Empirical study of software maintenance

Empirical study of software maintenance

M van Genuchten, G Brethouwer, T van den Boomen and F Heemstra

The paper describes an empirical study of software maintenance that was carried out in a system software department in 1989 and 1990. The study focused on error occurrence and fault detection. Over 400 problem reports were studied. The study showed some unexpected results. It showed, for example, no relation between the phase of error occurrence and the solution time. An explanation is the gap between the methods as they are supposed to be applied and reality. Assessment of the size of the gap is one of the contributions of this kind of empirical study.

maintenance, defects, metrics

Maintenance takes up a considerable amount of software engineer's time. Conte *et al.*[1] claim it takes up to 60 per cent of effort, and Lehman[2] states that 70 per cent of the expenditure on a program is incurred after initial installation. This paper describes an empirical study of software maintenance. The goal of the study was to gain insight into the origin of maintenance. This insight should enable software engineers and managers to take improvement measures that should reduce future maintenance efforts.

The study is an example of analysis of the software engineering process. Analysis of a software process should lead to improvement of that process. Data on the development process are required to be able to analyse it. The need for measurement and data collection of software development and maintenance is often stressed. Many organizations do not, however, practise these activities in software development. For instance, one survey indicated that 50 per cent of software development organizations in the Netherlands do not collect any data on the software process[3]. The study described in this paper can be perceived as an empirical study on the one hand. On the other, it is an example of the fact that data collection in software development can provide an organization with important insights and that data collection does not have to be complicated. It is hoped that this example inspires others to improve their software process by analysis that is based on facts and figures.

The paper consists of four sections. The first section discusses software maintenance — the topic under study. The second section addresses analysis of software development. The third section details the design of the study, with the results and interpretation appearing in the fourth section. Finally, the paper is rounded off with some conclusions.

Department Information & Technology, Pav. D-3, Faculty of Industrial Engineering, Eindhoven University of Technology, Post Box 513, 5600 MB Eindhoven, The Netherlands, and Lighthouse Management Consultants, Post Box 6427, 5600 HK Eindhoven, The Netherlands

SOFTWARE MAINTENANCE

Emphasis in software engineering research is on the development of new software products. In theory, the software engineer is finished by the time the product fulfils the requirements and the software is installed. The introduction to this paper has already revealed that the majority of the cost is spent after initial installation. Expenditure after initial installation is usually referred to as maintenance. The authors are aware of the fact that maintenance is an inappropriate term for software because maintenance is in fact prolonged development and that the terms software development and maintenance could be better replaced by the term software evolution[1]. They are also convinced that control of software development and control of software maintenance should be integrated. Control of software engineering cannot afford to limit its attention to, say, 30 per cent of the expenditure and consider the remaining 70 per cent as somebody else's problem[4]. This paper uses 'maintenance' in the sense that Swanson introduced when he distinguished three kinds of maintenance[5].

- Corrective maintenance is maintenance performed in response to processing, performance, or implementation failures.
- Adaptive maintenance is maintenance in response to changes in data and processing environments.
- Perfective maintenance is maintenance performed to eliminate processing inefficiencies, enhance performance, or improve maintainability.

Maintenance is often associated with software faults and failures. The IEEE glossary of terms[6] distinguishes between failures, faults, and errors. An error is defined as a defect in the human thought process. Faults are concrete manifestations of errors within the software. Failures are departures of the software system from software requirements. The terms 'error occurrence' and 'fault detection' will be used many times in this paper.

CONSTRUCTION VERSUS ANALYSIS

Basili and Rombach distinguish between analytical and constructive aspects in software development[7]. The dis-

 0950-5849/92/080507-06

tinction of construction of a software product and analysis of the software process is useful to direct improvement efforts. Analytic and constructive activities are distinguished, as well as analytic and constructive methods and tools. Whereas constructive methods and tools are concerned with building products, analytic methods and tools are concerned with analysing the constructive process and the resulting products. 'We need to clearly distinguish between the role of constructive and analytical activities. Only improved construction processes will result in higher quality software. Quality cannot be tested or inspected into software. Analytic processes (e.g. quality assurance) cannot serve as a substitute for constructive processes but will provide control of the constructive processes.'[7] (p 759).

