Maintenance as a function of design

by JAMES R. McKEE

*International Monetary Fund*

Washington, D.C.

ABSTRACT

Changing one's point of view on the maintenance function can lead to a better understanding of the relationship between maintenance and other aspects of software products. This can lead to an improved allocation of effort when building software products.
INTRODUCTION

The maintenance requirements of software products are generally given insufficient consideration by software product designers because they miscalculate the importance of the maintenance function as a cost component in the life of a software product. One aspect of the problem may be attributable to an inappropriate point of view. The life cycle model most commonly used to portray software development misrepresents the activity it is intended to explain and gives insufficient emphasis to maintenance.

Corrections to these problems may lead to more optimal solutions in the process of software development. This is likely because the trade-off between maintainability and other components of a software product will become more properly balanced. Correspondingly, the analysis and design documents associated with software products will include items of greater value to the maintenance function.

POINTS OF VIEW

When practitioners first started trying to bring some order to the process of software development, they developed the concept of a “life cycle” for new software. The cycle generally began with problem recognition or goals. It then stepped through analysis, design, coding, installation, testing, and operation. The last step of the cycle was maintenance. The problems with this model are numerous. As Zvegintzov has pointed out, this model does not accurately describe a system’s life. Moreover, the model is generally portrayed as a linear concept, not as a cycle.1 In reality the life cycle model mixes a linear concept with a cyclical concept. It ties the concept of the process by which good operational product is generated to the operation of a system that uses the product. Perhaps the most egregious error in the traditional life cycle model is the mishandling of the concept of maintenance. Maintenance is generally shown as a single step at the end of the cycle; in fact, it is better portrayed as second- (or 3rd-, 4th-, . . . , nth-) round development. The life cycle then becomes develop, operate, develop, operate, develop, and so forth. The model now looks more like a cycle, but has become less useful. This is because the relationship between product building and operations is not so tightly coupled. Much as an airframe manufacturer typically does not operate an airline (and vice versa), the operations of most software products are separated from their manufacture. As an aside, one can make the argument that the failure to isolate software development from operations is a fundamental error that results in a product of extremely poor quality.

What we have left when we dispense with the life cycle model is the mishandling of the concept of maintenance. There is one other effect of the wide acceptance of the life cycle model with which we must deal. When maintenance (dealing with old products) is included at the end of the cycle, then it is presumed that the beginning sections of the cycle are to be applied to new products. This leads not only to a rather wrong-headed view of how the efforts of the analyst-programmer are distributed, but also fosters the impression that structured techniques are best applied only to new projects. As shown in Figure 1, if we are to divide analyst-programmer activity between existing and new applications, at least two thirds of the activity will be attributable to existing applications.2,3

Although the analysis to prove the point has not been developed here, it is perfectly clear that the application of structured techniques is equally valid for all analyst-programmer activity. It then follows that the greatest absolute benefit will occur when the analyst-programmer is engaged in maintenance. While this conclusion has been recognized, the process by which we obtained it here has not.

COSTS AND ALLOCATION OF EFFORT

In software development, the validity of a project should be determined by traditional cost–benefit analysis.4 This approach uses a model in which costs are seen to be rising and benefits falling as the scope of a project expands. The disuc-
sion here will be limited to the cost side of the model with the operating assumption that minimization of the total cost of a software product over its entire useful life is a reasonable objective function for the software engineer. This assumption is held to be valid whether the product is an addition, correction, or modification to an already existing product, or a completely new product.

For our discussion the total cost to be minimized consists of three fundamental components: maintenance cost, operating cost, and original development cost. This schema includes all costs of fixing problems or errors, all enhancements, and all changes required by alterations in the operating environment of a product—that is, the costs of any and all changes to a product after it is first delivered—within the definition of maintenance. Operating costs include hardware costs, consumables, and any labor and management costs associated directly with the running of the product. Development costs include all the original analysis, design, coding, and testing costs of a new product. The behavior of these cost components is of considerable interest to the software engineer, as they should be a major determinant of the structure of his product.

The historical trends of these cost components are worthy of review. Operations costs per unit of work are declining largely because the hardware component of these costs is rapidly declining—this overwhelms other operations cost components. However, as the cost of a unit of work has declined, the demand for additional units has expanded in greater proportion. Thus, the overall trend of this expenditure is up, not down. (This behavior can be explained by a concept well known to economists, that of elastic demand. The demand for computer hardware has been highly price elastic throughout the history of the industry and is expected to remain so for the foreseeable future.) Development costs and maintenance costs are both labor intensive and thus are increasing. Maintenance costs may also be increasing because the useful life of software products is increasing. Certainly, our realization of the enormity of maintenance costs is increasing.

