International Symposium and Workshop on Systems Engineering of Computer Based Systems, March 1995, Tucson

A Case Study in Assessing the Maintainability of Large, Software-Intensive Systems

Alan W. Brown, David J. Carney, and Paul C. Clements

Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract: *Maintenance of a computer-based system accounts for a large proportion of the total system cost. However, no well-established techniques exist for assessing the maintainability of such a system. This paper presents a case study in assessing the maintainability of a large, software intensive system. The techniques we used are described, and their strengths and weaknesses discussed.*

1 Introduction

Maintenance of computer-based systems can be extremely expensive, rivaling or even dwarfing the cost of original development. Years of empirical data have established that in large, long-lived, software-intensive systems, as much as 80% of the overall life-cycle cost can accrue *after* initial deployment in the field [1]. Hewlett-Packard recently reported that they have 40-50 million lines of code in maintenance, and that 60%-80% of their research and development personnel are involved in maintenance activities [2].

Given the proportion of costs devoted to maintenance, it is clear that an assessment of the maintainability of a computer-based system should play an important role in any system evaluation, especially during early development. However, while the costs of maintaining existing systems are well-documented, it is much more difficult to find techniques that allow maintenance costs of a system in construction to be predicted.

As part of a larger audit of a computer-based system the authors were recently called upon to assess the maintainability of the software being written. It is a large, real-time, embedded process control system, comprising on the order of a million lines of Ada code. The system was designed to be distributed, flexibly configured, extremely reliable, and to meet strict performance and human interface constraints.

This paper presents the approach used to make this assessment. The remainder of this paper is as follows. Section two discusses the conceptual basis for our assessment approach. Sections three and four divide that into its two key aspects. Section five provides a summary and conclusions.

2 The Basis for Assessing Maintainability

As with any evaluative process, assessing maintainability needs to rest on a foundation that provides a reasonable basis for making any conclusions or predictions about a system's probable maintenance costs. This basis need not be exhaustive or complete — we do not believe that it is as yet possible for a truly reliable quantitative predictor for maintainability — but must yield a result that offers some qualitative conclusions based on objective evidence. We also stress that the assessment method itself be capable of repeated use.

We posit that assessing maintainability has two aspects. The first pertains to the nature of the system itself: How well does the software lend itself to change? How easy is it to understand when considered from various perspectives, i.e., from the architectural, low-level design, and coding points of view? With what level of fidelity does the implementation reflect the design chosen for it? How widespread are the effects of a particular modification? What is the quality of its documentation?

However, a single-minded product-oriented approach ignores the capabilities of the maintenance organization. Therefore, the second key aspect in our approach for assessing maintainability addresses the adequacy of the maintenance infrastructure: what processes are in place? How appropriate are the tools and technology employed by the organization? How familiar are the personnel with the system being maintained?

Our approach to assessing maintainability includes both of these aspects. We now consider how we incorporated both into the assessment of the actual system.

3 Maintainability as a Function of the System

Maintainability metrics typically concentrate on product complexity, reasoning that errors are more likely to occur in complex systems or complex parts of a system, and that those will be harder to understand and change (e.g., [4], [10], [8]). Complexity is usually measured by analyzing source code.

In defining our approach, we considered existing maintenance metrics. While not without benefit, traditional scalar maintenance metrics cannot serve as the basis for a comprehensive assessment. First of all, they embody a hidden set of assumptions about the nature and frequency of maintenance operations. Maintainability metrics should predict maintenance cost, and maintenance cost is a function of an *actual* maintenance (change) history, not a hypothetical one. Changes cannot be predicted by analyzing the system in isolation from its domain, development, and marketplace contexts, which is what program-analysis-based maintainability metrics do. Suppose a shared-data system is designed to accommodate new computing platforms, but not designed to accommodate a change in the format of the shared data. Is it maintainable? The right answer, we believe, is “Yes, with respect to new computing platforms, but no with respect to changing the data format.” If the development organization expects to adopt a new data layout, but does not expect to migrate to a new computer, then the answer may be simplified to “No.” No scalar metric can capture this multi-dimensional aspect of maintainability.

