EDP ANALYZER
Editor and Publisher: RICHARD G. CANNING

EDP ANALYZER

AUGUST, 1981
VOL. 19, NO. 8

EASING THE SOFTWARE MAINTENANCE BURDEN

Software maintenance still requires a significant portion of programmers' and analysts' time, in most data processing departments. But perhaps because software maintenance tends to be viewed as an uninteresting, necessary evil, it often appears that nothing can be done to reduce this workload. And, in fact, a 1978 survey showed little change in the overall maintenance workload during the 1970s. This month we look at software maintenance to see whether there are any new ideas on how to ease this maintenance burden. We begin by describing three different approaches taken by three companies for creating more maintainable systems.

The Monsanto Company, with headquarters in St. Louis, Missouri, is a leading manufacturer of textiles, chemicals, plastics, resins, and agricultural products. Annual sales exceed $6 billion, and the company employs about 64,000 people. In its central data processing facility in St. Louis, Monsanto uses large scale IBM computers.

In the late 1970s Monsanto began studying their need for a corporate-wide medical and environmental health information system (MEHI). On a joint venture with IBM, they identified seven major components. Three are: a materials system that contains information about all of the materials (both raw and manufactured) that Monsanto uses, an occupational exposure system that lists all employees and the substances they are exposed to in their work at Monsanto, and an emergency system that lists which employees to contact for each identifiable emergency and how the company could respond to these emergencies.

For the first sub-system to be developed, the materials system, Monsanto chose to use IBM's conversational Application Development Facility (ADF). The prime value of ADF, in Monsanto's eyes, would be in reducing development time. But a significant side benefit has been in decreasing program maintenance. Although ADF is not as efficient to run as, say, COBOL programs, Monsanto felt that with the expected low processing volume of the materials system, ADF would perform satisfactorily, so its use appeared cost justified.

ISSN 0012-7523. Multiple copy prices listed on last page. Photocopying this report for personal use is permitted, providing payment of $2.50 fee *per copy of report* is made to Copyright Clearance Center, Inc., 21 Congress Street, Salem, MA 01970; please include identifying fee code 0012-7523/81/080001-14$2.50.

ADF is an IBM application development tool for helping programmers create IMS/VS applications—applications which use IBM's IMS database management system under the VS operating system. ADF offers a conversational mode, a non-conversational mode, and a batch mode.

Creation of an ADF program is quite different from conventional programming. The programmer does not create a program *per se.* Rather he/she specifies parameters for common modules. Hand-coded subroutines also can be added. The parameters are stored in certain databases; data used by the application is stored in other databases. At execution time, ADF invokes the common modules in a pre-programmed order, and combines them to create a distinct program. (For more information on ADF, contact your local IBM sales office.)

Monsanto chose conversational ADF for its on-line materials system for several reasons. First of all, the security option, which was a major concern, was adequate and involved significantly less work to set up than would a COBOL IMS program. In addition, changes to the security network would be a relatively minor task. Second, conversational ADF could automatically generate menus of data items to aid users in searching a database. These two features contributed significantly to decreasing the project's development time and costs. The project took nine months less time to complete than if it had been written in COBOL—and that was a 75-80% reduction in programming and test time. An additional advantage of conversatonal ADF was that Monsanto's programmers learned to use it quickly.

ADF was able to handle thirty-four of the forty-seven transaction types required in the materials sub-system. For these, programming and testing took three days or less per transaction type. For the thirteen remaining transaction types, which needed to be hand-coded, each took six days.

ADF has also significantly reduced the amount of maintenance required by the system. In the first twelve months of use, there were *no* abnormal program terminations due to programming errors; all the things ADF did, it did correctly. So Monsanto has had virtually no corrective maintenance on the materials system.

As for enhancement maintenance, adding new transaction types and new features requested by users has been far easier than with COBOL. Monsanto is now using ADF on a second MEHI subsystem, because they find that it produces reliable systems and increases their programmers' productivity as well, for both development and maintenance work.

Western Carriers Underwriters

Western Carriers Underwriters is a four year old company that specializes in commercial automobile insurance for the transportation industry. Western Carriers is located in San Bernadino, California, just east of Los Angeles, and employs 50 people. Operating through independent agents, in 1980 they had insurance premium sales of $8.5 million.

In late 1979, Western Carriers had (essentially) a mini-computer, not much software that really met their needs, and three people in their data processing department—a manager (who also does a lot of the programming), a programmer analyst, and a data entry person. Today, a year and one-half later, they have the same three people, a larger model of the mini-computer, and four main-line applications that involve over 200 programs with some 500,000 lines of COBOL code.

The computer they now have is a Prime 550 computer with 1.25 million bytes of internal memory plus one 300 megabyte and two 80 megabyte disk drives, two printers, and eight work-stations. But the product that enabled Western Carriers to create and maintain so much software with so few people is a COBOL program generator obtained from David R. Black and Associates (see Reference 6). Western Carriers obtained the generator in December 1979, and the two programmers have created 75% of their software using it. The generator provides a data dictionary, a screen formatter, a menu generator, and a number of skeleton COBOL programs for performing the most common functions found in business applications.

For each of the application programs, the generator leads programmers through entering data definitions and validation rules. To perform more elaborate validation checks not provided

by the generator, the programmers wrote short COBOL routines which they imbedded in the validation routines created by the generator.

