Los Alamos National Laboratory is operated by the University of California for the United States Department of Energy under contract W-7405-ENG-36

LA-UR--83-3085

DE84 001722

TITLE PROLONGING THE LIFE OF SOFTWARE

AUTHOR(S) CONNELL, JOHN
BRICE, LINDA

SUBMITTED TO 1984 NATIONAL COMPUTER CONFERENCE
LAS VEGAS, NEV., JULY 1984

DISCLAIMER

This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor any agency thereof, nor any of their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency thereof. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof.

MASTER

By acceptance of this article, the publisher recognizes that the U.S. Government retains a nonexclusive, royalty-free license to publish or reproduce the published form of this contribution or to allow others to do so for U.S. Government purposes.

The Los Alamos National Laboratory requests that the publisher identify this article as work performed under the auspices of the U.S. Department of Energy.

Los Alamos Los Alamos National Laboratory
Los Alamos, New Mexico 87545

by JOHN CONNELL

and LINDA BRICE

Los Alamos National Laboratory

Los Alamos, New Mexico

PROLONGING THE LIFE OF SOFTWARE

ABSTRACT

Presented here are methods for successfully controlling software maintenance activity so that present systems will be more useful and less expensive to support. While it is based on experience at Los Alamos National Laboratory, it is not based on solutions developed and implemented there. Los Alamos is presently struggling with the problems identified in this paper and is impacted by them to the same extent as the rest of industry. An idea has emerged from this struggle: the deterioration of production software is basically a quality control problem which can and should be solved. Many data processing shops currently have two options concerning old (over five years), marginally useful systems; pay the high cost of supporting them or undertake a rewrite. If the principles presented in this paper are applied, a third option may become available; prolonging the useful life of software by making it more cost-effective to support.

KEYWORDS: Software Maintenance; Software Quality Assurance; Administrative Data Processing; Software Maintenance Staffing; Software Metrics and Standards; Software Inspections or Walkthrus.

INTRODUCTION

The Administrative Data Processing Division of the Los Alamos National Laboratory supports over seventy production software systems for various users within the Laboratory. Each system represents a particular financial, personnel, or inventory application consisting of a related set of software modules. Altogether there are about 900 computer programs in production, most of them written in COBOL. The systems reside on both minicomputers and mainframes. There are 50 programmer/analysts involved in developing, implementing and maintaining the systems.

When new systems are developed or old ones rewritten, a cost/benefit analysis is required prior to design. Assumptions must be made about the expected economic life of a proposed new system in order to estimate future operating costs and determine the payback period. The current standard is to design for a minimum five year economic life. Extrapolating to a ten-year goal for keeping software alive, return on investment is doubled and slack is built in for unanticipated changing requirements which can necessitate premature rewrites.

To prolong the life of software, it is necessary to maximize the continuing maintainability, operability, and usability

of current systems. This paper contains suggestions, based on experience at Los Alamos, for that maximization. No new software engineering concepts are introduced. Instead an extent body of knowledge is drawn upon and related to managing the maintenance of current systems in a cost-effective manner.

THE EFFECT OF UNSTRUCTURED MAINTENANCE

Entropy of structure:

Programmer productivity aids such as structured techniques, introduced in the last few years, are now in use in many DP installations and are expected to ease the future maintenance burden. However, many installations have been slow in adopting such techniques and those that have still experience the entropy problem. One of the worst effects that maintenance work can have on production software systems is the deterioration of the original structure of the system. Dozens to hundreds of small, seemingly insignificant patches applied during the life of a system can cause degeneration of even the most structured, modular, top-down original code. The author of an old program is rarely able to tell the current maintenance programmer what the code is doing, primarily because the author cannot himself remember. Many times the author cannot be located,

or even identified. A little-considered factor in the general maintenance dilemma is that the program is really co-authored by all of the programmers who have ever worked on it. Given that many different styles and design philosophies have been incorporated, there is little chance that the current code bears much resemblance to the original.

