NORTH-HOLLAND

Using the GQM Paradigm to Investigate Influential Factors for Software Process Improvement

Yasuhiro Mashiko
Matsushita Communication Industrial Co., Ltd., Japan

Victor R. Basili
Institute for Advanced Computer Studies and Department of Computer Science, University of Maryland

In planning software process improvement activities, it is essential to determine the factors that most influence the success of a software project. In this article, we present an investigative and analytical framework for evaluating software process factors based on the Goal/Question/Metric (GQM) paradigm. We built descriptive models of the software process, defects, and cost. These models were used as a common basis of quantitative analysis in the study. We also developed evaluative models that clarify the relationship between the basic metrics, the analysis method, and the goals of the analysis. We confirmed the usefulness of our analytical framework, by applying it in an actual development environment at Matsushita Communication Industrial Company in Japan, where we studied four communications-software projects. This article reports the patterns we noted in the data and suggests process improvement activities based on those findings.

1. INTRODUCTION

When planning improvements to the software process, it is essential to understand what process factors and customer needs influence product quality. Process improvement activities should be based on an understanding of the relationship between process characteristics, product characteristics, and customer requirements. By building models of the relationships between these factors, we can improve our understanding of the software process. Measurement is central in recognizing these relationships and in refining and validating them.

To improve a software process we must first specify our improvement goals and the criteria by which we will evaluate success. The goal may be, for example, an increase in customer satisfaction or a reduction in development cost. Specific goals depend on the needs of the organization. After organizational goals have been set and the corresponding criteria for evaluation have been defined, we can examine what factors influence those criteria. Based on the importance of each evaluation criterion, we can set priorities among the various possible improvement actions aimed at altering specific influential factors.

In this article, we report on a study of four software development projects at the Matsushita Communication Industrial Company in Japan. We studied the factors that influence product characteristics and business needs, and analyzed them, using the Goal/Question/Metric (GQM) paradigm and several other models (Basili and Weiss, 1981; Basili and Weiss, 1984; Basili and Selby, 1984; Basili and Rombach, 1988; Basili, 1992).

1.1. Purpose

In this study we analyzed the relationship between process and product characteristics. Our goal was to identify factors that influence customer satisfaction. We looked at three aspects of the projects, the type of defects, and relationship between defects and cost, and the product architecture, to answer the following questions.

Type of defect

- What types of defects are most observable to the customer?

Address correspondence to Dr. Y. Mashiko, Communication Industrial Co., Ltd., Japan.

J. SYSTEMS SOFTWARE 1997; 36:17–32
© 1997 by Elsevier Science Inc.
655 Avenue of the Americas, New York, NY 10010

0164-1212/97/$17.00
SSDI 0164-1212(95)00194-8

- During which phase are most of them injected?
- What kind of human error causes most of them?
- In what product part can most of them be found?

Relationship between defect and cost

- Which defects are the most costly to repair?

Product architecture

- What product architecture provides good customer satisfaction through initial product quality and ease/economy of defect correction and maintenance?

1.2. Approach

We used a four-step measurement, modeling, and analysis method, as outlined below.

1. Set a basis of quantitative analysis by building models.

   First, we built several descriptive and evaluative models.[1] Descriptive models explain various aspects of the software product and development environment, e.g., development process, defect, cost, and product architecture. Evaluative models define the criteria by which the product will be evaluated. These models lay the foundation for our analysis, and all evaluations, comparisons, and conclusions in this study are based on these models. We show these models in 2.1 and 2.2.
2. Define GQM models.

   Second, we specified, the structure of the analysis by defining the goals, questions, and metrics using the GQM paradigm. In this step we identified the metrics to be collected and how we would interpret them. We show these models in 2.3.
3. Look for patterns in the data.

   Third, we looked for patterns in the data, identifying patterns common to all projects and patterns specific to individual projects. In this step we determined which factors most influenced the process and product, based on the evaluative models defined in Step 1. We show the results of the evaluation and the patterns found in the data in 3.1 and 3.2, respectively.
4. Analyze how and where to improve the process and product design (based on the patterns found during Step 3).

   Finally, by quantitatively analyzing the patterns of product and process characteristics and the factors influencing them, we were able to predict, with varying levels of confidence, the effectiveness of possible improvement actions suggested by the patterns. We show them in 3.3.

1.3. Projects Studied

We studied four projects, referred to throughout the article as projects, A, B, C, and D. All are development projects for communications software having functionality in data communication, data entry by interactive human interface, and data output to peripheral devices. These products were developed as deliverables for customer contracts.

Three different product architectures were applied on the four projects: projects A and D used a *control matrix* model; project B used a *transactions switch* model; and project C used a *message-driven model.* Only project D had severe memory constraints.

In the control matrix architecture, the operation is described as a finite state machine in the form of a matrix. A product consists of a matrix control subsystem and a resource management subsystem. The matrix control subsystem manages the entire product operation by referencing the matrix. The resource management subsystem provides the matrix control subsystem with primitive functions to control resources like the display, keyboard, file, and communication device. In the transaction switch architecture, the operation is described as a set of transactions. A product consists of transaction control units and a resource management subsystem, similar to the one used in the control matrix architecture. Each transaction control unit provides its services by means of primitive functions in the resource management subsystem. In the message driven architecture, operation is described as message flow between resource control units. A product consists of these resource control units. No part of the product provides central operational control. Any of these three different product architectures could have been chosen for the four projects studied here. The product model for each project was selected by the projects themselves, based upon the decision of the engineers involved in the projects.

There is a common, standard development process model applied to all four projects. It consists of seven phases: requirement analysis, specification definition, software design, implementation, including unit testing, integration testing, system testing, and acceptance testing. For requirement analysis,

[1] Note that we use the word "model" in a rather broad sense. In this text, it represents an abstraction of the relationship among a set of variables; it does not necessarily imply predictive capability or include causal relationships between elements, as is sometimes the case.

specification definition, and the design phase, the development method is not formally specified. However, the development teams were familiar with the structured analysis and structured design technique and applied it to analyze and design the software products. The abstract process cycle model is shown in 2.1.1.

Each development team of the four projects had more than three years experience of the application domain, the development process, and the development method.