The distinction as made by Basili and Rombach provides insight into the control of software development. Data on the development process are required to be able to analyse software development. This is the appropriate place to quote Lord Kelvin from a century ago: 'when you measure what you are speaking about, and express it in numbers, you know something about it: but when you cannot measure it, when you cannot express it in numbers, your knowledge is of a meagre and unsatisfactory kind; it may be the beginning of knowledge, but you have scarcely in your thoughts advanced to the stage of science.'[8].

Analysis and measurement should focus on one or a few aspects of the software process because they are time-consuming activities. For example, it is possible to focus on reasons for delay in software development, on the impact of new tools, or on the impact of change requests on the development process. Anyway, the goal of the analysis should be clear. The goal should be quantified to allow for analysis and measurement. The Goal/Question/Metric paradigm[7] can be an aid in goal setting and operationalization of analysis of software development. The paradigm represents an approach by which analysis goals are tailored to the needs of an organization. The goals are refined into a set of quantifiable questions that, in its turn, specifies a set of metrics and data for collection. As such, analysis of software development is incorporated into the development process. The study described in this paper can be perceived as an example of the application of the Goal/Question/Metric paradigm.

DESIGN OF STUDY

The maintenance study took place in a department that was concerned with development and integration of system software in the operating-system and data-communications fields. The department employed 175 software engineers and covered a range of 300 products. The quality assurance department consisted of 10 people and the methods and tools department employed five people. The maintenance study was a consequence of an earlier study on reasons for delay in software development[9]. Reasons for delay were studied because insight revealed from these can lead to improvement measures that should enable future projects to follow the plan more closely.

Analysis of 160 activities, comprising over 15 000 hours of work, resulted in a number of improvement measures, of which two will be named. The first measure was the introduction of maintenance weeks. Analysis of the collected data showed that maintenance activities in particular were a constant interruption to development and caused a considerable part of the delay. A number of possible ways of separating development and maintenance was discussed. It was decided to concentrate the maintenance work as far as possible in maintenance weeks and to include two maintenance weeks in each quarter. By carrying out most of the maintenance during these two weeks, development could proceed more quickly and with fewer interruptions during the other weeks.

A second measure was the start of the study that is discussed in this paper. The goal of the study was to increase the insight into the origin of maintenance. The reasons for the delay study focused on the control aspects of time and cost, while the maintenance study focused on the aspect of quality. This study focused on maintenance reports. Pettijohn[10] pointed out that maintenance reports are one of the two primary sources of quality data; the other source is inspection data. Maintenance reports were named 'problem reports' by the department concerned. A problem report can be written by engineers within the development department, employees outside the development department, or customers who perceive a problem. A problem can be a software failure. Problem reports can, however, also be abused. An additional requirement may be formulated, incorrectly, by a user as a problem in a problem report. The term problem will in this paper refer to a shortcoming of a software product, as perceived by the writer of a problem report. Whether a problem is related to a software fault is still to be determined.

The study addressed four questions that will be discussed subsequently. They will be referred to as 'analysis questions' and represent the questions in the Goal/Question/Metric paradigm.

(1) How much effort does it take to solve a problem?
The study of reasons for delay made it clear that maintenance troubled development. It was not yet clear how much effort it took to correct faults. Knowledge of the distribution of the correction time can, for example, be used to plan maintenance activities.

(2) How is error occurrence distributed over the development phases?
Knowledge of the distribution of error occurrence over the phases of development pinpoints software construction problems and, as such, identifies areas for improvement. The long-term solution is improvement of the construction process. A short-term improvement measure may be to focus inspection and test emphasis on the phases that are most error prone.