With the data dictionary filled in, the programmer then specifies the data entry screen formats, reports, and menus, using different facilities within the generator.

The generator allows Western Carriers to create new programs quickly. But even more importantly, it has a greater impact on maintenance. For one thing, Western Carriers has found *no coding errors* in code generated by the program generator, so that most of their programs compile completely on the first try. The only coding errors are in the code they write to supplement the generator's code. The generator can create 70% of the code they need for their major programs, and 100% for the ad hoc requests.

After using the generator for a year and a half, they know how the generator will handle most situations, so they know what effect changes will have. Also, the generator produces modular code, so that changes tend not to spread from one module to another. The result thas been that Western Carriers has practically no corrective maintenance.

However, they do have enhancement maintenance. Although they generally do not use the generator for making these enhancements, the time it takes to perform this maintenance has been greatly reduced because of the generator. The generator creates programs using a standard format, so the programmers often do not need to print out a listing in order to figure out where to make the changes. They simply go to a terminal, call up the COBOL source code subroutine which performs the desired function, and make the change. All programs have the same subroutine structure and same conventions, so maintenance is fast. At first Western Carriers thought that standardization was a nice feature. Now they see it is the most important feature of the generator, especially for maintenance.

As an example, in January of this year, a change in the billing procedure led to a major maintenance project. All of the policy number fields had to be split into three separate fields, which required changes in all programs that use the policy number field. Fortunately, because the generator always uses one subroutine to reference this key index field, the programmer only needed to change this particular subroutine in each affected program. So this major maintenance project was conducted without their needing to hire more people.

Since the standard programs allow Western Carriers to perform changes so quickly, they have been able to respond to management's and end users' requests for enhancements very rapidly. In fact, the data processing manager has had to implement a more formal change control procedure to slow down somewhat the requests for enhancements.

The people at Western Carriers are delighted that the program generator allows them to have a small, efficient data processing operation. They estimate that without it, they would probably have required seven more people (project managers and programmers) just to create the systems they now have, not to mention maintaining them.

The Arizona Bank

The Arizona Bank is a full service bank with headquarters in Phoenix, Arizona. The bank has assets of $2 billion and employs 2200 people in 91 branches throughout the state.

Within the past few years, new government regulations for the banking industry have prompted the bank to offer interest-earning checking accounts and new types of loan arrangements. They have also installed a network of automated teller machines and an on-line teller and administrative terminal system. These operational changes have required their data processing department to install a number of new, sophisticated systems as well as make extensive changes to existing systems. The bank has an IBM 3033 and they use IBM's IMS database management system.

For the size of the bank, and the complexity of its application systems, The Arizona Bank has a small data processing department—only fifty programmers and analysts. This is because the bank has a policy of buying application packages rather than developing systems in-house. Cur-

rently, *90% of their application software consists of purchased packages.*

Only two of their banking applications—installment loans and savings—were written in-house. These were created more than fifteen years ago and are still being used. The rest of their banking applications are purchased packages. These include the rest of their banking applications as well as accounting systems, several electronic funds transfer systems, and others. In all, the bank uses packages from fourteen vendors.

Due to their policy of relying on purchased software, the application development cycle at the bank differs from the typical software development cycle. Whenever a new system is under consideration, a project study team is formed to analyze the make-or-buy alternatives. The teams work anywhere from one to six months studying the alternatives. Due to time and cost considerations, they almost always recommend purchasing an outside package.

Of particular interest is how this policy of procuring purchased packages impacts software maintenance. The story really begins with the work of the study teams.

The work of the study teams. The study teams generally consist of several people from data processing as well as several user representatives. Members of the team search out packages that might fit their needs by using several industry publications, by attending banking conferences, and by contacting people they know in other banks. Also, and very importantly, they become involved in vendors' user groups; often they find out about other products at user group meetings. In addition, these groups provide users with an effective means of requesting (and getting) enhancements and changes made to the vendor's products. This is the best way to get a change made by a vendor, we were told.

Team members are responsible for performing specific duties depending upon their areas of expertise. For example, the data processing operations person evaluates the impact of each alternative upon the bank's hardware. The IMS person looks at how much work would be involved in interfacing each alternative package to the IMS database system. And the user representatives study the user documentation and functions performed by each alternative package.

Based on management policies, they require that they receive from the supplier a package's source code, not object or intermediate code, so that they can create proper interfaces for the package. Also, with some exceptions, they require that the application packages be written in COBOL.

When evaluating application packages, one important factor that the teams consider is how much 'maintenance' each package will require. Their package maintenance takes several forms: (1) adapting a package to the bank's environment, (2) adding desired capabilities that are not included in the package, (3) correcting errors, (4) installing vendor changes in the future, and (5) making user-requested modifications. In particular, technical experts on the team try to determine how much in-house effort will be required to interface each package to their existing databases and systems.

The team also identifies required functions and features that are not included in each alternative package, and how the deficiencies can be remedied—either through in-house modifications, vendor-made changes, or a joint bank-vendor contract. And the team estimates how much ongoing support the vendor will supply after the package is installed, and how vendor changes and updates will be made.

The bank's policy is to make as many of the changes as possible by creating new stand-alone program modules which work with the purchased software. In this way, the bank is able to install new releases of the package without too much maintenance work on their part. Most often, however, the bank and the vendor perform maintenance jointly.