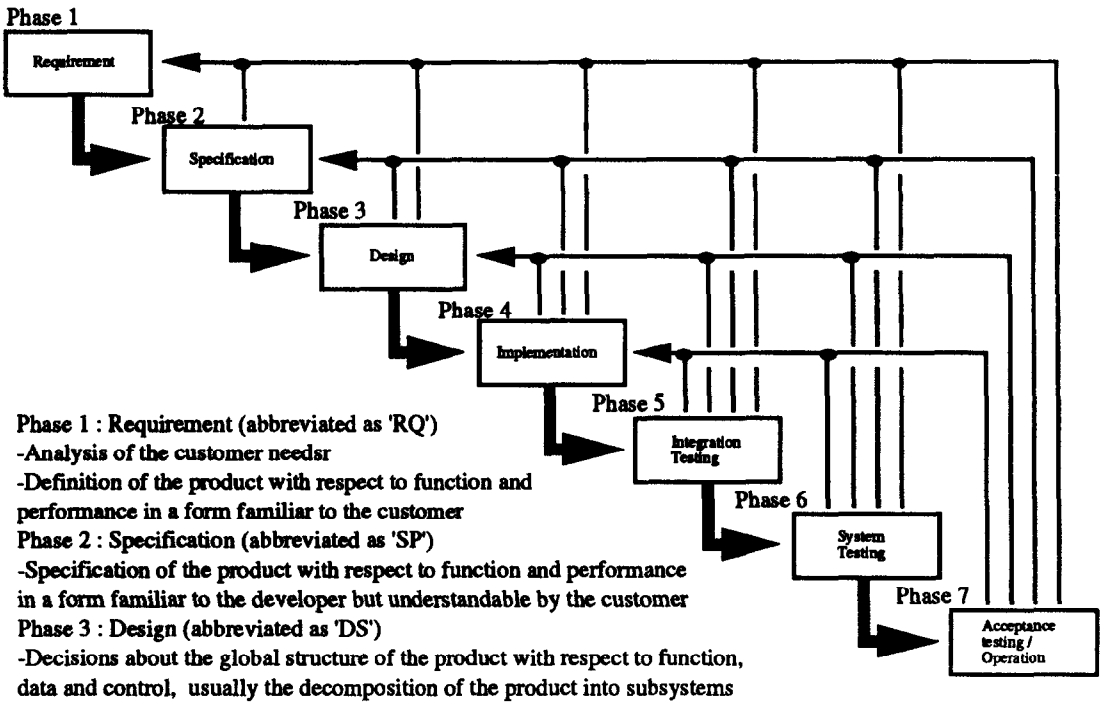
2. ANALYTICAL FRAMEWORK

The analytical framework for the study is composed of the descriptive and evaluative models described in the sections that follow and the set of goals, questions, and metrics (GQMs) outlined in Section 2.3.

2.1. Descriptive Models

We built descriptive models of process, defect, and cost.

2.1.1. Process model. Figure 1 shows the process model we applied to all of the projects analyzed in this study. In the model, project life-cycle activities are divided into chronological phases categorized as either *construction* or *testing* phases. Construction phases are those in which the product (including all documents except test specification) is created or changed. Testing phases are those in which the



Phase 1 : Requirement (abbreviated as 'RQ')
-Analysis of the customer needsr
-Definition of the product with respect to function and performance in a form familiar to the customer
Phase 2 : Specification (abbreviated as 'SP')
-Specification of the product with respect to function and performance in a form familiar to the developer but understandable by the customer
Phase 3 : Design (abbreviated as 'DS')
-Decisions about the global structure of the product with respect to function, data and control, usually the decomposition of the product into subsystems
-Decisions about the details of each subsystem, usually the decomposition of each subsystem into program modules
Phase 4 : Implementation (abbreviated as 'IP')
-Coding of each module
-Unit testing of each module
Phase 5 : Integration Testing (abbreviated as 'IT')
-Integration of modules into subsystems
-Testing of each integrated subsystem
-Integration of subsystems into the product
Phase 6 : System Testing (abbreviated as 'ST')
-Verification of the correctness of the product against its specification
Phase 7 : Acceptance Testing / Operation (abbreviated as 'AT')
-Validation of the acceptability of the product against its requirement
-Operation of the customer organization with the developed product

Figure 1. Process model.

product is verified or validated. In the process model, phases 1–4 (requirement, specification, design, implementation) are construction phases, while phases 5–7 (integration testing, system testing, acceptance testing/operation) are testing phases.

In Figure 1, thin arrows indicate defects found; the origin of the arrow indicates the phase during which a defect is detected, and its end indicates the phase during which a defect was injected.

2.1.2. Defect Model

(1) Definition and Description of Defect

We define *defect* as a factor that requires changes to be made to the software product, including documents as well as program code. We consider any change to the product a defect until the product becomes acceptable to the customer. According to this definition, any change of requirement or specification by the customer is considered a defect. Our definitions of defect, error, and fault conform to the IEEE/ANSI standard (IEEE982.2).

In this study, we intended to analyze the developer's capability to communicate with the customer during requirement and specification phases. We needed this kind of analysis because our goal was to clarify those factors that most influence customer satisfaction. From the viewpoint of customer satisfaction, it is essential to take all changes into account and analyze how well the product meets the real needs of the customer. For this purpose, we classified all changes as shown in (2) in this section.

Here, we use the work "communication" in the broad sense. In the requirement and specification phases, the developer defines what the product should do through a communication process with the customer. If the communication capability of the developer were perfect, the product would need no change in its requirement and specification after delivery unless the operational environment of the product changes or the customer was mistaken with respect to understanding of the original organizational needs. Various factors such as experience in the application domain improve communication capability.

We use the following notation for defects

$$\mathbf{D_n(d1_n, d2_n, d3_n, d4_n; Cd_n, Sd_n, Md_n)}$$

where **D** is defect; **d1**, **d2**, **d3**, and **d4** are defect parameters (defined below); **Cd**, **Sd**, and **Md** are defect attributes (defined below); and **n** is a unique identifying number.

Defect parameters.

d1 Injection phase

This parameter denotes the phase during which the defect was injected. The value of d1 is either RQ (Requirement), SP (Specification), DS (Design), or IP (Implementation).

d2 Detection phase

This parameter denotes the phase during which the defect was detected. The value of d2 is either RQ, SP, DS, IP, IT (Integration Testing), ST (System Testing), or AT (Acceptance Testing).

d3 Error type

This parameter denotes the type of human error that caused the defect. We categorized errors into those of *omission* and those of *commission* based on basic properties of human misunderstandings. Moreover, we further categorized errors that caused defects during the requirement and specification phases into *logic* and *communication* errors.

- A logic error causes an apparent logical contradiction or deficiency that can be detected by examination of documents without any knowledge of the application domain or special features of the environment in which the product is operated.
- A communication error is caused by poor communication between the customer and the developer; it has no logical contradiction or deficiency that can be detected by examination of documents but causes some trouble in the actual operation of the product by the customer. Defects caused by a communication error can be detected only by knowledge of the application domain and the environment in which the product is operated.

This distinction between logic and communication errors is applicable only in the requirement and specification phases. All defects injected during design or implementation phases are considered logic errors. The possible values for d3 are, for RQ and SP phases:

LO logic omission
LC logic commission
CO communication omission
CC communication commission

For DS and IP phases:

Om omission
Cm commission

Table 1. Fault Type

Fault type	Abbreviation	Definition
Global structure	Gstr'	Fault of relationships among subsystems
Data structure	Dstr	Fault of structure of files, tables or other data, including fault of data size
Algorithm	Algr	Fault of algorithm inside program module
Human interface	Hitr	Fault of human interface
External interface	Eitr	Fault of interface between the product and its external system
Internal interface	Iitr	Fault of interface between modules
Initialization	Init	Omission or commission of initialization of data entry
Constant value	Cnst	Fault of definition of constant value

d4 Fault type

According to the IEEE/ANSI standard, a *fault* is a concrete manifestation of an error. Table 1 shows the fault categories we used.

Defect attributes.

Cd Cost of defect

We define *cost of defect* as the direct expense to fix a given defect completely. It includes the costs listed below but does not include indirect costs such as management, facilities, and installation of the corrected program.

- Cost to correct documents and source code
- Cost to recreate new executable program from corrected source code
- Cost to confirm the corrected function and to regression test other portions of the product that may have been affected by the correction

When a defect is not corrected properly in the first attempt, subsequent iterations of the correct must be done until the defect is eliminated. In this study, we did not count iterations on the same defect as new defects. Therefore, the cost of the original defect includes all iterative attempts to fix it.

Sd Severity of defect

We categorized *severity of defect* into four levels and assigned a value to each level, as shown in Table 2.

Md Number of modules affected

This parameter denotes the number of modules that need to be corrected to fix a defect. Because a defect detected during a construction phase does not require correction of the source code, we define the value of Md for a construction-phase defect as 0 (e.g., the value of Md is 0 for a defect injected during specification and detected during design).