Introduction of defects:

Many good maintenance programmers might take offense at the suggestion that defects are inserted into systems as they perform their valuable work. They might argue that they always conduct thorough tests before putting changes into production. It should be pointed out that the term thorough, when applied to testing of maintenance changes, is probably a contradiction in terms. As an example, suppose three lines of code in one program of a sixty-program system are changed. Should the entire system be tested as thoroughly as it was during the development phase of the life cycle? If not, is there some chance that although the modified program will work fine, other unsuspected parts of the system will be negatively impacted? Might the system work perfectly for the first several production runs after the changes are put in place, only to have the inserted defect surface and cause trouble months later? Problems

involving the worth of regression testing and phenomena such as the ripple effect are well-documented.[1]

The above questions are difficult if not impossible to answer. In many cases, the maintenance programmer has no time to do complete, thorough testing for the same reason that there is no time for elegant coding; the fix is made in a crisis mode. It is not even clear that rigorous, extensive testing is cost-effective for minor changes. On the other hand, it should be assumed that the lack of such testing will guarantee the insertion of defects into the system in at least some cases. It is not pessimism but the logical conclusion that, over time, a system will become increasingly bug-ridden.

Introduction of psychological complexity:

Psychological complexity can be defined as elements of programming style which make programs difficult to understand. Complexity increases the effort required to make successful maintenance changes and thus increases maintenance costs. An example well documented and measured mathematically is use of the GO TO statement. Use of GO TO's is a violation of structured programming concepts and has been discouraged for some time. At the same time, a GO TO is the easiest, quickest way to modify the control flow

of a program and is frequently done, on the fly, to correct a logic flaw. In some cases to do otherwise would involve extensive rewriting of major portions of the program.

Again, a good programmer would be offended at the suggestion that some maintenance changes introduce psychological complexity into the system. Nevertheless, it must be true that at some point in our career all of us have been guilty of making a quick fix with a GO TO, neglecting to thoroughly document a midnight maintenance change, or adding another level of nesting to an already complicated IF statement to incorporate a new requirement. Several years of this type of activity will make the simplest program almost impossible to follow.

Increase in future maintenance costs:

As an old application system begins to deteriorate due to entropy of structure and the insertion of defects and psychological complexity, the cost of maintaining that system will begin to rise. At first the increase will be very slight, but as the factors mentioned above are compounded the rate of increase for maintenance cost will become geometric. The level of pain experienced in maintaining a production system can and should be measured; at some point it will become cheaper to scrap the old system

and build a new one from scratch. This theory is suggested graphically in Figure 1.

In most data processing organizations, it is politically advantageous and more satisfying to the users to devote development resources to desired new applications than to the rewrite of existing systems, even when it can be demonstrated that there would be a cost/benefit payoff derived from a rewrite. After all, it is somewhat embarrassing to confess to the user that his system has been damaged such that it is no longer maintainable and will have to be frozen for a period of time while it is being rewritten. Therefore, considerable benefits could be derived from putting into place goals, objectives and procedures that would help to delay the necessity for a rewrite by minimizing the rate of deterioration of applications systems. Zvegintzov has stated the desirability of this succinctly in a recent article in Datamation where he says, "Replacement of functions incurs a development cost that most DP organizations will not bear. 'Add on, not replace' is the trend in software." [2]

A BRIEF METHODOLOGY FOR MAINTENANCE WORK

Impact of changes on previous analysis:

Analysis documents, if they are accurate, can serve as valuable maintenance aids. If the maintenance programmer understands what the system is supposed to do and what the significance of the implemented functions is to the user, then s/he will have a good basis for knowing how to respond to emergencies that might arise. A document such as an essential requirements definition will also help the maintenance programmer know when the system is or is not successfully performing its required functions.

