Redocumentation: Addressing the maintenance legacy

by GARY RICHARDSON and EARL D. HODIL
Texaco Inc.
Houston, Texas

ABSTRACT

Over the past decade or so there has been much attention paid to techniques and methodologies to produce high-quality systems. A concurrent development has been the emergence of software tools that aid in the production and maintenance of software systems; yet the maintenance environment continues to be littered with poorly written and poorly documented programs.

The focus of this paper is to outline a conceptual approach to the allocation of software maintenance resources and the role of automated tools in this process. It is contended that software maintenance tools cannot be simply purchased or built and then used indiscriminately. Rather, it takes an administrative activity to quantitatively decide which code units are best for resource allocation. Finally, to demonstrate the utility of this approach, a case study based on the author’s experience is presented.
THE MAINTENANCE LEGACY

Over the past decade or so much attention has been paid to techniques and methodologies to produce high-quality, maintainable systems. Yet DP management still finds itself left with a swelling production library containing a hodgepodge of code that shows little resemblance to what we now define as good.

In the late seventies Dr. Gerry Tompkins of UCLA surveyed 120 DP organizations. This survey found the mean age of installed systems to be nearly five years and the average size of these systems to be approximately 23,000 lines of source code. A review of the typical production library often reveals high levels of poorly written code with inadequate documentation, a statistic that is not surprising when one considers the time-consuming, laborious nature of manually producing high-quality code that is also well documented. This impetus has stimulated the recent proliferation of software maintenance tools.

The author believes that structured code, clear mechanical format, and other such forms of architectural definition are positive when produced at reasonable cost. Studies indicate somewhat conclusively that structured programming can lower maintenance costs. One point, however, is becoming increasingly clear. That is, methodologies and tools in and of themselves will not automatically correct all the errors of the past. Indeed, the new techniques can become costly and ineffectual if they are used randomly. Our challenge here is to describe a rational approach to correcting this maintenance legacy by proper allocation of resources, including a growing set of software tools designed to aid in this process.

PROBLEM DEFINITION

The road to reduced maintenance effort begins with the answers to two questions:

1. Which programs abend most frequently?
2. Which programs, though they may run perfectly, are so poorly written and/or documented that they cannot be easily changed?

The significance of these two questions is considerable when one considers that two of the essential activities associated with software maintenance are correcting program errors and implementing user-requested changes to software. Even though many firms have recognized the need to answer these questions, most large DP shops have found the quest arduous.

Surprisingly, many organizations find the first question difficult to answer. They can neither locate nor statistically quantify their production source code, much less begin to describe quantitatively which code units could be classified as good, average, or poor. This situation must be resolved before subsequent steps, outlined below, can be undertaken. The three administrative systems following can aid in this process.

Library Control

An automated control package to insure that all production source code is located in approved libraries and that production load modules contain only these source modules. Though there are many reasons for installing such a system, its purpose is to bind the execution errors associated with executing a load module to the source code responsible for them.

Operations Logging

A tracking system that traps all production jobs and records completion status (e.g., good completion, space abort, JCL error, bad completion code). This tool should provide execution information at least down to the load module level.

System Profile

A text-oriented system, summarizing basic system metrics such as

1. age,
2. language,
3. total lines of source code,
4. user evaluations of the current system,
5. future enhancement plans at the aggregate level.

By using these three techniques it is possible to identify the target code population accurately, then array the code units according to abort frequency.