2) Classification of Defect

We classified defects as follows:

Developer-findable defect

Logic-based defects are detectable by any developer without special capability, while communication-based defects are detectable only with knowledge of the application domain and the customer's specific needs (i.e., product's operational environment). Hence, we consider those two types separately when we analyze software process and product. A *developer-findable defect* satisfies the following conditions:

$$D_n(d1_n = RQ/SP, d2_n = *, d3_n = LO/LC, d4_n = *; Cd_n, Sd_n, Md_n) \quad \text{or}$$

$$D_n(d1_n = DS/IP, d2_n = *, d3_n = *, d4_n = *; Cd_n, Sd_n, Md_n).$$

An asterisk here means that any value is acceptable for the attribute, i.e., it is not used for classification.

Table 2. Severity of Defect

Level	Name	Definition
4	Critical	Without fixing a defect at this level, delivery of the product to the client is impossible.
3	Essential	Without fixing a defect at this level, operation is possible by altering normal operational procedures for the system. It must be fixed as soon as possible.
2	Important	Without fixing a defect at this level, normal operation is impossible. However, the efficiency of system operation is improved by fixing it.
1	Desired	A defect at this level does not cause trouble in the efficiency of operation. However, it is desirable to fix it from the point of view of the product's impression on the user.

This is a defect detected before or after delivery that may or may not be observed by the customer.

Customer-observable defect

A *customer-observable defect* satisfies the following conditions

$D_n(d1_n = *, d2_n = AT, d3_n = *, d4_n = *;$

$Cd_n, Sd_n, Md_n)$

Customer-observable defects are further divided *into developer-findable defect observed by customer* or *addition or change* depending on whether the error is logic- or communication-based.

Developer-findable defect observed by customer

This class of defect satisfies the following conditions

$D_n(d1_n = RQ/SP, d2_n = AT, d3_n = LO/LC,$

$d4_n = *; Cd_n, Sd_n, Md_n)$ or

$D_n(d1_n = DS/IP, d2_n = AT, d3_n = *,$

$d4_n = *; Cd_n, Sd_n, Md_n).$

Addition or change

Sometimes during the acceptance testing phase, the customer may require additions or changes to the original requirement or specification. Although ideally all such defects would have been detected and corrected through effective communication between the customer and the developer during the requirement and specification phases, this type of defect does occur. An *addition or change* class defect satisfies the following conditions

$D_n(d1_n = RQ/SP, d2_n = AT, d3_n = CO/CC,$

$d4_n = *; Cd_n, Sd_n, Md_n)$

2.1.3. Cost model. We built a defect cost-to-fix model based on the process model and defect definitions outlined in the previous sections. We use the following notation to document cost

$\mathbf{CD_n}$ Cost spent to fix defect D_n

$\mathbf{RE_i(D_n)}$ Cost spent for reworking in phase i to fix defect D_n.

The value is 0 when the injection of D_n is after phase i or when the detection of D_n is before phase i.

Using this notation, we generated the following model. The cost of rework for defect D_n is the sum of the costs for reworking individual phases necessary to fix the defect.

$$\text{Cost of rework for defect} = CD_n = \sum_{i=d1_n}^{d2_n} RE_i(D_n).$$

We applied this cost model to measure the cost of each defect.

2.2. Evaluative Models

We assessed customer satisfaction using the four criteria defined below. The evaluative models are based on product properties after delivery and therefore are observable to the customer. We found that intermediate product properties (before delivery) generally have little bearing on customer satisfaction, and therefore, we did not include those factors in our model.

2.2.1. Reliability. Reliability was evaluated on the number of customer-observable defects in the delivered product and on the severity class of those defects. Figure 2 illustrates our evaluative model of reliability.

2.2.2. Reparability. Availability of the product is an important factor in evaluating customer satisfaction. Product availability depends on both mean time between failure (MTBF) and mean time to repair (MTTR). MTBF depends mainly on the number of customer-observable defects, while the MTTR depends on the ease of repairing those defects. Although we should consider availability alone, here we considered only reparability because it is difficult to predict the MTBF of a newly developed product.

Reparability of product was evaluated on the average cost of rework per defect. Figure 3 illustrates our evaluative model of reparability. Cost of rework was calculated using the cost model shown in Section 2.1.3.

2.2.3. User-Friendliness. In our defect classification, we divided customer-observable defects into two categories: developer-findable defect observed by customer and addition or change. In evaluating user-friendliness, we considered addition or change class defects separately to evaluate how well the

Reliability evaluation without regard to defect severity
5 : N1 = Much smaller than average
4 : N1 = Smaller than average
3 : N1 = Near to average
2 : N1 = Larger than average
1 : N1 = Much larger than average
N1 : Number of customer observable defects

Reliability evaluation with regard to defect severity
5 : N2 = Much smaller than average
4 : N2 = Smaller than average
3 : N2 = Near to average
2 : N2 = Larger than average
1 : N2 = Much larger than average
N2 : Number of customer observable severe defects
Severe defect : defect whose severity is 4 or 3.

Figure 2. Evaluative models of reliability.

Reparability evaluation
5 : C1 = Much smaller than average
4 : C1 = Smaller than average
3 : C1 = Near to average
2 : C1 = Larger than average
1 : C1 = Much larger than average
C1: Average cost of rework for customer observable defect

Figure 3. Evaluative model of reparability.

product conforms to the real needs of the customer. The number of defects of this class is closely linked with the degree to which the development organization understands the application domain and the customer's needs. We built an evaluative model for user-friendliness which considers both the number and severity level of defects of this class in the product. Figure 4 illustrates our evaluative model of user-friendliness.

2.2.4. Maintainability. Once a product goes operational, it must be updated and changed in accordance with changes in its operational environment. On many projects, maintenance costs represent a large part of the total cost spent throughout the product's lifetime. Hence, ease of maintenance is one of the central concerns of the customer.

The long-term maintainability of the product can be estimated by the average cost of rework for addition or change class defects detected during acceptance testing. This is because the number of logic defects detected declines sharply shortly after the beginning of operation for regularly managed projects. We evaluated the product maintainability by the average cost of rework for addition or change class defects. Figure 5 illustrates our evaluative model of product maintainability.

2.3. GQMs

We set the goals, questions, and metrics for this analysis based on the GQM paradigm. Although we set many questions and metrics, we show here only those that relate to our discussion of our goals stated in Section 1.1.

(1) Goal of Analysis

The goal of this analysis was to determine influential factors on the software process and to identify pro-

User friendliness evaluation without regard to severity of defect
5 : N3 = Much smaller than average
4 : N3 = Smaller than average
3 : N3 = Near to average
2 : N3 = Larger than average
1 : N3 = Much larger than average
N3 : Number of addition or change

User friendliness evaluation with regard to severe defect
5 : N4 = Much smaller than average
4 : N4 = Smaller than average
3 : N4 = Near to average
2 : N4 = Larger than average
1 : N4 = Much larger than average
N4 : Number of addition or change
Severe defect : defect whose severity is 4 or 3

Figure 4. Evaluative models of user-friendliness.

Maintainability evaluation
5 : C2 = Much smaller than average
4 : C2 = Smaller than average
3 : C2 = Near to average
2 : C2 = Larger than average
1 : C2 = Much larger than average
C2: Average cost of rework for addition or change

Figure 5. Evaluative model of maintainability.

cess improvement activities likely to affect those factors. From this overall goal, we derived two subgoals, evaluation and characterization, as illustrated in Figure 6.

(2) Questions of Interest

We devised questions relevant to our analysis goals and grouped them into three categories: product-related questions, process-related questions, and improvement-related questions. Figure 7 shows the questions of interest for each category.

- Product-related questions are for describing various aspects of the product, e.g., logical/physical attributes, context of operation, product model, cost, defects, and validity of the data collected.
- Process-related questions are for describing various aspects of the process, e.g., process model, method, technique, process conformance, domain understanding, and validity of the data collected.
- Improvement-related questions are for identifying and analyzing the influential factors on process improvement.