There are three direct implications of the above assertion. First, it implies that analysis documents such as a System Requirements Definition should become part of the retained system documentation for implemented systems. Second, the portions of this documentation which map the current system, such as Data Flow Diagrams should be accurately revised when the user's changing requirements result in maintenance changes that modify the functions of the system. Third, to understand and be able to modify an analysis document correctly, a maintenance programmer must also be somewhat of an analyst.

External documentation such as control flows and run procedures also helps to identify the impact of changes to one program on other parts of the system. It is helpful in testing systems and in returning them to production upon

successful test. Like requirements specifications, if the external documentation is to be useful and reliable, it must be revised accurately when maintenance changes affect its correctness.

Structured maintenance walkthrus:

Actual coding changes made to production source code files can be a frightening activity. Statistics indicate that a line of maintenance code costs ten to 100 times what a line of development code costs.[1] For the reasons given above, each new line of maintenance code contributes to destroying the viability of a system that cost thousands, maybe millions of dollars to develop. A worthy goal is to minimize mistakes made during this activity.

Walkthrus are becoming more common in the data processing profession. Managers have been accepting the fact that walkthrus save time and money by discovering errors more efficiently than any known testing method. Unfortunately, current opinion seems to be that this is a process applicable only to the development phase of the system life cycle. It is true that walkthrus are critically important during the early phases of development because errors are much less expensive to correct at that time than they are later. This does not constitute proof that walkthrus would

not be effective during the maintenance phase. If walkthrus are to be successful, they should contain the following elements: checklists, criteria, objectives, trained coordinators, established roles, feedback and feed-forward. The reader is referred to other works[3,4,5] for more information on the walkthru concept.

Applying maintenance walkthrus:

The following explains how maintenance is organized at the Los Alamos National Laboratory. Each system is identified with a unique two-digit number, e.g. General Ledger = 70. The table shown in Figure 2 shows how maintenance responsibility is allocated between these systems where primary responsibility is in the center column, secondary responsibility is in the column labeled "backup1", and the person responsible for the system in the event that the first two are unavailable is shown in the rightmost column labeled "backup2." In actual operation, systems are not maintained in a fashion as clean as the table suggests. In many cases the secondary backup knows nothing about the system and simply hopes it will never break at a time when neither the primary nor the backup are present. The backup often only has a passing acquaintance with the system, gained when the primary was sick or on vacation and a problem occurred. Even the primary's knowledge may be limited

because staff shortages and large service request backlogs mean assigning too much maintenance responsibility to too few programmers.

A suggested format for walkthrus of maintenance changes under the above circumstances is: the programmer making the change assumes the role of presenter/implementor; the other two programmers are responsible for the review and critique; and a fourth person with appropriate training becomes the coordinator/moderator/scribe. Such walkthrus do not always have to be as comprehensive as a walkthru for a major development project. A fifteen-minute walkthru for a change that took eight hours to make would seem sufficient. Such a process would simultaneously accomplish three objectives: insertions of defects and psychological complexity and deterioration of structure would be minimized; maintenance of external documentation would be maintained; and the members of the walkthru team would be educated through the preparation and attendance necessary for the walkthru. If such walkthrus were always required, systems would hopefully live longer, break less often, and be easier to maintain. In addition, the terms backup, and secondary backup, would come to have a more reliable meaning.

Summarizing the top-down approach to maintenance:

Good maintenance work requires a maintenance analyst who is just as professional in terms of software engineering know-how as a good senior programmer/analyst in the development area. The same basic activities are involved: analyze the problem, develop a solution, test the solution, implement the solution. The ideas presented in the preceding sections suggest a miniature life cycle approach to making maintenance changes as follows:

1. do a thorough analysis of the change request to determine needed modifications to system functionality;
2. study the old functional analysis to determine the impact of the proposed changes on the total system;
3. revise functional analysis as appropriate;
4. revise design and internal specification documents as appropriate;
5. make the changes according to the new analysis and design;
6. test the changes using both dynamic (standard test beds) and static (walkthrus) procedures;

7. implement the changes when they pass all tests.