As a point of interest, the bank's data processing people estimate that 50% of the in-house requests for changes have been for new types of reports. Therefore, they look very favorably on packages that provide report generator capabilities with which end users can create their own ad hoc report requests. Where such systems have been installed, there are fewer user-requested changes.

The Arizona Bank is pleased with its approach of purchasing most of its software for several reasons. Of most interest, of course, is getting applications set up and running more quickly—and with perhaps more sophisticated applications than they could have developed in-house.

On the point we were probing—maintenance—they have found substantial benefits also. As described, they minimize routine corrective maintenance and concentrate on adapting and enhancing packages.

Is software maintenance changing?

Within the past two years, we have seen an increasing amount of written material on software maintenance. Of particular interest are three books on the subject. *Software Maintenance Management* by Lientz and Swanson (Reference 1) describes their findings about software maintenance from a 1978 survey of 487 companies. *Techniques for program and system maintenance*, edited by Parikh (Reference 2), is a compilation of forty papers on the subject of software maintenance. And *Software maintenance guidebook* by Glass and Noiseux (Reference 3) discusses the people, technical, and managerial aspects of maintenance and presents the authors' views on which techniques and tools they have found most helpful for maintenance programming. We will reference these three books throughout this report.

In 1978, Lientz and Swanson, professors at the University of California at Los Angeles, performed a study to uncover the characteristics of software maintenance, mainly in the business data processing environment.

They found that on the average, software maintenance required 45% of systems and programming resources in most companies. This figure is about the same as the authors found two years earlier in a similar study. And it also matches the estimate we gave in our October 1972 report on maintenance. Therefore, it appears that the *total* maintenance effort is not increasing, as often reported in the press, nor is it decreasing, as predicted by many proponents of programming tools. However, in our research we did find that the characteristics of maintenance work have changed for the better, since our 1972 report, where companies have developed more maintainable systems.

To better understand software maintenance, consider the definitions created by Lientz and Swanson. They divide maintenance into three components: corrective, adaptive, and perfective (enhancement).

Corrective maintenance consists of dealing with failures in processing, performance or implementation, such as fixing errors and routine debugging. The authors found this type of maintenance generally accounted for about 20% of most companies' maintenance efforts.

Adaptive maintenance includes responding to anticipated changes in the data or processing environment. For example, it includes converting to a new operating system, responding to changes in government regulations, and so on. This type of maintenance accounted for about 25% of the maintenance effort, the authors reported.

Enhancement (or perfective) maintenance, refers to (1) enhancing processing efficiency, performance, and system maintainability, (2) improving program documentation, *and* (3) making enhancements for users. This form of maintenance accounted for the remainder (about 55%) of all maintenance work, and almost two-thirds of that went toward making enhancements requested by users—specifically, giving users more capabilities.

In our 1972 issue on maintenance we described some techniques for creating more maintainable systems. We will now review these points to see how much impact they have had on easing software maintenance.

In 1972 we recommended *designing for change* by installing software design standards and documentation standards. The rationale is that if applications are designed using the same basic company-wide standards, more people are likely to know the standards and therefore be able to maintain these applications more easily. (A good illustration of this point is the experience of Western Carriers Underwriters, described earlier.)

Since 1972 there has been a lot of activity in several of the software design and documentation areas. Numerous software design techniques

have been introduced, each offering its own form of standards. Yet at the time of their survey (1978), Lientz and Swanson found only three development tools in use in more than 30% of the companies. Surprisingly, one was decision tables; they were used by 34% of the companies. Chief programmer teams were used by 38% and structured programming by 30%. The other techniques mentioned by the authors—HIPO, data dictionaries, test data generators, and structured walk-throughs—were each used by less than 20% of the companies responding.

Lientz and Swanson found that use of such techniques *did* lead to more maintainable systems. They therefore speculate that organizations which use such techniques for software development may reduce the amount of *corrective* maintenance that they must perform, because there are fewer errors to correct. *However*, the relative ease of making additions and changes to these more maintainable programs, and the lower risk of creating errors through such changes, often encourages users to ask for more enhancements. So enhancement maintenance often rises. This explains the relatively constant total spent on *total* maintenance, say the authors.

In 1972 we also suggested installing *software modification and testing aids*, such as cross reference listings, on-line editing and debugging facilities, standard testing procedures, and generalized data management systems.

Each of these types of aids is being used, but the area in which we are just now seeing increasing interest is in the last named—the generalized data management systems (DMS). Within the past few years, these products have been enhanced substantially so that they now allow creation and modification of files, data definitions, and programs. We have discussed these products in some detail in recent months, particularly in the May, June and July issues.

One question these DMS raise is: how might they affect maintenance work in the data processing department? One company we talked with has said that although enhancements and changes are much easier to make, their backlog for such requests has increased. So the same thing may occur with the use of DMS as was mentioned for structured programming techniques—that is, corrective maintenance decreases but enhancement maintenance increases.

In 1972 we also suggested establishing data processing *configuration policies*. These consist of controlling changes in the organization's hardware, operating systems, utility software, and so on.

There appears to be an interesting phenomenon occurring in this area. Companies, by and large, have developed configuration policies for their mainframe computers. And computer mainframe manufacturers and vendors of products for mainframes are very aware that companies have large investments in existing applications—so they often include 'migration tools' when they introduce new hardware and software products.

But what about company-wide configuration policies for mini-computers, and even more recently, for micro-computers? Are users and vendors worrying much about configuration standards and migration tools for these machines? The answer appears to be mostly No. We see a few user companies trying to establish standards and guidelines for these smaller machines. However, we also see many companies not really worrying about the stand-alone systems that departments are bringing in on their own. We think this second attitude is a mistake.