(3) Metrics of Interest

We selected metrics to answer our questions. Figure 8 shows the metrics of interest.

III. DATA AND RESULTS

Using the analytical framework and the models described in Section 2, we examined the four projects and obtained the following data and results.

Goal		
Purpose:	Analyze	software development systems
	for the purpose of	improvement
Perspective:	with respect to	customer satisfaction
	from the point of view of	the customer
Subgoal 1 - Evaluation		
Purpose:	Analyze	software development systems
	for the purpose of	evaluation
Perspective:	with respect to	customer satisfaction
	from the point of view of	the customer
Subgoal 2 - Characterization		
Purpose:	Analyze	software development systems
	for the purpose of	characterization
Perspective:	with respect to	factors affecting the results of evaluation
	from the point of view of	the company

Figure 6. Goal of analysis.

Product-related questions
Q1.1 : What is the size of the product?
Q1.2 : What is the application domain?
Q1.3 : What are the hardware characteristics?
Q1.4 : What is the purpose of the product for the customer?
Q1.5 : What is the external system with which the product communicates?
Q1.6 : What is the total number of defects?
Q1.7 : What is the distribution of number of defects by defect class?
Q1.8 : What is the distribution of average cost to fix a defect by defect class?
Q1.9 : What is the distribution of defect severity by defect class?
Process-related questions
Q2.1 : What is the process model applied to the development?
Q2.2 : What methods and techniques are applied to each phase in the process model?
Questions to improve process
Q3.1 : What factors affect the number of customer observable defects?; Are there any noticeable patterns?
Q3.2 : What factors affect the number of customer observable severe defects?; Are there any noticeable patterns?
Q3.3 : What factors affect the cost of rework for customer observable defects?; Are there any noticeable patterns?
Q3.4 : What factors affect the number of additions or changes?; Are there any noticeable patterns?
Q3.5 : What factors affect the number of severe additions or changes?; Are there any noticeable patterns?
Q3.6 : What factors affect the cost of rework for additions or changes?; Are there any noticeable patterns?

Figure 7. Questions of interest.

3.1. Results of Evaluation

First we rated the projects using the evaluative models of reliability, reparability, user-friendliness, and maintainability. The number of each class of defect was normalized by the project size-measured as the number of lines of source code in the delivered product, not including comments.

In addition to examining the raw values of normalized number of defects and average cost of rework, we rated each project on a five-grade scale (where 1 is the minimum and 5 is the maximum) to visualize general tendencies. The rating was done by calculating the z score of each project using the following formula

$$z = \frac{x - \bar{x}}{s}$$

where s is the estimate of standard deviation and $\bar{x}$ is the estimate of mean. Table 3 shows the rating rule. When the distribution is the standard normal distribution, each rating interval includes 20% of all projects. Because these estimates were done for only the four projects, and not for a large number of past projects, we consider this evaluation only a comparison among the sample projects.

Related to entire project
M1.1 : Total size of code (LOC)
M1.2 : Programming language
Related to defect
M2.1 : Distribution of number of defects by defect class
M2.2 : Distribution of cost of defect, Cd, by defect class
M2.3 : Distribution of average cost of defect, E[Cd], by defect class
M2.4 : Distribution of severity of defect, Sd, by defect class
Customer observable defect
M3.1 : Distribution of number of customer observable defects by injection phase, error type, fault type
M3.2 : Distribution of number of customer observable severe defects by injection phase, error type, fault type
M3.3 : Distribution of cost of rework for customer observable defects by injection phase, error type, fault type
M3.4 : Distribution of cost of rework for customer observable defects by rework phase
Addition or change
M4.1 : Distribution of number of additions or changes by injection phase, error type, fault type
M4.2 : Distribution of number of severe additions or changes by injection phase, error type, fault type
M4.3 : Distribution of cost of rework for additions or changes by injection phase, error type, fault type
M4.4 : Distribution of cost of rework for additions or changes by rework phase

Figure 8. Metrics of interest.

Table 3. Division of Interval of Z, and assignment of rating value

range of z	rating
$-0.84 \geq z$	5
$-0.25 \geq z > -0.84$	4
$0.25 \geq z > -0.25$	3
$0.84 \geq z > 0.25$	2
$z > 0.84$	1

Table 4 summarizes the evaluation results. The projects have diverse ratings across the evaluation criteria. Except for project B, which rates very high overall, no one project is simply good or bad in every category.

For project A from the point of view of the customer, its reliability and user-friendliness are very poor, while its reparability and maintainability are very good. This implies that the product is likely to keep up with various changes in the operational environment although its initial quality seems poor. Hence, it is reasonable to expect that the product will satisfy the customer in the long run despite its poor initial quality evaluation.

For project B, the results of evaluation are good on the whole. Its reliability and user-friendliness are evaluated best among the four projects. We can expect that its good initial quality satisfies the customer. Its reparability and maintainability are also good, but not as good as project A. We can expect that the product will satisfy the customer both in the short term and in the long run as well.

For project C, its reliability is poor and user-friendliness is fair. Therefore, its initial quality may not satisfy the customer. On the other hand, because its reparability is good and maintainability is fair, we can expect that the product will keep up fairly well with changes required in the operational environment and may prove more satisfactory in the long run.

For project D, its reliability is good, and user-friendliness is not necessarily bad. The number of additions or changes due to severe defects (N4) is small. This indicates that perhaps most of the additions and changes were made to enhance the product, not to eliminate significant defects. Hence, it is reasonable to conclude that project D's initial quality is good. Its reparability and maintainability, however, are the worst among the four projects, indicating that it will be difficult for the project to keep up with the changes required in the operational environment. Therefore, it is unlikely that the product will satisfy

Table 4. Summary of Evaluation

Evaluation criteria		Project A	Project B	Project C	Project D	Estimate $\bar{x}$	Estimate SD
Reliability	N1/KLOC	1.40	0.25	0.95	0.67	0.82	0.48
(without regard	Value of z	1.21	−1.17	0.27	−0.30	—	—
to severity)	Rating	**1**	**5**	**2**	**4**	—	—
Reliability	N2/KLOC	0.90	0.15	0.74	0.17	0.49	0.39
(severe defect)	Value of z	1.06	−0.88	0.96	−0.83	—	—
	Rating	**1**	**5**	**2**	**4**	—	—
Reparability	C1	8.58	9.80	10.46	22.63	12.87	5.55
	Value of z	−0.65	−0.47	−0.37	1.49	—	—
	Rating	**4**	**4**	**4**	**1**	—	—
User friendliness	N3/KLOC	0.48	0.05	0.39	0.67	0.40	0.26
(without regard	Value of z	0.31	−1.34	−0.02	1.05	—	—
to severity)	Rating	**2**	**5**	**3**	**1**	—	—
User friendliness	N4/KLOC	0.42	0.05	0.25	0.17	0.22	0.16
(severe defect)	Value of z	1.26	−1.11	0.20	−0.36	—	—
	Rating	**1**	**5**	**3**	**4**	—	—
Maintainability	C2	10.65	14.00	15.06	22.63	15.59	5.06
	Value of z	−0.98	−0.31	−0.10	1.39	—	—
	Rating	**5**	**4**	**3**	**1**	—	—

the customer in the long run in spite of its initial good quality. This is just the opposite of project A.

We drew the following lessons from these findings:

1) Importance of evaluating projects from various viewpoints

We evaluated projects from various viewpoints based on the GQM paradigm, with a wide range of results, as illustrated in Table 4. This diversity indicates that it is not only insufficient, but dangerous, to analyze a project by a single criterion, such as number of defects. For example, project D has good quality in the context of the number of defects but poor maintainability, while project A has poor quality but good maintainability.