CONTROLLING THE QUALITY OF MAINTENANCE WORK

Given the above means for doing quality maintenance work, what controls should be put in place to assure that quality will improve? It is recommended that controls consist of workable mechanisms for measurement, evaluation, and feedback. A non-workable mechanism is micro-management, whereby the line manager watches the maintenance programmer carefully and constantly to ensure that mistakes are avoided. If, instead, meaningful measurement of the quality of maintenance work is taking place, it can provide the basis for effective performance evaluations and feedback that should produce the desired results. Ways of controlling quality include: user surveys; and measurement of maintenance costs via measurement of reliability and measurement of defects.

User surveys:

Since most data processing professionals belong to organizations whose budget or income is derived by providing a service perceived to be useful by users outside of their organization, user satisfaction surveys should be one of the most important means of measuring quality. Figure 3 shows

a portion of such a survey that was taken of users' of administrative application software systems at Los Alamos. Users were asked to give their degree of satisfaction for different classes of services on a numeric scale, providing a means for measuring the degree of user satisfaction quantitatively.

Because users have different personality profiles, some are easier to please than others. Ideally, user personnel would be held constant while the surveys were taken in a time-series fashion, allowing for measurement of change in degree of satisfaction over time. The problem of personnel turn-over in the user organization can be circumvented if a profile for the entire organization can be developed. Data relating to quality of maintenance work should come from feedback on the useability, operability, and usefulness of the user's system.

The importance of measuring maintenance costs:

Accurate measurement of maintenance effort in programmer hours is critically important for several reasons. Our goal is to control the quality and the expense of software maintenance, and it has been pointed out that "You can't control what you can't measure." [6] Maintenance effort measurement can be used for cost/benefit analysis of

proposed rewrites.[7] If maintenance effort is decreasing dramatically on a particular system, the decrease may be an indication that high quality maintenance work is being performed. Useful measurement should differentiate between bug-fixing and making changes necessitated by changing user requirements. This would provide a means for knowing when the quality of a production system was deteriorating if bug-fixing effort begins to rise significantly.

Measuring software reliability:

Aborts, reruns, and user trouble calls are costly. They can also be reduced by the performance of high quality maintenance work, although recognition of quality can sometimes be difficult. It is possible to force a program to execute successfully under almost any circumstance, but if the output is not correct this will usually be caught either by production control or the user, resulting in a rerun or a trouble call. Careful records in the form of production logs must be kept and published in order to determine who is truly doing work toward the reduction of the problems.

Measuring insertion of defects:

We should be very concerned about the rate of insertion of defects into a production system. The walkthru procedures

discussed above should help reduce the insertion of defects, but it provides no guarantee that zero defects will be inserted. Furthermore, it provides no measurement of the insertion of defects since the walkthrough team must stipulate that they are unable to find any defects before a change will be put into production. The only reliable measure of defects is a count of fix-a-bug requests from users. The important measure here is the actual number of such requests, not the amount of effort spent on them. Increases in the receiving rate of these requests should be an indictment of the walkthrough team as well as the responsible programmer. Decreases in the receiving rate would indicate that high quality maintenance work is being performed.

Evaluating maintenance performance:

In order to effectively implement the controls suggested above, criteria for acceptable performance of maintenance work should be published and distributed among the maintenance programmers. In order to do high quality maintenance work, the staff needs to know what the goals are, how goal achievement will be measured and what constitutes a satisfactory level of performance. Figure 4 shows suggested performance evaluation guidelines for maintenance programmers.

CAREER PATHS FOR MAINTENANCE PROGRAMMERS

Who should do maintenance work? How long should they do it? What should appropriate rewards be for successful maintenance programmers? What should the organizational goal be for maintenance activity as a whole? These topics could serve as the basis for further research, but they deserve at least brief attention within the scope of this paper.