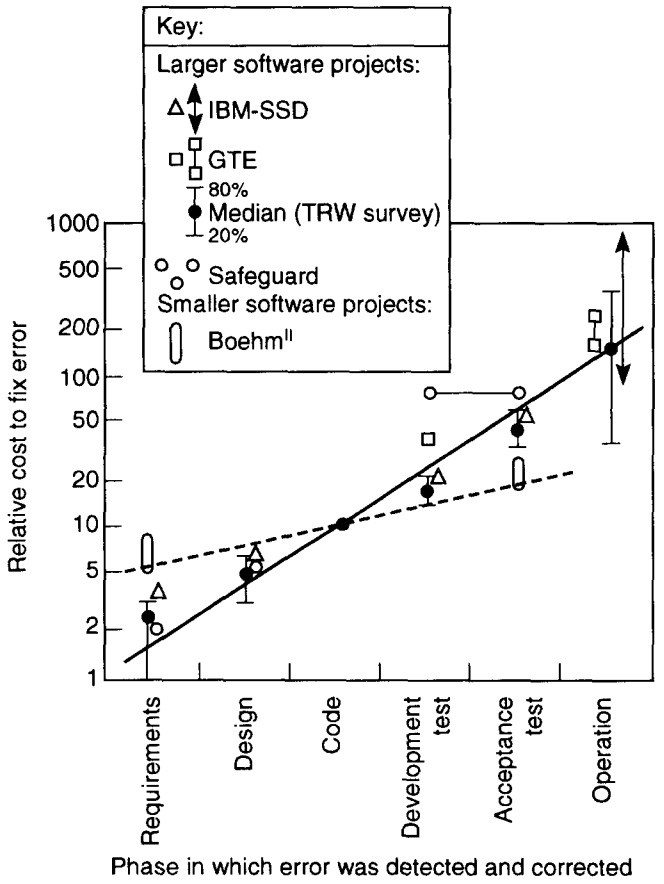

Figure 1. Increase in cost to fix or change software through development life-cycle[11]

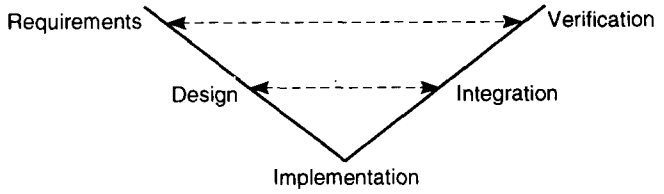

Figure 2. V model of development and testing[13]

(3) Do late corrections require more effort?
It would be expected that the cost to fix faults increases towards the end of the project because when an error is detected in a test not only must the fault be corrected, but also the upstream documents must be updated as well. Figure 1 gives the relative cost to fix failures in subsequent phases of development, as they were found by Boehm[11].

The authors wanted to check whether the figure reflected the experience in the department concerned. Additional effort can be spent to avoid errors in the earlier phases of development if the figure reflected experience. If not, the department should ask itself why it does not behave as expected.

(4) Does the V model of development and testing work?
The department had adopted the V model of development and testing introduced by Myers[12] (see Figure 2).

Table 1. Three multiple-choice questions added to problem report

(1) How many hours did it take to solve the problem?
- Less than one hour
- 1 to 2 hours
- 2 to 4 hours
- 4 to 8 hours
- Over 8 hours

(2) In what phase did the error occur?
- Requirements
- Design
- Implementation
- Other, ...

(3) In what test was the fault detected?
- Integration test
- Verification test
- Validation test

The V model shows the phases as distinguished by the department concerned. The phases are requirements, architectural design, implementation, integration, and verification and validation.

The V model shows that a development team tests a concept exploration document in the validation test, a requirements specification in the verification test, and a design document in the integration test. It would be expected that the fault-detection phase is related to the error-occurrence phase.

The metric is derived from the four analysis questions above. The metric consisted of the answers to three multiple-choice questions that were added to all the problem reports. The metric represents the third step in the Goal/Question/Metric paradigm. The multiple-choice questions are given in Table 1.

The multiple-choice questions are derived from questions proposed by Basili and Rombach[14]. The three questions were stated for every problem that was solved during four months. Eleven project leaders and some of their team members were involved in data collection. The data collection did not take much time for the project leaders involved. The authors' experience is that this is a prerequisite for successful data collection in software development. Another prerequisite is feedback of the results to the participating project leaders. This will be returned to when the results of the study are discussed in the next section.

RESULTS

Over 400 problem reports were analysed. The most important results of the study are presented in Table 2 and Figures 3–5. Table 2 comprises the answer to the first two analysis questions stated in the previous section. It shows the correction time versus the phase of error occurrence. For example, 22 errors that were incurred in the requirements phase took less than an hour to solve.