To improve software processes efficiently and rapidly, it is critical to accumulate experiences effectively and to integrate them as a whole. Hence, we tried to draw the maximum number of lessons, even from projects that seem to be failures. In this study, by closely analyzing the projects that seem to have poor quality (projects A and C), we can see that these projects may rate differently when evaluated on different criteria (e.g., quality and maintainability).

2) Relationship between product quality and product maintainability

From Table 4, we cannot observe any pattern between quality (measured as the number of defects) and maintainability (measured as least cost to fix). At the beginning of operation of a newly developed system, customers are apt to evaluate the product by the number of initial customer-observable defects. However, in the long run, another property, such as maintainability, may become more important to the customer.

3) Associated factors

Overall, product reliability and user-friendliness seem to be affected mainly by human factors, such as process conformance and domain understanding. Product reparability and maintainability seem to be affected more by other factors, such as methods, techniques, memory constraints, and product architecture.

3.2. Influential Factors

After the initial evaluation of reliability, reparability, user-friendliness, and maintainability, we sought to identify factors that appeared to have influenced each of these evaluation criteria. We classified influential factors in two groups: those common to all projects and those specific to individual projects. In the sections that follow, we discuss the common factors and the project-specific factors as they affect each of the evaluation criteria.

3.2.1. Common Factors. Influential factors common to many projects usually stem from general properties of the development organization and the application domain, e.g., level of personnel, management strategy, process model characteristics, peculiarities of the application domain. Hence, these factors can be especially useful when identifying activities to improve the overall project organization.

A Reliability. We set the number of customer-observable defects as the measure of reliability. Questions 3.1 and 3.2 in Figure 7 identify the factors that influence this number. Figures 9 and 10 summarize the data for those questions.s Figures 9.1–9.3 show, respectively, the distribution of defects by injection phase, error type, and fault type. Figures 10.1–10.3 show, respectively, the percentage of severe defects by injection phase, error type, and fault type.

These patterns of interest were noted:

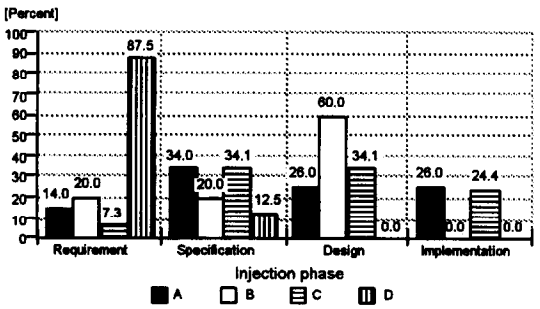


Figure 9.1. Distribution of customer-observable defect by injection phase.
■ A, □ B, □ C ■ D

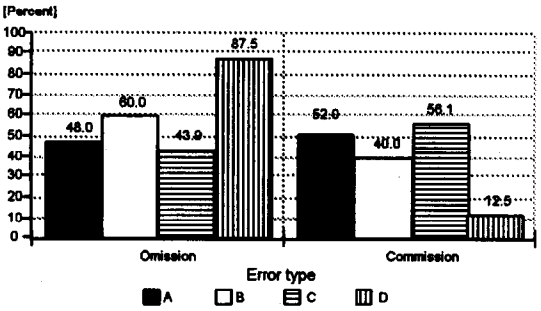


Figure 9.2. Distribution of customer-observable defect by error type.

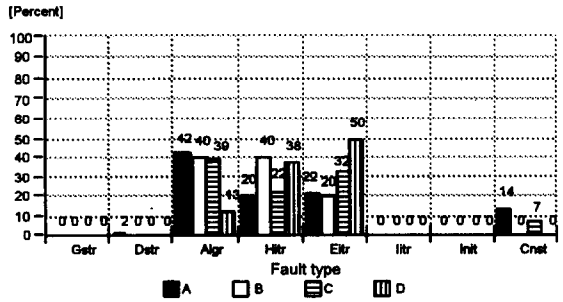


Figure 9.3. Distribution of customer-observable defect by fault type.

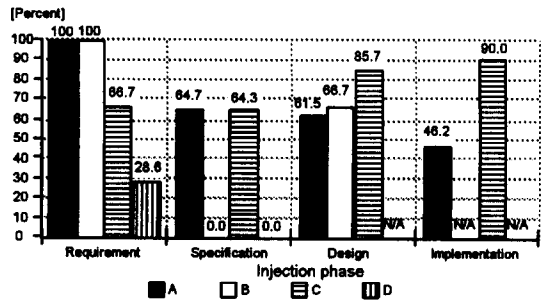


Figure 10.1. Distribution of proportion of severe customer-observable defect by injection phase.

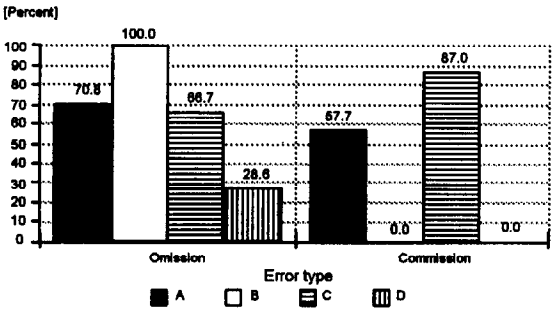


Figure 10.2. Distribution of proportion of severe customer-observable defect by error type.

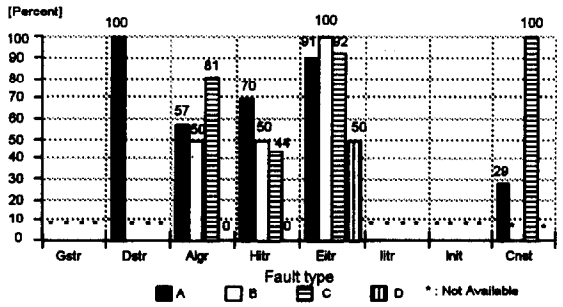


Figure 10.3. Distribution of proportion of severe customer-observable defect by fault type.

- **A1:** The fault type for more than 80% of the defects is either algorithm, human interface, or external interface (Figure 9.3). For project D, the proportion of algorithm type faults is smaller than the others. This is probably because review processes during design and implementation were effective performed (Figure 9.1), and defects of algorithm fault type in these two phases were removed before the customer used the product. For the other three projects, on the other hand, more than half of customer observable defects were injected during design and implementation.

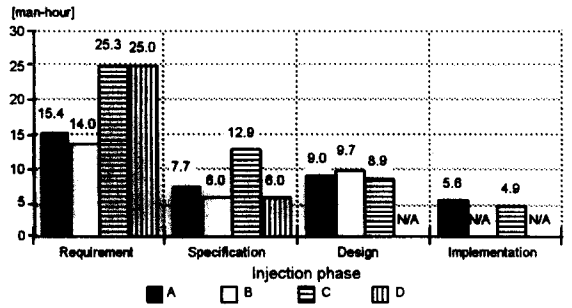


Figure 11.1. Distribution of average cost of rework for customer-observable defect by injection phase.

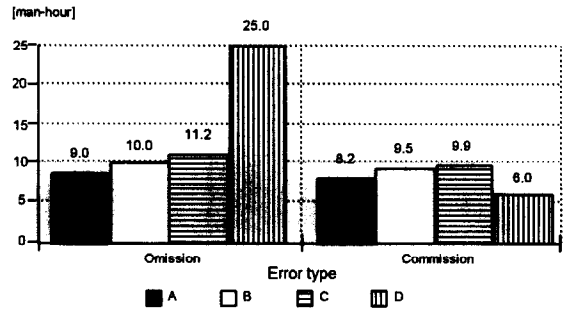


Figure 11.2. Distribution of average cost of rework for customer-observable defect by error type.