Phase 2 of the problem definition activity begins once operational statistics are available regarding code performance. It is then necessary to divide code units into three broad categories:

1. Good Code—low abort frequency
2. Bad Code—high abort frequency
3. Marginal Code—borderline abort frequency

Here we are left with both a philosophical and a technological problem. Philosophically, we may believe that well-written code has a low abort history and vice versa. Alternatively, some believe that abort history is independent of code
structure. It is observed that some systems require highly skilled operational support personnel and code modifications; are complex, owing to a lack of a coherent design architecture; yet are stable, judging by abort statistics. It is the authors’ opinion that the subject of good versus bad code is multidimensional, involving both mechanical and operational factors. The maintenance function involves both aspects of operation and enhancement; therefore goodness of code must involve more than one view. A second philosophical issue surrounds the idea of documentation value. When one looks at the millennia of existing production code without supporting documentation, some doubt must exist about whether it is of value to be concerned about such things. In attempting to rationalize such behavior there is at least the obvious conclusion that the cost of documentation production outweighs its value. The authors believe that an automated approach to producing documentation improves both software accuracy and cost effectiveness.

Now for the technical problem: It is theoretically possible to quantify abort frequency and arbitrarily divide code units into good, marginal, and bad categories; however, we have already said that this is not enough. There are at least two other code grading technical issues that should be addressed. First, code complexity needs to be evaluated. McCabe and others have defined quantitative measures of code complexity, although once again there is no broad agreement about when a code unit is too complex. Indeed, some productive code requires complexity; and in some cases it is rationally added to the code architecture for efficiency or other reasons. In any case, high-complexity index values could be warnings to review an existing code unit and decide whether it is feasible to simplify it in some way. A third aspect of the technical problem is the question of the code architecture of the code unit itself. This is manifested by unstructured or large modules. Within this realm one might attempt to review style, language, structure, size, and existing documentation of the unit in order to supply a qualitative grade. The final aspect of code review requires judgment about whether the code should be a candidate, based on strategic objectives. For example, if an old batch system is being replaced in less than one year with a new online system, then it makes sense not to give that code any extra support. Alternatively, an old system with no upgrade planned would be a candidate. This activity is designed with a view to future evaluation.

We have indicated that in order to effectively allocate maintenance resources it is necessary to quantify where current operational problems now exist through formalized abort history statistics. In addition to this we should provide some type of grading scheme at the code unit level to identify potential modules for which resources can be profitably allocated to repair. It is feasible to use automated tools to do much of the scanning work for items such as size (lines of code), complexity, adherence to code standards, and other related functions. After all the automated statistics are summarized it should be possible to select high-priority targets for closer manual examination. From this aggregation of data it is then necessary to select and rank code units to be given special consideration for rework. Some day this process can be highly automated; however, it currently will involve a high degree of subjective judgment.

THE PURIFICATION PROCESS

We have outlined an analytical process designed to identify systems and code units (i.e., programs) that are candidates for rework. The key question now is, "What do we do with the subset of problem code defined?" Figure 1 shows schematically the process described above. Note that two new items show up at the bottom of the figure, rewrite and redocumentation. Each of these deserves more discussion here. Rewrite represents code units in such shape that manual rearchitecture of the system is required to resolve the indicated problem. Typically this means that new functionality is required or that the basic database design approach is flawed. Obviously placement of code in this category should be done only as a last resort because of inherent cost and time to accomplish.

The second form of code repair is automated redocumentation, which is defined as the software-driven process of producing documentation for existing code directly from the syntax itself. Elshoff and Marcotty from General Motors have documented their company’s approach to the use of similar automated techniques to improve code readability and modification. We feel that these tools are most useful when used as an aid to the maintenance programmer who is trying to draw understanding from a block of unyielding (and usually undocumented) source code. These tools may be categorized as follows:

1. Dynamic analyzers
2. Static analyzers
3. Restructure/recoding tools

Dynamic analyzers have long been accepted as a part of the maintenance programmer’s workbench. Debugging compilers and interpreters compose this group of tools. Usually, the dynamic analyzer is used in conjunction with test data during an interactive session. Features commonly associated with dynamic analyzers are (1) fast syntax checking, (2) one step

![Figure 1—Decision schematic for production code](from the collection of the Computer History Museum (www.computerhistory.org))
compile and run, (3) program path tracing, (4) execution suspension and restart, and (5) variable dump and modification.