Second, complexity may actually *improve* maintainability. In the process control case we studied, the program initialized by reading a large data base of site-specific configuration data and version-specific rules for display generation. This data affected the program’s timing behavior, the content and layout of operator displays, and hundreds of parameters concerning its complicated user interface. The use of this so-called “adaptation data” made the program decidedly more complex, but it could be tailored to produce entirely new output displays with virtually no change to the run-time code at all. It was highly maintainable *with respect to the changes facilitated by the adaptation data*, scalar complexity-based metrics to the contrary notwithstanding.

Our assessment procedure took into account the domain-specific, project-specific, and organization-specific aspects of maintainability. It proceeded as follows:

1. We enumerated the quality attributes that were considered important to achieve and maintain. In our example, these included ultra-high availability (accomplished by a sophisticated distributed, fault-tolerant design and implementation scheme), performance, and the ability to extract a functionally useful subset from the system.

2. We enumerated a set of change classes that were deemed likely to occur to the system over its lifetime. These change classes may come from anticipated requirements for the system, or domain knowledge about changes made to legacy systems of the same genre. In our case, likely changes included upgrades in network and processor hardware and the operating system, importation of third-party application or support software, additional functional or performance requirements, and extraction of functionally useful subsets for staged deployment.

We were careful to consider a change class that affected each of the quality attributes listed above (e.g., increasing the system’s availability requirement). Also, we were sure to consider change classes that affected the system at the architectural level (i.e., affect its highest-level components), the module level, and the code level.[1]

3. For each change class, we defined a specific instance of the change as a *change scenario*. For instance, to test the system’s ability to accommodate increased performance, we posited a 50% increase in the maximum number of vehicle radar tracks the system was required to monitor.

1. For some systems, there may be no distinction between highest-level components and modules, or between modules and code units.

4. For each change scenario, we conducted a *change exercise*, in which the developers were asked to accommodate the change by showing us all components (from architecture-level components, to design-level modules, to Ada packages) and documentation that would be affected by the change. The result was a set of *active design reviews* [9] in which the participants were pro-active, each in his or her own area.

The purpose of the change scenarios is to assess the system design against likely, rather than arbitrary changes. During each exercise, we investigated the process to implement each change, and viewed and catalogued the code and documentation that was or would have been produced, accessed, or modified as a result of the change. During some of the exercises, we actually made code changes; for others, the developer had anticipated us by preparing working prototypes with the change installed. Table lists some of our change exercises, and the quality or aspect that each one tested.

In our case, the system was implemented; we could actually compile lines-of-code statistics for each change exercise. However, the exercise could be performed on a system for which design (but no code) existed. If the design documentation was not complete enough or detailed enough or informative enough to identify specific areas of change, then the exercise would serve to uncover those documentation deficiencies. In this way they could be corrected, rather than having implementation being allowed to proceed based on incomplete designs.

The result of the change exercises was a set of high-confidence metrics, one per class of change, with which project management could project the cost of performing concrete maintenance operations to the system.

Finally, since all changes cannot be anticipated, we assessed whether or not generally-accepted software engineering standards had been followed which, in the past, have resulted in systems that were easily modified with respect to normal life-cycle evolutionary pressures. This step included the use of standard code quality metrics, as well as traditional documentation inspection and quality assessments. We also inquired after the design rationale to see what information was encapsulated at various design levels. This encapsulation implies classes of changes that the designers had in mind, implicitly or explicitly, against which the system is insulated.

4 Maintainability as a Function of the Environment

In addition to the inherent maintainability of the system itself, the quality and consistency of that maintenance environment will have a direct impact on the maintainability of the system being supported. We define “maintenance environment” as the set of tools, techniques, and processes that are applied to a system during its maintenance.[2] The successful long-term maintenance of any system requires an appropriate maintenance environment (i.e., an appropriate set of tools. techniques, and processes).

Some existing approaches to maintainability metrics have also considered questions of infrastructure; approaches like those of Pickard and Carter [6] “must be embedded in a maintainability measurement framework that is adapted and calibrated to a specific software development environment.” Boehm’s venerable COCOMO [1], which takes maintenance organization factors into account, has a similar viewpoint. Our approach agrees with these two.