For all projects, most defects of fault type other than algorithm, human interface and external interface, are removed before the customer began to use the project in this development environment.

- **A2:** The highest proportion of severe defects is among those caused by an external interface fault. More than 90% of the external interface fault defects for projects A, B, and C, are severe. For project D, the proportion is 50%, but *all* of project D's severe defects were due to *external interface* faults (Figure 10.3).
- **A3:** Errors of omission represent at least 40% of the defects (Figure 9.2). Except for project D, they are mostly categorized as severe (Figure 10.2). Project D has a lower severe defect rate than the others (Table 4).

B Reparability. We set the average cost of rework for a customer-observable defect as the measure of reparability. Question 3.3 in Figure 7 identifies the factors that influence this number. Figure 11 summarizes the data for this question. Figure 11.1–11.3 show, respectively, the distribution of the average cost of rework for this class of defect by injection phase, error type, and fault type. Figure 11.4 shows

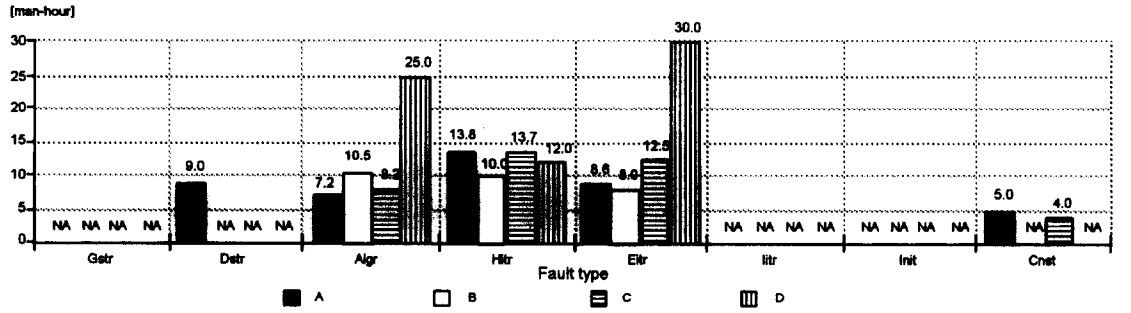


Figure 11.3. Distribution of average cost of rework for customer-observable defect by fault type.

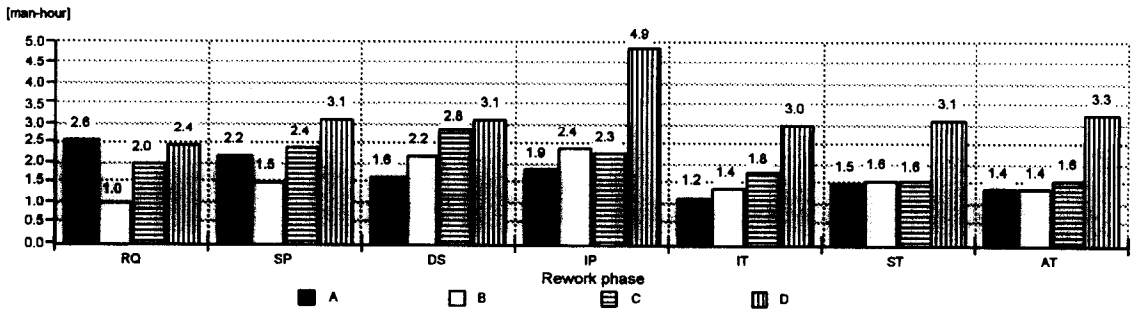


Figure 11.4. Distribution of average cost of rework for customer-observable defect by rework phase.

the distribution of average cost of rework for this class of defect among rework phases. Based on the cost model definition, the sum of average costs for each rework phase is not necessarily equal to the average cost of rework for a class of defect.

These patterns of interest were noted:

- **B1:** A defect injected during the requirement phase costs at least 44% more to fix than an error injected during any other phase (Figure 11.1).
- **B2:** A defect caused by an omission error is more costly to fix than a defect caused by a commission error. The difference is from 5% to 13% for projects A, B, and C, while it is more than 400% for project D (Figure 11.2).

C User-Friendliness. We set the number of addition or change class defects as the measure of user-friendliness. (It is possible in actual business environments that requests for addition or change are desirable but not practical because of their very high costs. We should take this factor into account when examining user-friendliness. For example, we may consider a software product not user-friendly if it has a small number of addition or change defects throughout its lifetime because it requires too much effort to make even small additions or changes.) In our study, however, very few requests for addition or change by the customer were impossible because of high cost. Therefore, we consider the evaluative model of Figure 4 as valid in this study. Questions 3.4 and 3.5 identify the factors that influence this number. Figures 12.1–3 and 13.1–3 summarize the data for these questions. Figures 12.1–12.3 show, respectively, the distribution of additions or changes by injection phase, error type, and fault type. Figures 13.1–13.3 show, respectively, the percentage of severe defects for each division of addition or change by injection phase, error type, and fault type.

These patterns of interest were noted:

- **C1:** More than 60% of addition or change class defects are caused by omission errors (Figure 12.2).
- **C2:** More than 80% of addition or change class defects are due to *human interface* or *external interface* faults (Figure 12.3).

D Maintainability. We set the average cost of rework for addition or change class defects as the measure of maintainability. Question 3.6 in Figure 7 identifies the factors that influence this number. Figures 14.1–4 summarize the data for this question.

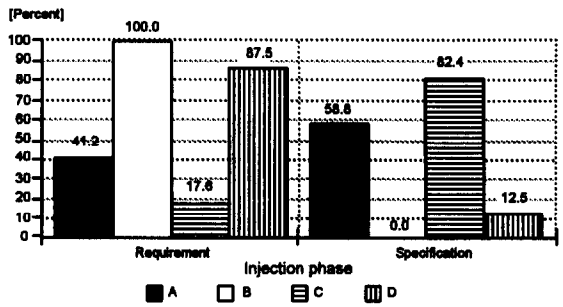


Figure 12.1. Distribution of addition or change by injection phase.

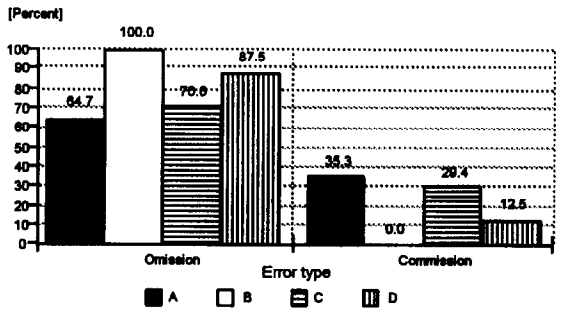


Figure 12.2. Distribution of addition or change by error type.

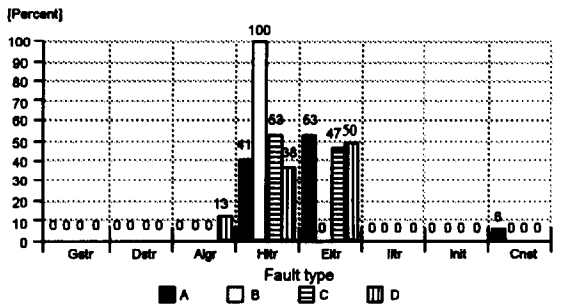


Figure 12.3. Distribution of addition or change by fault type.

Figures 14.1–14.3 show, respectively, the distribution of the average cost of rework for this class of defect by injection phase, error type, and fault type. Figure 14.4 shows the distribution of average cost of rework for this class of defect among rework phases. Again, note that the sum of the average costs for each rework phase is not necessarily equal to the average cost of rework for a class of defect.