While local networks *may* provide the means for connecting incompatible machines, we think configuration policies should be created for smaller machines for two reasons. One is software portability. Creating software for small machines is expensive, so it is wise to exchange programs where possible. The second reason is for compatibility among data used by several departments—portable data.

Companies should also evaluate these smaller systems with an eye toward whether they allow easy migration to larger machines—not only program migration but also data migration. This can be a serious problem, as we discussed five months ago (March). Not all small machine vendors provide easy vertical migration.

In 1972 we also recommended instituting several types of *controls and audit procedures* including: a formal change procedure, the use of a librarian software package for controlling dif-

ferent versions of a system or application, compiler pre-processors to check for standards violations, audits of the documentation system, and design review procedures.

Lientz and Swanson found organizational controls more widely used than the development techniques mentioned earlier. For example, 79% of the companies logged and documented users' requests for maintenance, and 77% logged and documented changes made.

These widespread practices did not appear to reduce the maintenance effort, say the authors. But the practices did indicate where management placed its emphasis—that is, requests were logged where companies stressed enhancement maintenance.

The one procedure that reduced corrective maintenance (by improving software quality) was formal audits of production systems. But only one-third of the applications described by respondents received such scrutiny.

And lastly, in 1972 we recommended *organizing for maintenance.* The question has been pondered whether to perform maintenance work within the development group(s) or to create separate maintenance groups. Most organizations still use a common organization for both development and maintenance, partly because of their concern over the low morale that might occur in a dedicated maintenance group. Management believes that most programmers do not like maintenance work and will not stay long in a maintenance-only group. Reasons typically given are: maintenance is not challenging, rewarding, or creative, and it requires less skill and experience.

Lientz and Swanson found that only 17% of the companies responding to their survey had separate maintenance groups. Yet these few organizations tended to use fewer resources for maintenance. These companies also were the larger organizations, and larger organizations generally spend a larger percentage on maintenance. So these companies may actually be doing even better than it first appears.

Aside from this dilemma, the real issue is not which alternative is better, but rather that the subject of maintenance be given any serious attention at all by data processing management. Ignoring the maintenance problem appears to make it worse. Establishing specific procedures to deal with it seems to ease the burden.

These then are the recommendations we made in 1972 to ease the maintenance workload. The major change appears to be that companies now have more tools with which they can create more maintainable systems. These tools have reduced the need for corrective maintenance in newly-created systems but, at the same time, they have encouraged enhancement requests from users.

Prospects for easing maintenance

In light of what has happened during the past ten years, here are four steps that can be taken to improve software maintenance: fine tuning the organization, fine tuning maintenance procedures, programming for maintainability, and offloading maintenance onto others, either end users or package vendors.

Fine tuning the organization

In the Lientz and Swanson survey, the respondents ranked managerial concerns above technical concerns, so we will deal with management issues first.

We have come across success stories for separate maintenance groups, combined maintenance and development groups, and variations of these two approaches. This implies that the type of organization may not be the important factor, but rather that conscious management attention is. Here are some examples of how a few companies have eased the software maintenance burden through fine tuning their organizations.

A separate maintenance group. One company that has had success with a formally separated maintenance group is Spring Mills, Inc., a textile manufacturer in South Carolina. Mooney (in Reference 2) reports that prior to the separation, maintenance was not a budgeted item. Development projects were interrupted to correct deficiencies in production programs, make revisions caused by changed business practices, and improve program efficiencies. These were all performed as needed. Major enhancements to existing systems were scheduled along with develop-

ment work. The staff of forty handled between 70 and 80 maintenance requests a month.

The company decided to tackle the maintenance problem by creating a maintenance team of ten senior and junior programmers under a project leader. The team was to: (1) log all requests in and out, (2) evaluate requests and assign priorities, (3) assign tasks within the team, (4) ensure that standards were met and documentation was upgraded, (5) test new releases of programs, and (6) get programs back into production, as quickly as possible.

Since management feared the team would see its work as distasteful, several special policies were implemented. Team members were to continually and automatically receive the largest merit pay raises allowable, they could request a transfer after six months, and they would be assigned to write special, one-time programs so that their development skills would not deteriorate.

Management was surprised to find that morale throughout the entire department improved with the re-organization. The maintenance people became multi-lingual experts, so they were respected by their peers in development. They learned how *not* to write programs. And they suggested a number of changes to company programming standards and documentation—many of which have been implemented.

The team handled 1000 requests within the first year, and allowed the department to handle almost 13% more new development work than the year before. Maintenance was reduced to 20% of the department's total workload, down from 30% the year before. A year later a system analyst was added to the team, to study how planned changes to programs would affect succeeding jobs or systems. And new systems are now developed with more thought of future maintenance than was true in the past.

Where development and maintenance are organized separately, Lientz and Swanson suggest using 'maintenance escorts.' An escort is a development programmer who accompanies a system when it is transferred to the maintenance group, and he or she stays with it until it is absorbed by the group. The authors note that this procedure takes advantage of the programmer's knowledge of the system and thereby increases the initial efficiency of the maintenance group. The converse of this idea has also been tried—that is, moving a maintenance programmer onto a development project to help complete it and escort it into maintenance.