Existing approaches to evaluating software maintenance environments (as well as development environments) fall into one of two categories. On one hand, a technology-oriented view of software maintenance concentrates on the selection of individual computer-aided software engineering (CASE) tools that may aid in supporting particular maintenance activities. The IEEE recommended practice on the selection and evaluation of CASE tools [3] is typical of guidelines that can be used to help with tool selection. On the other hand, a more process-oriented view of software maintenance concentrates on assessing the practices used during maintenance. This view can be seen in the Software Engineering Institute’s (SEI’s) software capability maturity model [5] through which an organization (not the software system) is assessed. The assessment is based on the extent to which certain key practices (configuration management, project tracking and oversight, software quality assurance, etc.) are defined, repeated, measured, and optimized. A similar approach is that of the International Standards Organization (ISO) 9000 standard for certification of an organization’s software practices [7]. These process-centered views provide a gener-

2. It is also true that the maintenance environment is itself a large software-intensive system that must be managed and maintained. This “recursive” property is common to systems that produce other systems.

al, overall impression of an organization's ability to define and enact processes, which in turn has implications about the success of that organization's maintenance activities.

We incorporate both of these views in our approach, though with some important differences. First, we do not consider either that tools and techniques or that processes should be considered in isolation: our notion of an "environment" is the combination of all three, each providing context for the other two. From this, it follows that an assessment of an environment must also take this view: an assessment must consider tools, techniques, and processes as a whole, and not as separable factors to be evaluated. Tool selection can only be understood and evaluated in the context of the techniques and process being supported, and the appropriateness, repeatability, and cost-effectiveness of the process being employed during maintenance can only be considered with a knowledge of the available automation for those processes.

We have also found it essential to concentrate attention on how the maintenance environment specifically applies to the current system being maintained; in particular, this attention includes examining the goals of the organization that developed, is maintaining, and is using the system in question.

These considerations lead us to suggest that the wide range of activities carried out in assessing a maintenance environment fall into the following categories:

- Comparing the development and maintenance environments;
- Evaluating the plan for transition of responsibility;
- Assessing the key maintenance practices;
- Examining the organization's maintenance of other systems.

We discuss each of these in detail below.

4.1 Comparing the Development and Maintenance Environments

To examine the maintenance environment for a system, a necessary place to start is the development environment that produced that system. The development environment leaves a legacy of documents, data, and knowledge concerning the system that must be brought forward into maintenance. The form, format, and accessibility of these artifacts are strongly affected by the environment through which they came into being; to the extent that the maintenance environment is similar or different, the use of those artifacts will either be facilitated, constrained, or even impossible. In our assessment, therefore, we compared the two environments by focussing on four key questions:

- When in the system's development cycle is the maintenance environment ready for use?
- Is the tool makeup consistent between the development and maintenance environments?
- Aside from consistency, what is the intrinsic quality of the tools?
- Is the maintenance environment documentation complete and consistent?

Creation of the Maintenance Environment

The maintenance environment must be in place long before the system is brought into maintenance. An environment is itself a system that needs to be debugged, and that has a breaking-in period. Maintainers are typically different persons from developers, and gaining tool expertise is commonly a time-consuming process. Finally, without a precise specification of the maintenance environment, there is no way to assess the consistency of the tools between the two environments. In the system under examination, the maintenance environment was largely in place; as far as could be determined, the maintenance personnel were gaining experience using the tools.

Consistency of Tools

Any inconsistencies between the development and maintenance environments represent potential problems, and in making an assessment, such inconsistencies should be analyzed, their reasons recorded, and their impact noted. It should also be noted that inconsistencies may have a positive as well as a negative aspect, and both must be examined.

We saw two examples of this in the system we examined. First, the code development tool suite was proprietary to the development organization, and a different set of coding tools was planned for maintenance. For example, the likely impact of changing Ada compilers and other essential development tools had a huge element of risk from a maintenance point of view.

Second, many thousands of pages of documentation were recorded in the proprietary form of an archaic documentation system. The maintenance organization, however, had invested heavily in the use of Interleaf as the means for storing and browsing all on-line documentation. While this divergence of documentation technology needed to be resolved, both the development and maintenance organizations were under internal pressures not to change.