The distribution of costs between these major components is likely to vary widely depending on the nature of the work, the maturity of the system, and the work style of the organization. Figure 2 shows the implied distribution between maintenance activity, hardware operations activity, and all other activity within fifteen federal installations surveyed by the General Accounting Office (GAO). The other category includes personnel costs attributable to operations, administrative support, and management, as well as new-product development. The figure is interesting because it demonstrates the great importance of the maintenance function as well as the continuing importance of hardware cost.

The point of this aspect of our discussion is that while hardware costs have traditionally been given, and should continue to be given, great attention, the next most important cost component is software maintenance. Original development costs, which receive tremendous attention in the structured-analysis literature, are a distant third in the actual cost of most systems.

TRADE-OFFS

In all development projects there are many trade-offs. For our purposes, the trade-off between maintenance and other cost components is of interest.

The strong relationship between a well-structured development process and the maintainability of a system is well recognized in the software-engineering literature. In almost every treatise on structured analysis or structured design, long arguments are made about the efficacy of these structured techniques. The arguments always include testimony to the fact that structured development produces systems that have fewer errors, are much easier to understand, and thus much easier to maintain. However, they tend to view maintainability as a fallout of good structured techniques. A better point of view would be to view maintainability as a quantifiable characteristic of software. Maintainability could then be included more usefully in the objective function for a product, and more or less of this quality could be included in the delivered product as a result of design decisions.

Using this view, one can trade additional product development effort for reduced maintenance costs. The technical optimum is when the last added-development costs are just covered by the reduced-maintenance costs, the assumption being that any further development efforts generate insufficient benefits. On a practical basis very few people have hard numbers to cover this issue. Nevertheless, it is probably safe to assert that in most cases the trade-off between development and maintenance costs can be pushed much further in terms of increased development costs. It is also most likely to be the case that this development effort should be pushed beyond the amount of maintainability that falls out of good structured techniques. This additional maintainability is designed in the product.
The same optimality presumptions apply with respect to the trade-off between maintenance and operations costs. However, one should take great care in making any assumptions about operations costs. In all probability the sum of all operations costs for a product over its useful life is not declining. Nevertheless, operations costs have always been given considerable attention, while maintenance costs have not. Thus, on this latter basis alone one could presume that some trade-off in favor of increased operations costs and lowered maintenance costs would be reasonable.

PLANNING FOR MAINTENANCE

As Reutter points out (see Figure 3), most of the activity in maintenance is directed toward product capabilities or characteristics not included in the original product design. Moreover, most of the remaining maintenance activity is directed toward changes in the environment in which the software product operates. Only a small portion of maintenance is directed toward correction of errors. While this may not reflect the experience with all software, it probably does represent what one should expect from fairly well-designed and well-written software products. In high-quality software the error rate may approach zero; this should be an attainable objective. On the other hand, we expect the environment to be changing. We also expect demands for enhancement. Moreover, we expect both of these to occur on a regular basis. What needs to be done is to develop software that is very amenable to these expected changes.

Many areas of expectation for change are identified at the analysis and design stages of product development. In these stages decisions are made that determine the scope of the project. Characteristics to be included in the product are then given the detailed attention necessary to complete the development process and characteristics to be excluded are frequently forgotten. While it is true that many specification documents have a brief statement about avenues of possible extension for the product—and a few even have sentences scattered throughout about points of expandability—these statements are usually treated as asides to the process of building the specified product.

There is another side to the coin of features not included in a product design. This has to do with features or technical solutions that were rejected as being in some way unsuitable for the product. These include all those dead ends encountered during the analysis and design stages. Also to be considered are those features that once looked so promising, only to be found fundamentally inconsistent with the accepted development of the product. The information and knowledge associated with these considered but rejected features are almost never found in any specification document.

A major set of additions to the specification document is necessary to capture the analysis of features excluded from a product. These additions may be of some value to the builders of the currently specified product, but their objective is specifically to aid the maintenance analyst–programmer. In a sense, these additions will be a resource library that the maintenance programmer can explore to see if his problem has already been addressed. It will also serve another important purpose. It will stand as the justification for the design decisions in the current product that are related to potential extensions of the product. Finally, these additions will be spread throughout the specification and design documents. They will serve as a continuing reminder to all those involved in the development process to include maintenance-related issues in every decision process.