This pattern of interest was noted:

- **D1:** An addition or change injected during the requirement phase costs roughly twice as much (or more) to fix than one injected during specification (Figure 14.1).

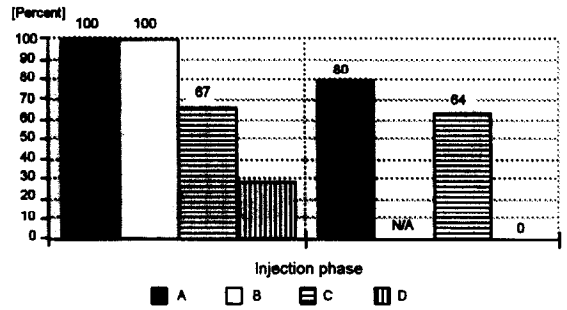


Figure 13.1. Distribution of proportion of severe addition or change by injection phase.

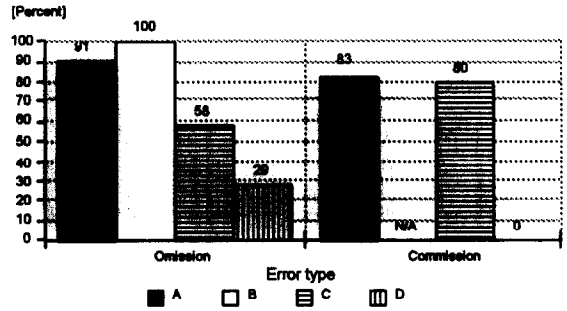


Figure 13.2. Distribution of proportion of severe addition or change by error type.

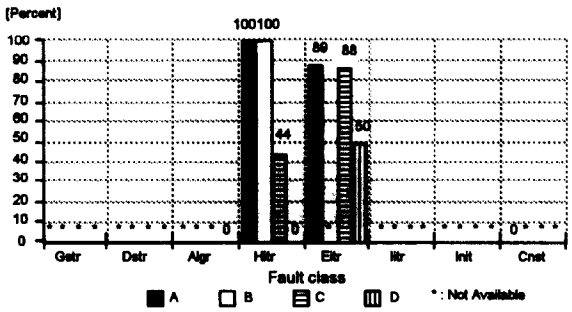


Figure 13.3. Distribution of proportion of severe addition or change by fault type.

3.2.2. Project-Specific Factors. By analyzing the patterns particular to a given project, we can assess the strength or weakness of various aspects of its development environment, e.g., method, technique, personnel, and management approach. This knowledge can influence the processes that may be used on future projects with similar characteristics. We observed the following project-specific patterns in the four projects. the reasons for the resulting patterns will be discussed in Section 3.3.

Project A

PA1: The cost of rework on phases after the design phase is at least 20% less for project A then for any other project (Figures 11.4 and 14.4). We discuss the reason in 3.3, (3).

Project B

PB1: There are no addition or change class defects due to an external interface fault for project B (Figure 12.3) because the external interface was same as an old system. Therefore, special design consideration of external interface was not necessary.

Project C

PC1: Cost of rework for addition or change class defects caused by a commission error for project C is at least double the average for any other project (Figure 14.2). We surmise the reason is product architecture. We discuss the reason in 3.3, (3).

Project D

PD1: Cost of rework for a defect caused by an omission error for project D is approximately double the average for any other project (Figures 11.2 and 14.2).

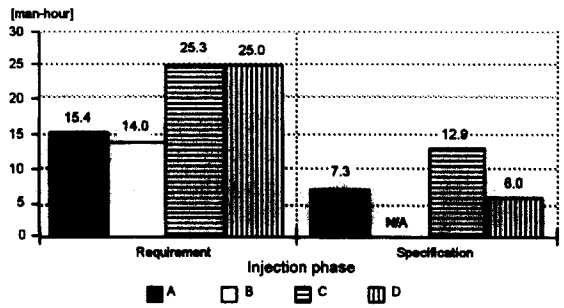


Figure 14.1. Distribution of average cost of rework for addition or change by injection phase.

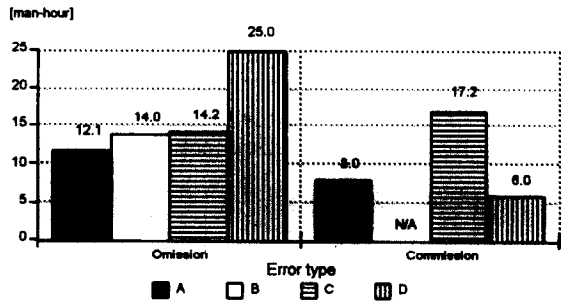


Figure 14.2. Distribution of average cost of rework for addition or change by error type.

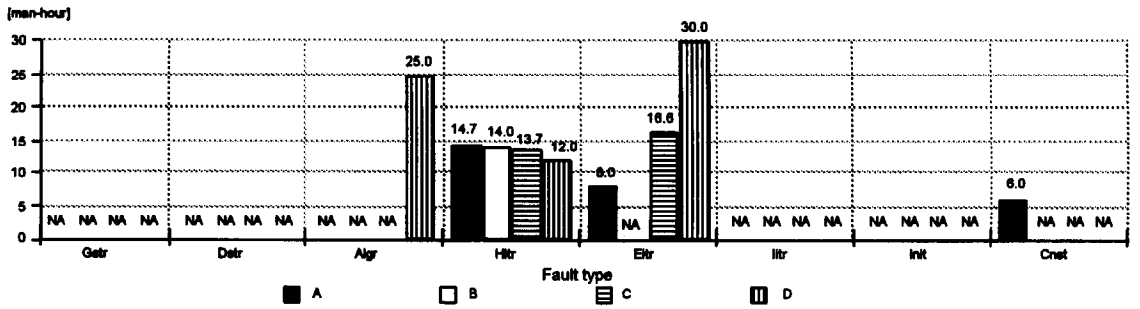


Figure 14.3. Distribution of average cost of rework for addition or change by fault type.

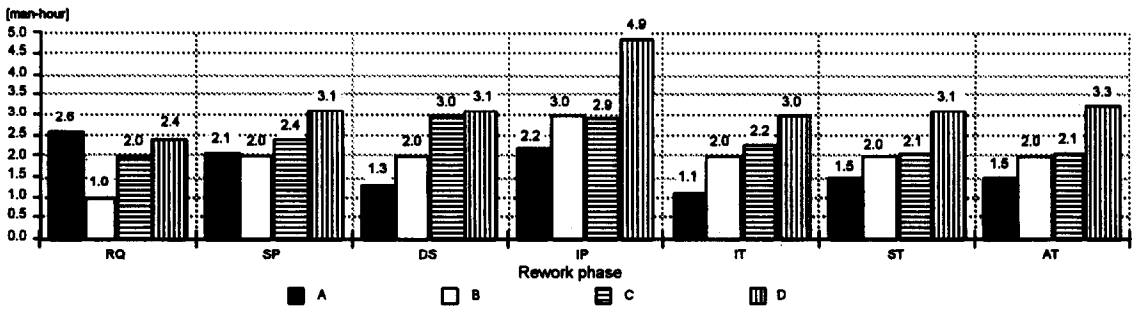


Figure 14.4. Distribution of average cost of rework for addition or change by rework phase.

PD2: Cost of rework after the implementation phase is much higher for project D than the average for other projects. For customer-observable defects, the cost is approximately double the average for other projects: for addition or change class defects, the cost is at least 60% more than the average for other projects (Figures 11.4 and 14.4).

We surmise the reason is memory constraint. We discuss the reason is 3.3 (4).

3.3. Discussion and Recommendations

In this section, we suggest process and product design improvements based on our analysis of the pattern observed. We recommend improvement activities in the context of the three main study areas: type of defect, relationship between defect and cost, and the effect of product architecture.