The difficulty with this method of analysis is that it considers only the paths traveled by the selected test data. Dynamic analysis is, therefore, analysis by trial and error. It is best suited for the investigation of a particular test case or a limited set of test cases, not for gaining an all-path understanding of a program.

**Static analyzers** are more of a newcomer to the maintenance environment. To be sure, flowcharting programs have existed for some time. Yet the flowcharting program merely provides a rehashed version of program logic in graphic form. In the output of a typical static analyzer, we see the beginnings of an attempt to unravel program logic. Moreover, static analysis can provide useful information regarding program style and complexity.

Yet of all the tools now available to maintenance programming, the restructuring/recoding tools are surely the most exciting. They combine the intelligence of the static analyzer with the ability to generate code. Unstructured code (i.e., code with GOTO statements) is the input to this tool. The tool analyzes the unstructured code and produces a structured version. Collectively, this family of tools represents our central focus here.

**THE ECONOMICS OF REDOCUMENTATION**

We believe that automated redocumentation is the preferred alternative for code repair. For some justification of this let us first look at the resource economics involved in the code repair decisions.

Type I and II code (see Figure 1) represent the code library that is to be essentially left alone. For this segment of the library it is generally possible to allocate resources at the rate of one maintenance programmer per 40,000 to 70,000 lines of source code (independent of the language). This allows for a small amount of enhancement but generally provides for very little extra resources for more than daily operational requirements. Obviously, numerical guidelines such as this need to be validated locally before extensive reliance is placed on them. For the Type III subset, it is a truly complex job to specify an appropriate level of resource allocation. In many DP organizations, the aggregate resources dedicated to the maintenance function can range from almost 90% to as low as 30%. A proper number lies only in management’s eyes and is closely tied to a general philosophy of maintenance. We are suggesting that at least 10% of the maintenance library has been neglected. Various studies, reported by Jones at IBM and Hermann at Shell Oil and others, document the development cost of systems at values ranging from $5 to $50 or more per line of code produced. Our experience, however, is that automated documentation can be produced at a cost of between $0.20 and $2.00 per line. This represents a cost ratio of 25:1! In stable database situations the redocumentation strategy is often viable and cost effective. A small allocation of resources can produce dramatic results for properly chosen code units. It is true that even more dramatic improvements can be made through the rewrite process. However, the allocation of resources is concomitantly much higher; and the benefit often occurs much later, after an extended development cycle.

Having now examined how to identify targets for profitable use of redocumentation tools and the economic rationale for using automated redocumentation, let us turn to a case study, drawn from the authors’ own experience, to demonstrate the utility of this approach.

**A CASE STUDY**

Texaco Inc. is typical of many large DP operations and recently faced the problem of rising maintenance costs. There were a large number of diverse applications, each with its own maintenance staff and procedures. Also, like many DP organizations, Texaco had invested a considerable amount of money and staff time in learning to use new design technologies and tools. These efforts notwithstanding, many staff members felt that the level of effort expended on maintenance was still too high, primarily because of the large volume of old, poorly written code that had existed before the new methodologies were implemented.

To quantify the actual maintenance effort, functional applications were manually inventoried. This inventory confirmed the previously held suspicion that approximately half of the professional programming staff worked on maintenance. Because of the increasing backlog of new applications and enhancements to existing systems, and because of the omnipresent goal of holding costs to a minimum, this situation was deemed unacceptable. Early schemes to reduce this effort called for the mass redocumentation of all the production libraries. However, despite the relative cheapness of these tools, cost-benefit estimates precluded the use of this tactic. Hence it was decided that particular systems and subsystems would be targeted for rewrite or redocumentation.

First, manual methods were used to identify the relevant applications. Two points became apparent as this process was carried out: (1) manual code reviews were too time consuming, and (2) manual records of abends were difficult to organize.