Quality of Maintenance Tools

The questions of tool quality and of tool consistency are different, yet almost impossible to separate. As the previous section mentioned, some tool choices might be made on a basis of consistency, be driven by factors such as cost-effectiveness, yet result in the maintenance environment containing tools of very poor quality.

It is clear that regardless of the consistency question, the inherent quality of the tools to be used will have a significant impact on maintainability. One typical example is the complexity and comprehensiveness of the configuration management system used. This can be a major constraining factor throughout the maintenance process.

Consistency of Documentation

In addition to examining documentation on the maintenance environment itself, and seeing demonstrations of the tools and techniques that were available, it is important to talk with some of the development engineers in person in order to evaluate the fidelity of the available documentation. The development engineer is perhaps the most obvious, but often ignored repository of information concerning development practices that ought to transition to maintenance.

For example, it is through discussions with development engineers that key optimizations in development practices can be identified. For instance, through such discussions it may be found that many home-grown scripts and filters are used during debugging and testing.

4.2 Evaluating the Plan for Transition of Responsibility

For many large, software-intensive systems, the development and maintenance organizations are entirely separate. This may be due to the fact that different organizations have been contracted for development and maintenance aspects of the system, or that a single organization is internally structured with separate development and maintenance divisions. In either case, it is inevitable that much valuable information about the system will be lost in transitioning it from development to maintenance.

To aid transition from development to maintenance, a number of key documents need to be in place, up-to-date, and of high quality. In our experiences, we have found that the following documents are essential:

- A high-level overview of the architecture of the system. This should establish the major design requirements for the system, the choices made in implementing the system to meet those requirements, the basic functional components of the system, and the typical operation of the system. Again, while the need for such a document may seem obvious, we have found that members of the development organization have a shared understanding of the basic design of the system which has been built up over the course of development (often a number of years). They find such a document unnecessary for their own needs, and forget the needs of the maintainers.
- A detailed, agreed transition plan for moving the system to maintenance. This plan must define in detail the tools, techniques, and practices to be used in maintenance, the responsibilities and expectations of each of the participants, operating procedures for finding and eliminating system errors, and so on. In the system we examined we found that there were three conflicting versions of this plan — one produced by the maintenance organization, one produced by the development organization, and one produced by the project monitoring organization.

4.3 Assessing the Key Maintenance Processes

There are a number of key processes that take place during maintenance; these practices parallel the key practices that occur during development. It is our experience that while most software developments consider these as critical aspects of the development phases, they are often severely neglected when establishing the maintenance environment. The quality, efficiency, and effectiveness of these processes is essential. The maturity of these processes *in the context of the maintenance environment* must be examined in detail to ensure that they are well-defined. Automation of these processes can ensure that the data is accurate, quickly accessible, and easily manipulated.

The key process areas include:

- Code inspection processes: These include techniques for fixes and enhancements to the system. Such inspections are just as critical during maintenance as they were during development. These processes must be applied consistently and effectively to ensure the integrity of the system. In fact, inspection processes may need to be more rigorous during maintenance than development.[3]

- Test procedures: Just as in development, adequate testing using test scripts and acceptance test suites is an essential component of maintenance. In most cases these should mirror those used during development of the system

- Build and release practices: Developing new system releases, fielding those releases, and moving users from one release to the next must be carefully defined. A number of choices concerning system releases must be made, and rationale for those choices documented and periodically examined. Examples of choices include frequency of new releases, whether some or all of the sites are upgraded at the same time, handling of emergency fixes for critical errors, and customizations at different sites.

- Change request procedures: Handling the influx of change requests based on unanticipated behavior of the system, or suggested enhancements to the system is an important task. For large systems there are likely to be many thousands of such requests. Each must be recorded, considered, and appropriate action taken. The change management process can be detailed and complex, particularly for systems with high availability, performance, and safety requirements. In assessing such processes aspects to consider include the prioritization scheme in use, allocation of changes to releases, and reporting and statistics gathered on change requests and fixes.

3. As any maintainer of a large system will attest, software *does* age and decay — without extreme care each “enhancement” to the system is likely to make the system more complex and less easy to maintain.