A large number of errors is classified in Table 2 as 'other'. It became clear during the study that some problems that were reported were not related to software errors or could not be attributed to a particular phase.

Table 2. Number of problem reports, distributed over correction time and error occurrence

Phase of error Occurrence	Time to correct fault (hours)						
	<1	1–2	2–4	4–8	>8	Total	(%)
Requirements	22	10	7	0	1	40	10
Design	12	8	6	1	4	31	7
Implementation	93	37	15	6	12	163	40
Other	103	25	16	7	26	177	43
Total	230	80	44	14	43	411	100
%	56	20	11	3	10	100	

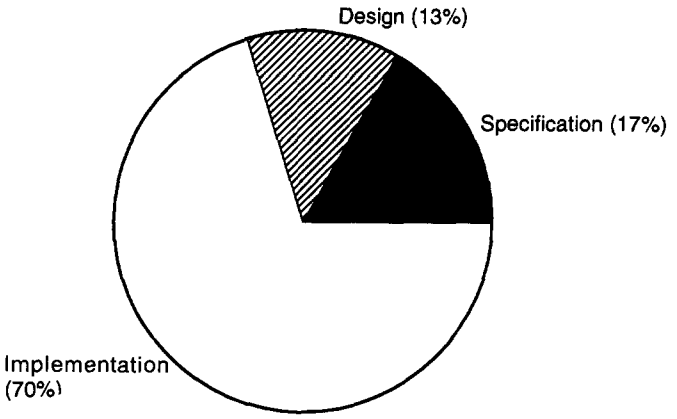

Figure 3. Distribution of error occurrence

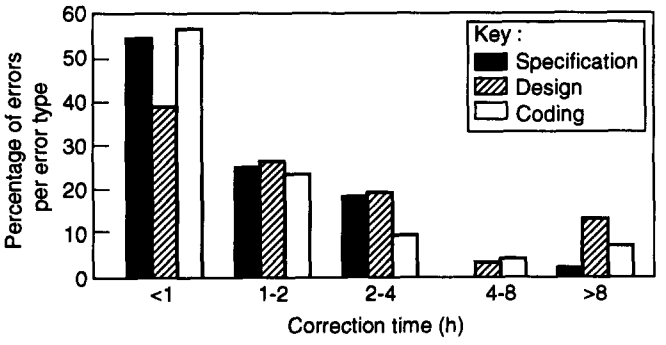

Figure 4. Distribution of error occurrence over classes of correction time

An example of the first case is change requests that are reported as problems.

Now focus is switched to the analysis questions that were stated in the third section. The first analysis question was 'How much effort does it take to solve a problem?' Table 2 shows that over 50 per cent of the problems are solved within just one hour. Apparently, most of the problems are not too difficult to solve. A conclusion is that the maintenance problem is more a lead-time than an effort problem: it takes more time to get the problem to the right developer than to solve the problem.

The second analysis question was 'How is the occurrence of errors distributed over the project?' The distribution of errors over the requirements, design, and implementation phases is given in Table 2 and is presented in the pie chart in Figure 3. The errors that were classified as other are left out of the diagram.