Case Study—The Economic Information System

The Economic Information System (EIS) is a large (15 gigabyte) database system for the time series data describing the economies of all countries in the world. The system is currently under development at the International Monetary Fund and is scheduled to begin operation in June 1985. The EIS serves well to illustrate some of the points that have been made in this paper. It is a moderately large software project (budget in excess of $3.5 million) that in some aspects is a conversion of a current system and in other aspects a major extension of that system. Thus, it is typical of most of the software projects found in the commercial world. Both components of the project fall within the realm of maintenance.

The current database system consists of a set of ISAM files and home-grown database programs resident on a Burroughs mainframe. In addition, a large set of operations programs have been developed to generate a number of major publications that are run from the database. Most of the code for both the database and the operations are in COBOL. All of the operations code and a subset of the original database code (152,000 lines) will be converted directly to the IBM environment. This will be the batch production part of the new sys-
tem. An on-line access and update system is also being constructed as an addition to the previous system.

The original charge to the development team was to move the current system to an IBM environment with the on-line extensions, use a commercially available database management system (DBMS), and be up in 18 months. In the initial justification for the project it was stated that “productivity aids would become available in the form of programming tools and software packages which will significantly reduce staff resources required for future systems development and ongoing systems maintenance.” Thus, the continuing cost of maintaining systems was given primary focus prior to project initiation.

The first major decision in this project was the choice of DBMS. The question was formed around the type of DBMS (hierarchical, network, inverted file, and relational) as much as the particular vendor. Hierarchical- and relational-type DBMSs were dropped early in the decision process, the former because of its inflexibility to change and large up-front design requirements, and the latter because of known performance problems and the absence of any product with performance experience in large database applications. In the evaluation of the remaining two types of DBMSs, three critical areas—DBMS data structures, database implementation and maintenance, and user access and manipulation capabilities—were identified. Critical requirements were developed within each of these areas. Candidate systems were then evaluated against these requirements.

This DBMS choice provides an excellent example of trade-off. Because of the mix between batch and on-line activity in this application, neither the network- nor the inverted-file-type of DBMS was found to have an advantage with respect to hardware resources. However, with respect to implementation and maintenance, the inverted-file-type DBMS had an overwhelming advantage. The database design process is much simpler in an inverted-file database. Moreover, inverted-file structures are much more amenable to extension and change than network structures. This became the basis of our choice.

Another example of the maintenance concept entering into a major decision in this project arose in the database design process. In the batch operations process on the current system large data records (10 Kbyte) are read into a buffer. The applications then use a central utility to obtain the sections of the records that they need. This works well in the current batch system; however, the approach is completely inappropriate for on-line update and inquiry activities. The on-line requirements of the project have led to the development of much smaller records in the target database. The question is then whether to build up the large buffer the entire batch stream expects, or to make some major changes in the data-gathering procedures of the batch application code. From the design and development effort point of view, building the buffer would be the best choice. From an operations point of view, building the buffer would be more expensive. However, overnight batch costs are 10% of daytime costs in our environment and there is a succession of use of various parts of the large buffer in our current operations. Thus, the operations costs are not an overriding issue. What is clear is that the large buffer structure is not likely to be suitable for the extensions of this application that will be forthcoming after it is put in place. Moreover, the structure that is chosen now will be cast, if not in steel, at least in bronze for some years to come.

It was decided to change the data presentation procedures. This decision will raise development costs for the project. The decision will also have a negative effect on our ability to produce a product on a timely basis. However, the ability to enhance the product after its initial delivery will be significantly increased.

CONCLUSION

There is still substantial room for improvement in our understanding of the process by which software products are constructed. A more carefully constructed life cycle model will improve this understanding. In addition, a clear analysis of the cost trade-off between maintenance and other cost components of a software product is likely to lead to a better resource allocation. However, these suggestions are limited to creating the setting in which improved maintainability may be developed. The many techniques that may be employed for improving maintainability have not been explored. This remains the task of future explorers in this field of endeavor. The growing cost of software maintenance suggests such efforts be given high priority.

ACKNOWLEDGMENTS

I would like to thank my colleagues, Soon Choi, Thomas L. Williams, Kathleen X. Nelick, and S. Stuart Morrison, and my wife, Mary Jane McKee, for the many suggestions and improvements they have provided in the production of this paper, and the Graphics Section of the IMF, for providing the charts. The errors and omissions remain my own.

The ideas and opinions expressed herein are solely those of the author and are not necessarily representative of, or endorsed by, the International Monetary Fund.

REFERENCES


SUGGESTED READINGS