Combined maintenance and development. When development and maintenance are performed by one group, improved maintenance efficiency seems to come from instituting more formal maintenance practices.

For example, in the April issue we reported on the use of a job diagnostic survey for measuring job satisfaction among data processing employees at one company. A primary point that emerged from the use of that survey was that the company's programmer/analysts had strong negative feelings about the maintenance work they were performing. At the time, each person was totally responsible for maintaining one or more application systems as well as for performing new system development. With users continually asking for changes to existing systems, the programmer/analysts felt trapped; they were being held back from development work because of the continual demands for maintenance. One change that was made was to assign other programmer/analysts to provide backup maintenance support for each existing system. So the maintenance workload was shared.

In addition, a user liaison position was created to be a buffer for maintenance requests, so that only relevant requests flowed through to the software people. Also, more formal procedures for making changes were instituted. Although these changes did not reduce the maintenance workload, they helped ease it in the eyes of the development staff.

In light of the fact that there are organizations satisfied with each of these different arrangements, several points can be made.

First, if maintenance is not to be separated from development, then it is necessary to clearly define the responsibilities of the staff members, so that conflicts between their maintenance and development responsibilities do not create unreasonable pressures.

Second, the flow of user requests, including requests for both ad hoc reports and for en-

hancements, needs to be controlled in some manner.

And third, it seems possible to have a part of the organization devoted to maintenance without severe morale problems. It does require management attention—but it can be done.

Fine tuning maintenance procedures

Numerous practitioners have suggested various ways to add some spice to the maintenance function. We will discuss four ideas: scheduled maintenance, maintenance reviews, worst-first maintenance, and structured retrofit.

Scheduled maintenance. Lindhorst (in Reference 2) describes the benefits that The Boatmen's National Bank in St. Louis has received from implementing a scheduled maintenance program. They have designated specific months for performing maintenance on each of their systems. Most requests for maintenance are saved and then evaluated and costed during those scheduled times. Although the program took several months to initiate, and some reshuffling of schedules was needed, they now have a maintenance procedure that forces users to think more about the changes they request (and their associated costs). It also forces data processing to periodically evaluate the maintenance costs of each system. It has prompted better personnel planning in data processing. And it has eliminated 'the squeaky wheel gets the grease' syndrome. Finally it gives data processing requests for changes equal consideration with user requests.

The bank has found that most changes can be postponed and consolidated with others, leading to more efficient maintenance. They believe they are more in control of their data processing workload with scheduled maintenance.

Maintenance reviews. Freedman and Weinberg (in Reference 2) suggest several types of reviews to improve the maintenance function. One is *speed reviews.* Here a number of maintenance programmers meet together and are given five to ten minutes to evaluate a small section of code, either for its readability or to find errors. The authors find such short, timed reviews are most useful.

They also suggest *maintenance reviews*, which are separate from the traditional review held before transferring development work to production status. If the two reviews are combined, as they usually are, maintenance considerations are given little attention, since they would delay putting the system into production. But if two separate reviews are performed, maintainability considerations can be given more thought.

And finally Freedman and Weinberg suggest *fix and improve reviews.* They say these not only improve the quality of programs but also increase the morale of maintenance personnel. The objective is to not just fix an error but also to improve the software being changed. They note that this requires quite a bit of programming ingenuity, thereby increasing the challenge. And if the improvement is limited to the requested change, maintenance costs do not increase. They have seen this technique measurably improve programs (over time), where *management* has made improved software a goal and has recognized improvement efforts.

Based on information uncovered in a recent survey, Chapin (Reference 4) reports that most of the factors that make programs harder to maintain are under control of data processing management. In his paper, he lists a number of these program attributes, such as arbitrary data names, monolithic program structure, and a large number of switches and flags. And like Freedman and Weinberg, he recommends that maintenance managers give programmers the time and support to improve the programs they maintain which have these attributes.

Worst first maintenance, as described by Weinberg in Reference 2, requires finding out where you are spending your maintenance money in each system. He has found that often 80% of the maintenance work is spent on just 20% of a system. Uncovering this type of phenomenon does take some data gathering, but he notes that such discrepancies are relatively easy to spot once you begin to keep track of where maintenance people spend their time. He has seen several systems greatly improved by uncovering these 'worst' portions and then rewriting them to be more maintainable.

Structured retrofit involves using a software product (sometimes called a 'structuring engine') to analyze a program of ill-defined structure and try to convert it into one of 'well-defined' structure—that is, one that follows certain structured programming conventions. The resulting program should produce the same transformations on input data as the original program.

For COBOL programs, we are aware of one such product, offered as a service by The Catalyst Corporation in Brookfield, Illinois, but we have not had a chance to talk with users of the service. Their service consists of five steps, which can be performed at their site or the user's site. (1) First, an operational program supplied by the user company is put through the structuring program. This product cleans up the existing language and verb usage and also introduces a structure to the code. For example, it reduces GOTOs, removes ALTERs, removes dead code, and so on. Also, it isolates the control hierarchy, highlights looping conditions, and groups and standardizes input/output, etc. It does *not* remove logic errors nor produce functional changes. (2) Next the restructured source code is put through a formatting package to enhance its readability. The formatter indents and formats code, standardizes paragraph prefixes, field alignment, and reserve words, and so on. (3) Then the operational program is recompiled to ensure that no syntax errors have been introduced. (4) The operational program is next validated by running both the old and new versions against the same input data. The results are compared by a file-to-file comparator package and discrepancies are analyzed by the supplier's project team. (5) And finally, the validated program is run through an object code optimizer to reduce the overhead introduced by restructuring. The people at Catalyst say their restructured programs increase processing overhead by about 8% following the optimization step. The company then provides a tape of the restructured program and suggests ways to further enhance the program. (For more information on the structured retrofit concept, see a paper by Miller in Reference 2; for more information on Catalyst's retrofit service, see Reference 7.)