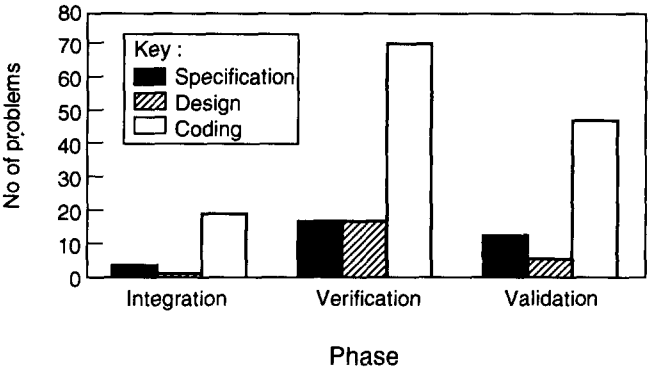

Figure 5. Error-occurrence phase versus fault-detection phase

Figure 3 shows that the majority of errors was reported as implementation errors. A comparison with other studies shows that error distributions depend on the development processes and environments. Two studies by Basili *et al.*[14,15] show different results. The first study concerned a general-purpose program for satellite planning studies. The error distributions shows that 48 per cent of the errors was attributed to incorrect or misinterpreted functional specifications or requirements[15]. The second study concerned a ground support system. A large amount of software was reused. The error distribution showed that 78 per cent of the errors was related to design or implementation errors of a single component[14]. The two studies by Basili *et al.* and the study described in this paper show that the error distributions are determined by the software engineers, the kind of application, and the development process. It is recommended that every software development organization gains insight into the distribution of errors to be able to take improvement measures that allow the number of errors to be reduced in the future.

The third analysis question stated was 'Do late corrections require more effort?' Figure 4 shows the distribution percentages of requirements, design, and implementation errors, distributed over the classes of correction time. For example, 55 per cent of requirements errors, 39 per cent of design errors, and 57 per cent of implementation errors are solved within the hour.

It was expected to find that, for example, requirements errors take more time to solve than design and implementation errors. At first sight there is no relation between error occurrence and correction time.

The analysis question was stated in terms of the hypothesis 'There is no relationship between the error occurrence phase and the solution time'. Loglinear analysis[16] was used to test this hypothesis. Loglinear analysis can be used to analyse cross-tables. The cross-table concerned (Table 2) gives the distribution of problems over the error-detection phase on the one hand

and correction time of the fault on the other. The structure of the table can be described by the independence and the association model as well. The hypothesis is therefore not rejected. This indicates that there is no relation between error occurrence and solution time, contrary to what the authors expected and what Boehm[11] has found. The reasons behind this will be discussed after the last analysis question, related to the one just discussed, has been dealt with.

The last analysis question was 'Does the V model of development and testing work?' Figure 5 gives the number of requirements, design, and implementation faults found in the various tests, with the integration, verification, and validation tests on the horizontal axis. The number of faults that are detected is distributed over the phases in which the errors occurred. For example, three requirements errors, one design error, and 19 implementation errors are found in the integration test.

At first glance, there is no clear relation between the kind of error and the phase in which a fault was detected. This was confirmed when the hypothesis 'There is no relationship between the kind of error and the phase in which a fault is detected' was tested. Loglinear analysis showed that the hypothesis did not need to be rejected. It is concluded that the empirical data did not reflect the use of the V model.

The answer to the last two analysis questions did not confirm expectations. The results were discussed with the relevant project leaders and their manager. The authors consider it to be important to discuss the results with the participants. This is a general starting point that Bemelmans has called closed-loop information supply[17]. The idea is that the people who have to provide input to an information system should benefit from the output of that information system. The fact that they benefit from correct output and are harmed by incorrect output is the incentive to provide accurate input. The principle could be summarized by 'nothing for nothing'. The closed-loop principle can be applied to information supply in software development: the data that are collected by software engineers should support the engineers in doing their job.

From discussion of the results, the authors concluded that there was a serious misfit between the methods as they were supposed to be applied and reality. There will always be some kind of gap between the theoretical concepts and the application in reality; it is important, however, to assess the size of the gap. For example, one explanation for the fact that there was no relation between the kind of error and the solution time could be the fact that any fault is resolved in the code. It is to be expected that a requirements error that is detected in a test will take more time to solve because upstream documents, such as design and requirements documents, will have to be updated. If the upstream documents are not updated there is a serious gap between theory and practice, returning to code and fix practices. The facts illuminated the misfit between theoretical concepts and reality. This has led to a renewed discussion of the role of theoretical concepts and their application in the department concerned. A major difference is that the discussions can now be based on facts provided by this empirical study.

Another result of the study is the questions that it raised. For example, the fact that this study showed that handling maintenance is mainly a lead-time problem raised the question 'How can maintenance be organized so that lead time is shortened?' A second question that was raised was: 'Are change requests handled properly?' This question was raised because this study has shown that many users formulated their change requests as problem reports. The answering of these and similar questions will require additional data collection and analysis that should result in further improvement of the software development process.