1) Type of defect

The main cause of customer-observable defects is algorithm, human interface, and external interface faults (from A1). The main cause of addition or change class defects is human interface and external interface faults (from C2). Therefore, preventing algorithm faults would improve reliability, and preventing human interface and external interface faults would improve both reliability and user-friendliness. The prevention of external interface faults would be especially effective in reducing severe defects (from A2).

These findings support the introduction of techniques such as prototyping of human-machine interface, code inspection, and the thorough review of external interfaces. In the current development process standard, engineers are not forced to apply those techniques in the appropriate development phases. This result shows adoption of those popular techniques would be useful for improving the development environment of this study. Without quantitative analysis, however, we might choose popular but not effective improvement actions in a given environment. This study illustrates the efficiency of analyzing type of defects when we try to plan improvement actions most suitable to individual environments.

2) Relationship between defect and cost

The injection phase is an important factor influencing the cost of rework (from B1 and D1). Defects injected during the requirement phase are extremely

expensive. Therefore, it is critical to prevent defect injection during this phase. It is expected that the introduction of such requirement analysis techniques are prototyping would be effective in preventing requirement-phase errors.

This study also shows that customer-observable defects caused by omission are more expensive than those caused by commission. At first glance, our results appear to contradict the results of Selby and Basili (1991) with regard to the relative cost of defects caused by omission errors versus those caused by commission errors. In their study, errors of omission were less expensive than errors of commission when found during the review process. However, in this study, if we considered all logic defects found during all phases, not just those that were customer observable, we have supportive results; i.e., we find the cost of rework for commission defects to be at least 10% higher than the cost for omission errors across all four projects.

This combination of results may have an important implication; defects of omission may be much chapter when caught earlier in the life cycle.

3) Effect of product architecture

At the beginning of the study, we surmised that product architecture influences the cost of rework. Although we could not find definitive quantitative evidence of this relationship, we suspect that patterns PA1 and PC1 stem from characteristics of the product architecture.

Project A has the lowest overall cost of rework of the four projects (Table 4). The cost of rework after the design phase is very low, while that cost during the requirement and specification phase is the highest of the four (from PA1) (Figure 14.4). These low rework costs after the design phase are the most remarkable feature of project A. We suggest that the product architecture used in project A influenced this factor. In project A, specification was described in the form of finite-state machines, and the specification was transformed into code by previously determined procedures. Tests were planned and performed based on the specification document. Therefore, most product changes were made sing semiautomated procedures.

Project C, on the other hand, has the highest cost of rework of the three, excluding project D (Table 4). PC1 suggests the reason for this high rework cost may be the fact that project C had several addition or change class defects caused by commission errors, which we determined are more costly than those caused by omission errors. The five addition or change class defects caused by commission errors in project C required fixes to 1, 2, 3, 4, and 4 modules, respectively. In project C, a modification to a requirement and specification spread across several modules, which perhaps indicates a weakness in the product architecture.

4) Effect of memory constraint

We found that memory constraint causes high cost of rework. In project D, which had a severe memory constraint, we saw two patterns that support this. We found that the addition of some functionality to the product was extremely expensive (from PD1). We found that the cost of rework on phases after implementation was much higher than it was during the phases prior to implementation (from PD2). We attribute these two features of project D to its severe memory constraint.

IV. CONCLUSION

We introduce a framework to measure and analyze software processes. The framework consists of descriptive models that abstract various aspects of process and product, evaluative models that formalize the analysis criteria, and a set of GQMs that clarify the relationship between the metrics, the analysis, and the goals.

In our evaluation, we discovered the importance of analysis from different viewpoints, because projects may rate completely differently depending on the criteria applied. In particular, we found that the number of defects (representing reliability) does not correlate to the cost of rework (representing maintainability).

In characterizing the projects, we found some patterns among types of defect, relationship between defect and cost, and effect of product architecture. They provide a quantitative basis for recommending certain process improvement activities with confidence in their effectiveness. Some results suggest that product architecture influences the software process, especially in the area of maintainability, although our analysis did not cover a large enough sample to prove that relationship quantitatively.

We may have gained some new insight about the cost of defects caused by omission versus commission depending upon the time when they are found in the life cycle. When we considered *all logic defects*, our results agreed with those of Selby and Basili (1991). On the other hand, when we considered only customer-observable defects (which was our initial measure), our results disagreed. This difference of approach/findings points out the importance of clarifying the definition and range of the

criteria used in any study or discussion of software issues.

In looking for influential factors, we studied the basic data using an intuitive pattern-searching technique. This method was fairly effective but does allow the possibility of overlooking some important patterns in the data. An automated pattern-recognition technique is needed, which would allow us to find every statistically meaningful pattern in the data.

ACKNOWLEDGMENTS

We thank Lionel Briand, Christopher Hetmanski for their careful review and suggestions on an earlier version of this paper and Jyrki Kontio for his review and suggestions on the current version.

REFERENCES

Basili, V. R., Software Modeling and Measurement: The Goal/Question/Metric Paradigm, University of Maryland Technical Report. UMIACS-TR-92-96, 1992.

Basili, V. R., Katz, E. E., Panlilio-Yap, N. M., Loggia Ramsey, C., and Chang, S., Characterization of an Ada Software Development, *IEEE Computer Magazine*, 53–65 (September 1985).

Basili, V. R., and Perricone, B., Software Errors and Complexity: An Empirical Investigation, *ACM Communications*, 27, No. 1, 45–52 (1984).

Basili, V. R., and Rombach, H. D., The TAME Project: Towards Improvement-Oriented Software Environments, *IEEE Transactions on Software Engineering*, SE-14, No. 6, 758–773 (1988).

Basili, V. R., and Selby, R. W., Jr., Data Collection and Analysis in Software Research and Management, in *Proceedings of the American Statistical Association and Biomeasure Society Joint Statistical Meetings* (1984).

Basili, V. R., and Weiss, D. M., Evaluation of a Software Requirements Document by Analysis of Change Data, in *Proceedings of the Fifth International Conference on Software Engineering*, 314–323 (1981).

Basili, V. R., and Weiss, D. M., A Methodology for Collecting Valid Software Engineering Data, *IEEE Transactions on Software Engineering*, SE-10, 728–738 (November 1984).

IEEE Guide for the Use of IEEE Standard Dictionary of Measures to Produce Reliable Software, IEEE/ANSI Standard 982.2-1988.

Selby, R. W., Jr., and Basili, V. R., Analyzing Error-Prone System Software, *nsactions on Software Engineering*, 141–152 (February 1991).

Weiss, D. M., and Basili, V. R., Evaluating Software Development by Analysis of Changes: Some Data from the Software Engineering Laboratory, *IEEE Transactions on Software Engineering*, SE-11, No. 2, 157–168 (1985).

Briand, L. C., Basili, V. R., and Thomas, W. M., A Pattern Recognition Approach for Software Engineering Data Analysis, *IEEE Transactions on Software Engineering*, SE-19, 931–942 (November 1992).

Chillarege, R., Bhandari, I. S., Chaar, J. K., Halliday, M. J., Moebus, D. S., Ray, B. K., and Wong, M-Y., Orthogonal Defect Classification—A Concept for In-process Measurement, *IEEE Transactions on Software Engineering*, SE-18, 943–956 (November 1992).

Ghezzi, C., Jazayeri, M., and Mandrioli, D., *Fundamentals of Software Engineering*, Prentice-Hall, Englewood Cliffs, NJ, 1991.

McGarry, F. E., Results of 15 years of measurement in the SEL, in *Proceedings of the Fifteenth Annual Software Engineering Workshop*, NASA/Goddard Space Flight Center, November, 1990.