Programming for maintainability

There are several ways to develop programs with future maintenance in mind. One way is to use development tools which reduce the need for corrective maintenance. Use of data management systems, program generators, or application development systems, as described earlier, decrease the need for corrective maintenance because much of the code is automatically produced by such systems.

Another method is to use structured programming techniques to improve the quality of the software, thereby reducing corrective maintenance.

Higgins (Reference 5) discusses the question of converting existing 'unstructured' programs into well-structured ones. He says the idea of a 'structure analyzer,' which works on the structure of a bad program, only gives a picture of the mess that is there. Instead, he advocates taking the time to determine the data structures associated with the old program and creating a new design using those data structures.

Still another approach is to develop systems by prototyping. We will discuss this approach in detail next month. The point we want to make here is that with prototyping, the users are able to actually obtain useful outputs from the system early in its development life cycle, since the system functions in a true productive sense but in a prototype form. This generally—in fact, almost always—affects the users' perceptions of the system and prompts them to request changes—'enhancements' in maintenance parlance. These changes are made to the prototype through an iterative process, until the user is satisfied. With the prototyping tool (such as a data management system), these iterations are easily and quickly made. The end result is a good definition of what the production system must perform.

So prototyping appears to reduce the amount of enhancement maintenance requested by users for the production system, because so many of the changes are captured by the prototype.

Yet another approach—which improves the general maintainability of programs—is what Glass and Noiseux (in Reference 3) and others call 'defensive programming.' Like defensive

driving, in defensive programming programmers design and code with their senses tuned to potential problems. They write programs which have a better chance of detecting problems and pinpointing them. Glass and Noiseux describe five programming practices that enhance defensive programming—use of assertions, margins, commentary, audit trails, and flagging unsafe practices.

Assertions are statements that a programmer can make about the acceptable behavior of a program or data. These assertions must be testable. If one assertion fails a test, the program prints out a message describing the failure; it then either recovers from the failure or aborts. (These assertions are not the same as those used in proofs of correctness.) For example, an assertion might be used to detect erroneous or improbable input data, an improper logic flow, or an exception condition. The authors' point is that programmers should sprinkle assertion statements and tests throughout their programs so that they, and the eventual maintainers, can monitor a program's behavior.

Margins are reserved resources. Programmers should not use up all of the available computing resources in any of their programs, because then new features cannot be added without first freeing up some resources. Programmers should leave unused some portion of resources in each program—the margins. The authors note that on smaller machines, maintainers often spend one-half of their time shrinking the amount of resources used by a program, and the other half putting in the new features. This wastes a lot of time.

Commentary. Glass and Noiseux note that often the only reliable documentation that a maintainer receives is the program listing. Therefore, the listing should be made as complete as possible. They suggest placing comments in a program in at least five places: (1) before each subfunction, to explain its function, (2) before each interface to other modules or programs, (3) before each group of related declarations, (4) before each declaration, to explain its role and its possible values, and (5) before each difficult-to-understand portion of the program. These comments should not only explain what is happening but also why it is happening. Using program design language statements may serve this purpose.

Audit trails. Programmers should also provide statements in their programs that help maintainers track the action occurring in the program. During an audit run, these statements would record when a module has been entered, assertions made, when phases have been completed, and so on. This facility is most useful when it is optional, so that it need not be run for each production run.

Flagging unsafe practices. The authors note that programmers often do use unsafe programming techniques, and when used, these should be flagged. They list a number of such practices. One is the use of GOTOs to return to the program following a recoverable exception. These flags must be provided by the programmers; they are not automatically generated by most pre-processors or compilers.

These then are some suggestions we came across for creating software with maintainability in mind.

Off-loading maintenance work

In the future, we expect data processing departments to off-load not only some development but also some maintenance work. We see this happening in two ways: off-loading work to end users and off-loading work to package vendors by buying application packages. The major reasons for using these two approaches may not be to decrease maintenance, but that will be a side benefit. We briefly look at these possibilities.

Off-loading work onto end users. From a maintenance point of view, perhaps the most disturbing finding of the Lientz and Swanson survey was that applications that used a database management system or data dictionary grew faster than those which did not—indicating that such applications probably required more, not less, maintenance. With the growing use of database management systems, this could mean data processing departments will be even more swamped with maintenance work.

Yet, at the same time, DBMS-based tools are now available that allow end users to do some of their own programming. So once these tools,

and databases, are made available to users, data processing may be able to off-load some programming and maintenance work onto these other employees. In the May and June reports we gave several examples of companies where this is actually happening. We believe this is an important trend—end user programming.

From the cases of end user programming that we have seen, the responsibility for maintaining these applications has not been given to data processing; it has stayed with the end users. As users do more of their own programming, they will also do most of the maintenance required.