It was decided to expand the use of automated tools to address these problems more effectively. In addition to the previously stated features, an automated library management system was required to improve control of source and load libraries across multiple sites. Having unsuccessfully searched the outside software market for an integrated tool that would meet these requirements, it was decided to create a custom library management system, LIBMAN. LIBMAN is a control system using the services of several existing software tools (SPF, VTAM, PANVALET, ACF2, etc.) to provide control over both the repair and enhancement of production programs. The operational logging system used for the actual identification of problem programs was the MVS Integrated Control System (MICS) from Morino Associates, Inc., which gathers information from diverse sources such as SMF and TSO/MON. This information was then collected on a SAS database from which reports on code unit performance were derived. Finally, profiles were created to assist in the process of describing current systems. Originally a manual effort, this
system has now been converted into an online one, using DATAMANAGER as a repository.

After the administrative-level systems were in place and the code universe was well defined, it was possible to identify code that was structurally poor. This subset of the code population became the target code, which would be examined in more depth. Through the process outlined earlier, some of these code units were amenable to automated redocumentation. At this point several automated tools were applied to the selected programs. First, for the COBOL systems an outside product, SCAN/370 from Group Operations, Inc., was selected. SCAN/370 produces a report that traces all the logic paths of a given program. This program also provides a source listing containing imbedded path data, complete with identification of dead code.

Later a restructuring/recoding tool for COBOL source programs became available. This program, called SUPER-STRUCTURE (also by Group Operations, Inc.), creates a scorecard that identifies unacceptable program flaws such as (1) interparagraph GOTO statements, (2) run away paths, and (3) fall-through execution of paragraphs. Having created the scorecard and identified the paths of a program, SUPER-STRUCTURE rewrites the program paths using only structured constructs (sequence, iteration, and selection). The resultant source code contains essentially none of the flaws of the original source program.

Most of the company's developmental programming is produced in PL/1. Though the language itself contains elements that may encourage good programming style, a number of older systems were found to be inconsistent and were difficult to modify. A significant review was undertaken to find analyzers and documentors that fit a PL/1 development environment. Unfortunately, no vendor-supplied tool was found that would be compatible with the current methodologies, so an in-house tool was developed. The tool, TEXIT, conducts static analyses of program paths via code scanning and renders several forms of documentation:

1. Complexity measures
2. Jackson style structure charts
3. Module hierarchy charts
4. Annotated source code

The next documentation tool selected was a system redocumentation tool linked to JCL. This tool, DOCU/TEXT from Diversified Software Systems, Inc., was tested on a few selected applications; and it appeared that it could be used on all the JCL libraries. This was in marked contrast to the way the other tools were used, but in this case it seemed to be feasible. Our evaluation is that system-level tools of this type can be used to identify code that is not used, or to change existing code. The tool, TEXIT, conducts static analyses of program paths via code scanning and renders several forms of documentation:

The scorecard and identified the paths of a program, SUPER-STRUCTURE rewrites the program paths using only structured constructs (sequence, iteration, and selection). The resultant source code contains essentially none of the flaws of the original source program.

CONCLUSION

There are many disjointed software tools on the market today, and more are emerging daily. Various combinations of these tools will fit unique organizations. We have attempted to outline an approach to the selection of target code units and general types of tools that collectively aid in the maintenance function. A most important conclusion resulting from our experience is that tools cannot be purchased or built indiscriminately. Rather, they require an administrative activity to identify which code units are best for the resource allocation. Then, management has to support these efforts with rational levels of resources designed to "purify" production libraries. Even more pertinent, it requires a high level of management focus to cause the process to occur in an orderly manner. Within the software tools marketplace we anticipate more innovation in the area of automatic restructuring/recoding. It seems inevitable that artificial intelligence (expert systems) may lead the way in this area. One possible way to implement such a scheme would be to create an expert system that is well versed in one of the popular design methodologies (Jackson, Yourdon, etc.), give it access to the path information provided by static analysis tools, then restructure accordingly. Once this can be successfully done, the family of redocumentation tools will become more coherent.

Whatever the case may be, it is probable that tools will continue to play an increasingly visible role in the maintenance of software systems and will require continued management effort to keep them cost effective.

REFERENCES