Maintenance as a training experience:

In many organizations, maintenance work serves as an initiation period for programmer trainees. This is not an entirely bad idea. Recent graduates have been schooled in the latest structured programming techniques and may have an inclination to keep the code they are responsible for as clean as possible. Also, it provides a series of little problems for the trainee to solve before being faced with a big problem. It becomes a bad idea when an organization only has green recruits supporting its production systems. This situation usually signifies an organizational attitude that maintenance work is not as important or technically demanding as development work. It has been pointed out recently[2] that this attitude is not appropriate since it is software maintenance that keeps the business running smoothly by supporting critical applications.

Maintenance trouble shooters:

In most medium to large size organizations, it is possible to find several maintenance experts. These software "doctors" are proficient at quickly identifying and solving very complex problems. They usually derive a great deal of enjoyment from it. This is not hard to understand since people usually enjoy doing things at which they excel. These people should be provided with career paths and monetary rewards which encourage them to keep doing what they enjoy and do well. They should not be "promoted" to development projects, which among other negative results, starve them of the pleasure of immediate feedback present in problem-fixing.

These seasoned professionals can provide excellent supervision and guidance for the trainees mentioned above. However, if assigned to a development project, they may not view the assignment as a reward and might actually have a subconscious tendency to insert problems into the product so that they will have an opportunity to do something enjoyable at a later time.

Maintenance groups as a separate entity:

If it makes sense to have different types of employees doing the maintenance work, then it follows that it is also sensible to have a separate maintenance group in the data processing organization chart. This group would have a different set of talents and/or interests than those doing development work and would be evaluated on a somewhat different basis. Trainees could work on teams with more experienced maintenance analysts supporting production systems. Very successful maintenance analysts could be promoted to team leaders. Those who are very successful and have valuable management talent (proven as team leaders) become likely candidates for line manager of the maintenance group.

Reductions in force:

If a maintenance group is successful, would its staffing requirements diminish? Perhaps, temporarily. One of the criteria for successful performance of maintenance work is that maintenance effort and the occurrence of problems be reduced. The implication is that less manpower will be required to support the current applications. Nothing is said about existing backlogs of requests for new development. If maintenance effort per system were to be

reduced, users would no doubt be delighted to see development proceed at a brisker pace. Development in turn creates more systems requiring maintenance, returning the maintenance staff requirements to original levels or higher. Thus, there is no need for maintenance programmers to fear that good work will eliminate their job. The extent of automation is still increasing in all fields, albeit at a slower pace than most users would like to see.

Performance rewards and appropriate goal setting:

A suggested goal for the organization is to minimize required maintenance effort and the occurrence of problems with production software on a per-system basis over all production systems. Hopefully, some of the ideas detailed above will prove useful in accomplishing this goal. If all these ideas are implemented, how should a successful maintenance analyst be rewarded? This person has improved the degree of user satisfaction with data processing service, reduced the amount of effort required to maintain systems, extended the useful life of critical applications and provided excellent guidance and training for new hires. It doesn't take much imagination to see that this is one of the most valuable people in the entire organization and should receive monetary rewards and career opportunities accordingly. If, for example, most data processing

organizations are spending the largest portion of their budget on software maintenance, then an effective data processing manager is one who has demonstrated that s/he can control this activity successfully.

CONCLUSION

Each modification made to a software system carries a risk of weakening it through the introduction of defects or the compounding of psychological complexity or both. As systems become more complex and defect-ridden, they become more costly to maintain. A data processing organization will accomplish its mission more effectively if it is able to prolong the life of the software it supports.

Solutions to the application systems dilemma include: the retention and maintenance of design documents; the conducting of dynamic system tests and the conducting of static tests. Static testing assumes the form of team walkthrus where the maintenance programmer can share responsibility, maintain external documentation, educate others in the functioning of the system, and minimize entropy.

.