Off-loading work onto package vendors. The other way we believe companies will off-load software maintenance is by purchasing application packages. With this approach, corrective maintenance and some general enhancements will be performed by vendors. However, to take advantage of this type of package support, purchasers must be very careful not to invalidate their contracts. If a package is changed in any manner, the vendor may have legal grounds for discontinuing support of the package. One company told us that they go to great effort *not* to change a purchased package. If, for some reason, changes are needed, they contract with the vendor to help them perform analysis of the proposed changes and verify that these changes will not invalidate their support.

The purchase of packages will probably reduce corrective maintenance, and it may reduce some types of adaptive maintenance. For example, a main reason why companies purchase the ALLTAX package from Management Science America (MSA), to use in their payroll programs, is so that MSA will make the necessary adaptive changes in response to changing government tax regulations. The user companies need to maintain the other parts of their payroll programs, of course, but tax calculation maintenance is handled by MSA.

Boehm (in Reference 2) estimates that about 70% of the life cycle programming cost of an application comes from maintenance. With companies now spending about one-half of their programming time maintaining existing systems, it behooves data processing management to stress maintainability even more than in the past.

In this report we have suggested a few ways to help maintain older systems, to create more maintainable new systems, and off-load some maintenance work onto other people. It does not look as though program maintenance will go away, but there are ways to ease its burden on data processing departments.

REFERENCES

1. Lientz, Bennet P. and E. Burton Swanson, *Software maintenance management*, Addison-Wesley Publishing Company (Jacob Way, Reading, Massachusetts 01867), 1980; price $12.95.
2. Parikh, Girish (Ed.), *Techniques of program and system maintenance*, Ethnotech, Inc. (P. O. Box 6627, Lincoln, Nebraska 68506), 1980; price $25. (Parikh also publishes the *Software Maintenance News*; for information, write him at Shetal Enterprises, 1787 B West Touhy, Chicago, Illinois 60626.)
3. Glass, Robert L. and Ronald A. Noiseux, *Software maintenance guidebook*, Prentice-Hall, Inc. (Englewood Cliffs, New Jersey 07632), 1981; price $21.95.
4. Chapin, Ned, "Productivity in software maintenance," *Proceedings of the AFIPS 1981 National Computer Conference*, AFIPS Press (1815 North Lynn Street, Arlington, Virginia 22209), pp. 349-352; price $75.00.
5. Higgins, D.A., "Structured maintenance; New tools for old problems," *Computerworld* (375 Cochituate Road, Framingham, Mass. 01701), June 15, 1981, special section page 31; price $1.25.
6. For more information on the DRB Program Generator, contact David R. Black and Associates, 1780 Maple Ave., Northfield, Illinois 60093.
7. For more information on their structured retrofit service, contact The Catalyst Corporation, 9408 West 47th Street, Brookfield, Illinois 60525.

COMMENTARY

EVALUATING SOFTWARE MAINTAINABILITY

The need for software maintenance generally makes itself known by someone asking for a change—due to a program error, a change in the hardware or operating system configuration, a desired enhancement, or such. We have not heard of many cases of where organizations have made a study of their software, to rate it as to its maintainability.

We were pleasantly surprised, therefore, to come across a U.S. Air Force project that has developed a methodology for studying existing software and rating it as to its maintainability.

The method is described in *Software Maintainability Evaluator's Handbook*, prepared by the Computer/Support Systems Division at the U.S. Air Force Test and Evaluation Center, Kirtland Air Force Base, New Mexico, 87117. It is one of five handbooks developed by this division for use in operational testing and evaluation of software. These five handbooks were prepared for the positions of (1) software test manager, (2) software assessment team chairman, (3) software maintainability evaluator, (4) software operator-machine interface evaluator, and (5) computer support resource evaluator. As the position titles imply, the methodology has been developed for testing and evaluating large bodies of software. But it seems to us that a good many of the ideas can be applied in computer-using organizations of all sizes.

The goal of the software evaluator's handbook is to provide a procedure by which members of a team of evaluators can independently evaluate a body of software as to its maintainability. Each team member rates the source code of each piece of software (programs, modules, or whatever) on up to 89 characteristics. Further, the documentation associated with each piece of software is also rated, on up to 83 characteristics. Further, the statistical analysis of the ratings is performed on a computer. So the net evaluation of each piece of software, and its documentation, is the product of a team of evaluators, rather than the opinions of one or two individuals.

Once the software has been so evaluated, what then? Although not specified in the handbook, the answer seems evident. It is up to management to decide what to do about the problems. The poorest programs, from a maintainability standpoint, are generally well known within an installation. Programmers often shudder when asked to tackle these. When a number of programs have been evaluated, the 'worst offenders' probably will show up with the lowest ratings. But from the evaluation team's report, management will get an idea of just how bad these programs are, as compared with the others.

Management should then be in a better position to decide what to do about allocating systems and programming resources to clean up the offending programs. If prior maintenance costs can be associated with the various programs, it may become apparent where clean-up actions (such as discussed in this issue) should be taken.

But back to the software evaluator's handbook. As indicated, the method considers not only the source code for each piece of software but also that software's documentation. (It is noted in the handbook that the source code is often the only documentation available, but it is considered separate from the documentation in the methodology.) The documentation consists of the program design specifications, program testing information and procedures, and the program maintenance information. These documents are evaluated both as to their content and their format (for ease of use).

Both the source listings and the documentation are rated according to six characteristics: modularity, descriptiveness, consistency, simplicity, expandability, and instrumentation (aids which enhance testing). The ratings are performed by means of 'questions'. So the method consists of 12 sets of questions: the source listings evaluated on the six characteristics and the documentation likewise.