CONCLUSIONS AND RECOMMENDATIONS

This paper has discussed an empirical study of software maintenance. Maintenance takes a considerable part of the total expenditure on software. The study is an example of analysis of the software process. The distinction of construction of a software product and analysis of the software process is useful to direct improvement efforts. Basili and Rombach[7] state that there needs to be a clear distinction between the role of constructive and analytical activities. Analysis requires measurement and data collection in software development. The lack of data collection in current software processes indicates a lack of interest in analysis of the software process.

This study is an example of the fact that measurement and analysis can be done in a way that is simple and provides useful insights. It is not possible to measure everything at the same time, and therefore attention should be limited to one or a few aspects of the software construction process. The Goal/Question/Metric paradigm[7] may be helpful in this focus. Examples of analysis studies that are similar to this study have been reported[14,18].

This study focused on maintenance, particularly error occurrence and fault detection. It was not possible to show a relation between phase of error occurrence and phase of fault detection. It was also not possible to show a relation between the phase of error occurrence and solution time. This indicated a gap between the methods as they were supposed to be applied and reality. The study also showed, once more, that data should be collected and used by every software department. It makes little sense to try to gain insight from somebody else's data because software development differs considerably from place to place. An important requirement for data collection and analysis in software development is the cooperation of the engineers involved.

Another typical result of this study was that it raised more questions than it answered. The authors intend to continue to raise and answer questions that increase insight into the software engineering process.

ACKNOWLEDGEMENTS

The authors thank the participating project leaders for their cooperation in the study. They also thank Ben Smit and Rob Stobberingh for their support and for entertaining discussions during the study. Finally, thanks to Stephen Speirs for his comments.

REFERENCES

1 **Conte, S D, Dunsmore, H E and Shen, V Y** *Software engineering metrics and models* Benjamin/Cummings (1986)
2 **Lehman, M M** 'Program evolution' *Inf. Process. Manag.* Vol 20 Nos 1–2 (1984)
3 **Siskens, W J A M, Heemstra, F J and van der Stelt, H** 'Cost control in automation projects, an empirical study' *Informatie* Vol 31 (January 1989) (in Dutch)
4 **Genuchten, M J I M van** *Towards a software factory* Kluwer Academic, Dordrecht, The Netherlands (1991)
5 **Swanson, E B** 'The dimensions of maintenance' in *Proc. 2nd Int. Conf. Software Engineering* (October 1976)
6 **IEEE** 'IEEE standard glossary of software engineering terminology' *Rep. IEEE–std–729–1983* IEEE (1983)
7 **Basili, V R and Rombach, H D** 'The TAME project; towards improvement oriented software environments' *IEEE Trans. Soft. Eng.* Vol 14 No 6 (1988) pp 758–773
8 **Gilb, T** *Principles of software engineering management* Addison-Wesley (1988)
9 **Genuchten, M J I M van** 'Why is software late? An empirical study of reasons for delay in software development' *IEEE Trans. Soft. Eng.* Vol 17 No 6 (June 1991)
10 **Pettijohn, C L** 'Achieving quality in the development process' *AT&T Tech. J.* Vol 65 No 2 (March/April 1986)
11 **Boehm, B W** *Software engineering economics* Prentice Hall (1981)
12 **Myers, G J** *Software reliability: principles and practices* John Wiley (1976)
13 **Boomen, T and Brethouwer, G** 'On the analysis of software development' *Master's thesis* University of Technology, Eindhoven, The Netherlands (April 1990)
14 **Basili, V R and Rombach, H D** 'Tailoring the software process to project goals and environments' in *Proc. 9th Int. Conf. Software Engineering* (1987)
15 **Basili, V R and Perricone, B T** 'Software errors and complexity: an empirical investigation' *Commun. ACM* Vol 27 No 1 (January 1984)
16 **Fox, J** *Linear statistical models and related methods: with applications to social research* John Wiley (1984)
17 **Bemelmans, T M A** 'Management information systems; questions, no answers' *Informatie* Vol 31 (June 1989) (in Dutch)
18 **Weiss, D M and Basili, V R** 'Evaluating software development by analysis of changes: some data of the SEL' *IEEE Trans. Soft. Eng.* Vol 11 No 2 (February 1985)