The methods of controlling the solution include: conducting user surveys; measurement of the maintenance effort;

measurement of insertion of defects; measurement of system reliability; establishment of proper criteria by which to evaluate maintenance performance; and creation of a separate maintenance group where motivation and incentives are consistent with talents and interest.

REFERENCES

1. Martin, James and McClure, Carma. Software Maintenance: The Problem and Its Solutions. New Jersey: Prentice Hall, Inc., 1983.
2. Zvegintzov, Nicholas. "Nanotrends." Datamation, August, 1983, pp. 105-116.
3. Fagan, M. "Design and Code Inspections to Reduce Errors in Program Development." Writings of the Revolution. New York: Yourdon Press, 1982.
4. McCabe, T. Software Quality Assurance: A Survey. Maryland: McCabe & Associates, 1980.
5. Yourdon, E. Structured Walkthrus. New York: Yourdon Press, 1978.

6. DeMarco, T. Controlling Software Projects. New York: Yourdon Press, 1982.
7. Brice, L., Connell, J. and Taylor, J. "Deriving Metrics for Relating Complexity Measures to Software Maintenance Costs." Proceedings of the 1982 Computer Measurement Group International Conference, pp. 134-141.

FIGURE CAPTIONS:

Figure 1. Breakeven/Payoff

Figure 2. System Responsibility

Figure 3. User Satisfaction Survey

**Figure 4. Acceptable Performance Criteria for
Maintenance Work**

system_id	system_name	primary	backup1	backup2
12	payroll	hastings	smith	mcdonald
20	commitments	tompkins	johnson	zeindt
25	materials dist.	benjamin	garfunkle	conners
30	employee info.	hastings	temple	roberts
36	travel	hunker	lowe	stamp
40	stores	halpert	garfunkle	albertson
70	general ledger	lowe	davis	hunker
71	accounts payable	zeindt	schutz	tompkins
75	operating plans	marks	wacker	lake
79	operating budget	schmidt	lake	wacker

The following systems are supported by ADP for your organization. Please fill in the blanks rating ADP services using: 1 = poor, 2 = below average, 3 = average, 4 = above average, 5 = excellent.

ID	SYSTEM NAME	PROGRAMMING			OPERATIONS	
		DEV. QUALITY	MAINT. QUALITY	QUANTITY	PROD. CONTROL	DATA ENTRY
71	ACCTS PAYABLE	<u>3</u>	<u>3</u>	<u>4</u>	<u>2</u>	<u>5</u>
12	PAYROLL	<u>5</u>	<u>4</u>	<u>4</u>	<u>3</u>	<u>2</u>
30	EIS	<u>4</u>	<u>4</u>	<u>3</u>	<u>3</u>	<u>2</u>
20	COMMITMENTS	<u>1</u>	<u>4</u>	<u>3</u>	<u>3</u>	<u>4</u>
70	GENERAL LEDGER	<u>1</u>	<u>4</u>	<u>3</u>	<u>2</u>	<u>4</u>
36	TRAVEL	<u>4</u>	<u>4</u>	<u>4</u>	<u>3</u>	<u>3</u>

Criteria Description	Measurement	Satisfactory Level
-----	-----	-----
1. Encourages a free exchange of ideas. Gives and accepts criticism and comments.	Walkthru reports	Effectively participates structured walk thrus.
2. Contributes in a positive manner to user satisfaction with production systems.	User Survey	User satisfaction does not deteriorate over time.
3. Makes changes which do not cause systems to be more difficult to maintain.	Maintenance effort statistics	Effort required to make changes does not increase.
4. Makes changes in a manner which tends to increase the reliability, operability and useability of systems.	Aborts, reruns and trouble calls	Problem incidents decrease over time.
5. Makes changes in a manner which tends to preserve the functionality of the system.	Number of user requests for enhancement changes. *	Receiving rate of incoming enhancement service requests does not increase.

* Note that a burst of changes may indicate a need for a new system.