Each 'question' really is a statement about a desirable feature of a source listing or a document—a feature which enhances or supports maintenance. The evaluator is supposed to rate the source listing, or the document, for that feature by using a six-value rating scale. An 'A' rating means the source listing, or document, exhibits the feature in the best possible way. 'B' means that the feature is used very well; 'C' means acceptable, 'D' means acceptable but some improvements should be made, 'E' means unacceptable with major improvements required, and 'F' means the software (or documentation) must be completely redesigned.

The method recognizes that not all questions apply to all pieces of software. An example is given in the handbook: a particular module may not have any statement labels. So a question such as "Statement labels have been named in a manner which facilitates locating a label in the source listing" may appear irrelevant for that module. But, in this instance (the handbook continues), the software is actually more understandable because there is no branching to labelled statements. So the evaluator might well assign a very high ('A') rating on this question, just as though labels were used that had been very well named.

If you are interested in studying the subject of evaluating software maintainability, this handbook probably would be very useful. You can obtain a copy by writing to the Computer/Support Systems Division at the address given above. As yet, there is no charge for the handbook. If demand is appreciable, we would guess that the Air Force would have it distributed through the National Technical Information Service.

SUBJECTS COVERED BY EDP ANALYZER IN PRIOR YEARS

1978 (Volume 16)

Number	*Coverage*
1. Installing a Data Dictionary	G
2. Progess in Software Engineering: Part 1	H
3. Progress in Software Engineering: Part 2	H
4. The Debate on Trans-border Data Flows	L
5. Planning for DBMS Conversions	G
6. "Personal" Computers in Business	B
7. Planning to Use Public Packet Networks	F
8. The Challenges of Distributed Systems	E,B
9. The Automated Office: Part 1	A
10. The Automated Office: Part 2	A,D
11. Get Ready for Major Changes	K
12. Data Encryption: Is It for You?	L

1979 (Volume 17)

Number	*Coverage*
1. The Analysis of User Needs	H
2. The Production of Better Software	H
3. Program Design Techniques	H
4. How to Prepare for the Coming Changes	K
5. Computer Support for Managers	C,A,D
6. What Information Do Managers Need?	C
7. The Security of Managers' Information	C,A,L
8. Tools for Building an EIS	C
9. How to Use Advanced Technology	K,B,D
10. Programming Work-Stations	H,B
11. Stand-alone Programming Work-Stations	H,B
12. Progress Toward System Integrity	L,H

1980 (Volume 18)

Number	*Coverage*
1. Managing the Computer Workload	I
2. How Companies are Preparing for Change	K
3. Introducing Advanced Technology	K
4. Risk Assessment for Distributed Systems	L,E,A
5. An Update on Corporate EFT	M
6. In Your Future: Local Computer Networks	F,B
7. Quantitative Methods for Capacity Planning	I
8. Finding Qualified EDP Personnel	J
9. Various Paths to Electronic Mail	D
10. Tools for Building Distributed Systems	E,B,F
11. Educating Executives on New Technology	K
12. Get Ready for Managerial Work-Stations	A,B

1981 (Volume 19)

Number	*Coverage*
1. The Coming Impact of New Technology	K
2. Energy Management Systems	M
3. DBMS for Mini-Computers	G,B
4. The Challenge of "Increased Productivity"	J,K
5. "Programming" by End Users	H,G,B,C
6. Supporting End User Programming	H,G,B,K
7. A New View of Data Dictionaries	G,B,
8. Easing the Software Maintenance Burden	H,B,G,

Coverage code:

A Office automation	E Distributed systems	I Computer operations
B Using minis & micros	F Data communications	J Personnel
C Managerial uses of computers	G Data management and database	K Introducing new technology
D Computer message systems	H Analysis, design, programming	L Security, privacy, integrity
		M New application areas

(List of subjects prior to 1978 sent upon request)

Prices: For a one-year subscription, the U.S. price is $60. For Canada and Mexico, the price is $60 *in U.S. dollars,* for surface delivery, and $67 for air mail delivery. For all other countries, the price is $72, including AIR MAIL delivery.

Back issue prices: $7 per copy for the U.S., Canada, and Mexico; $8 per copy for all other countries. Back issues are sent via AIR MAIL. Because of the continuing demand, most back issues are available.

Reduced prices are in effect for multiple copy subscriptions, multiple year subscriptions, and for larger quantities of a back issue. Write for details.

Please include payment with order. For payments from outside the U.S., in order to obtain the above prices, *use only an international money order or pay in U.S. dollars drawn on a bank in the U.S.* For checks drawn on banks outside of the U.S., please use the current rate of exchange and add $5 for bank charges.

Editorial: Richard G. Canning, Editor and Publisher; Barbara McNurlin, Associate Editor. While the contents of this report are based on the best information available to us, we cannot guarantee them.

Missing Issues: Please report the non-receipt of an issue within one month of normal receiving date; missing issues requested after this time will be supplied at the regular back-issue price.

Address: Canning Publications, Inc., 925 Anza Avenue, Vista, California 92083. Phone: (714) 724-3233, 724-5900.

Microfilm: EDP Analyzer is available in microform, from University Microfilms International, Dept. P.R., (1) 300 North Zeeb Road, Ann Arbor, Mich. 48106, or (2) 30-32 Mortimer Street, London WIN 7RA, U.K.