected during the unit test by providing many standardized test case (4.8 faults/klines using standardized test case; 3.7 faults/klines designing test case by hand). This reduces the efforts at the integration test.

(3) The enhancement of EAGLE reduced the effort for learning program development very much. For example, improve rate has been reduced from 86% to 72%.
Users inexperienced with CASE require 4–5 works to efficiently develop programs with EAGLE.
Elapsed time for developing program was reduced by EAGLE2 to 77% for inexperienced developers and to 44% for experienced ones.

As described in previous sections, various enhancements of EAGLE have improved productivity. However, the following problems still remain:

(1) Automating test phase
The testing process is not automated yet and is inefficient. Figure 4 shows the time required for each phase of program development. We compared the data of the first program development and the 10th program development in 1990 project.
It is easily seen that the time for the test phase does not decrease greatly with experiences and it occupies a large part of the total development, although the time for the program coding and test case design reduces greatly.
EAGLE2 generates the standard test cases and EAGLE3 generates the standard check list; however, it does not generate test data and commands completely. Since programmers still have to modify and add their own commands by hand, they can easily make errors on the command and they have to do retest. Thus, in the future version of EAGLE, we plan that

- Not only the test cases but the test data and commands for unit tests will be generated based on the source programs, specifications and data dictionaries.
- User interface of test support tools and performance analysis tools will be enhanced.

The program static verification tools in addition to the test support tools have been developed in EAGLE3. This tool can detect the bug code, the resource consumption code, and the performance deterioration code. The observance of the coding rule is achieved. The detection of the security hole will be enhanced in the future.

(2) Increasing generation rate
Program generation rate has been improved but there still remains room for further improvement. We are going to develop reusable specification component, as well as the program components.

References

[1] B.W. Boehm, "Improving software productivity," Computer, vol.20, no.9, pp.43–57, Sept. 1987.
[2] E.J. Chikofsky and B.L. Rubenstein, "CASE reliability engineering for information system," IEEE Softw., vol.5, no.2, pp.11–12, March 1988.
[3] M.A. Cusumano, Japan's Software Factories, Oxford University Press, New York, 1991.
[4] H. Komuro, O. Ohno, Y. Furuhata, K. Yasuda, A. Harada, and Y. Makuta, "Improving accuracy of estimation in a software development project with a specific program development automation," Proc. International Conference on Project Management, pp.461–469, 2002.
[5] H. Maezawa, M. Kobayashi, K. Saito, and Y. Futamura, "Interactive system for structured program production," Proc. 7th International Conference on Software Engineering, Orlando, FL, pp.162–171, 1984.
[6] Y. Matsumoto, An Overview of Japanese Software Factories, Japanese Perspectives in Software Engineering, pp.303–320, Addison-Wesley, Massachusetts, 1989.
[7] K. Morooka, Dynamic Analysis, Kenpakusha, Tokyo, March 1985.
[8] O. Ohno, Y. Furuhata, H. Komuro, T. Imajo, and S. Komiya, "Automated software development based on composition of categorized reusable components — Construction and sufficiency of skeletons for batch programs —," IEICE Trans. Inf. & Syst. (Japanese Edition), vol.J83-D-I, no.10, pp.1055–1069, 2000.
[9] M. Tsuda, Y. Morioka, M. Takadachi, and M. Takahashi, "Productivity analysis of software development with an integrated CASE tool," Proc. 14th International Conference on Software Engineering, pp.49–58, 1992.

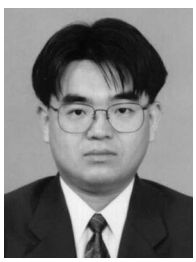
Michio Tsuda received the BE degree in electrical engineering from Doshisha University in 1970. He was a systems engineer of industrial field and developer for software engineering, Hitachi, Ltd., from 1970 to 1999. He is currently a senior chief engineer of Software Engineering Department, Hitachi Systems & Services, Ltd. He is a member of the IPSJ.

Sadahiro Ishikawa received the BE degree in Industrial and Systems Engineering at Aoyama Gakuin University in 1984. He is currently a department manager of Engineering Support Division, Hitachi, Ltd. He has been engaged in technological support of enterprise information system development since 1989.

Osamu Ohno graduated from Ube National College of Technology in 1969. He received the DE degree from Saitama University. He is currently a general manager at Information Technology Division, Hitachi, Ltd.

Akira Harada received the BS degree from Saitama University and the MS degree from Nagoya University. He had worked for Hitachi, Ltd. from 1975 to 2005. Currently, he serves in Japan Information Processing Service CO., Ltd. His research interests include project management and software development methodology. He is a member of the IPSJ and the SPM.

Mayumi Takahashi received the BE degree in psychology from Tokyo Metropolitan University in 1975. She has worked for Hitachi Systems and Services, Ltd. and developed the Computer Aided Software Engineering products. Currently Her research interests include business performance management, business modeling, and strategy. She is a member of the IPSJ.



Shinji Kusumoto received the BE, ME, and DE degrees in information and computer sciences from Osaka University in 1988, 1990, and 1993, respectively. He is currently an associate professor at Osaka University. His research interests include software metrics, software maintenance, and software quality assurance techniques. He is a member of the IEEE, the IEEE Computer Society, the IPSJ and the JFPUG.

Katsuro Inoue received the BE, ME, and DE degrees in information and computer sciences from Osaka University, Japan, in 1979, 1981, and 1984, respectively. He was an assistant professor at the University of Hawaii at Manoa from 1984–1986. He was a research associate at Osaka University from 1984–1989, an assistant professor from 1989–1995, and a professor from 1995. His interests are in various topics of software engineering such as software process modeling, program analysis, and software development environment. He is a member of the IEEE, the IEEE Computer Society, and the ACM.