4.4 Examining Maintenance of Other Systems

Most organizations employing large, software-intensive systems are in the position of maintaining a number of such systems. Each such system has been developed, installed, and maintained over a long period of time. Hence, the maintenance of one such system cannot be considered in isolation. Many decisions must take a wider picture of systems maintenance of a range of such systems. We can consider a number of issues in this regard.

First, the system may interface with a number of existing or planned future systems. Decisions concerning system interfaces, performance, etc. may be fixed, and may provide substantial design challenges during maintenance. For example, the system we examined interfaced to a wide range of systems (e.g., radar systems, weather systems, collision warning systems) constructed over a 25 year period. The continued correct operation of these existing systems was paramount in any proposed enhancements.

Second, maintenance engineers have an existing base of tools, techniques, and practices for maintaining large, software-intensive systems. The maintenance environment for a new system must harmonize with this existing environment. This provides a substantial opportunity to build on established successful aspects of the existing environment. It also provides a major challenge in introducing new approaches that differ significantly from tried and trusted methods. An obvious example seen in our system is that the system was written in the Ada programming language, while the existing systems being maintained were in assembler, FORTRAN, JOVIAL, and many other languages (but not Ada!).

Third, the recent climate of systems development is toward the use of commercial off-the-shelf (COTS) components to provide large parts of any large software-intensive system. This provides significant maintenance challenges: large parts of the system are maintained by COTS vendors, the ability to change COTS software is often severely limited, new releases of COTS occur when the vendor decides not when you need them, and tracing errors can be problematic in systems that include COTS components. Many organizations are struggling to come to terms with maintenance of such systems. We know of a number of examples of major systems that include old, un-maintained versions of COTS components where the COTS vendor no longer is in business. Contingency plans for such situations must be made.

Fourth, it must be remembered that it is not just the operational software that must be maintained. Maintenance is required for all of the support software necessary for the development, testing, maintenance, and release of the operational software. In comparison with the operational software, in most large, software-intensive systems the support software is larger in size, more varied in the languages and styles of development (e.g., database systems for data entry and manipulation, assembler code for network support, high-level scripting languages for test scripts). This is in addition to maintaining large amounts of documentation, data for testing, and administrative information on system configurations. Maintenance for each of these is necessary and must be considered.

5 Summary and Conclusions

In this paper we have emphasized the importance of assessing the maintainability of large, software-intensive system, and the need to do this by using a set of techniques that provide a range of perspectives on the maintainability of a system. Two aspects of assessing maintainability were discussed: maintainability as a function of the system itself, and maintainability as a function of the environment supporting the system.

To assess maintainability as a function of the system, the techniques available include:

- Scalar techniques which provide a quantitative assessment of certain aspects of the system (e.g., cyclomatic complexity).
- Scenario-based techniques which provide more specific assessments of quality attributes of the system (e.g., availability, performance).

In our discussions we emphasized the second of these approaches, with particular examples illustrating how appropriate change scenarios could be selected for use as the basis of active design reviews of the system.

To assess maintainability as a function of the environment supporting the system, the techniques available include:

- Software process maturity assessment and certification, which provide an overall impression of the ability of an organization to define, measure, reuse, and optimize its key processes.
- CASE tool assessment, which supports the selection and evaluation of individual tools to be used during system maintenance.
- Environment analyses and assessment, which take place via a combination of qualitative assessments of the development and maintenance environment within the context of the organization's past, present, and future maintenance goals.

Again, in this paper we have emphasized the latter techniques as being particularly valuable as it provides specific information that is tailored to the current system under investigation.

During the course of our work we have had the opportunity to examine a number of large, software-intensive systems with the aim of assessing maintainability. We have found that all of the above techniques can be valuable, and can be used to provide a spectrum of information relevant to a system's maintainability. A major difficulty is in taking this wealth of information and defining a realistic, cost-effective plan for improving the system based on these results. The technical information provided by making such a maintainability assessment suggests a number of courses of action for restructuring the system, improving documentation, increasing tool support, redefining key maintenance practices, and so on. These actions can then be costed and their impact on the project determined. It has been our experience that in this most important step it is programmatic, political, and economic factors that dominate the decisions made.

Acknowledgments

The authors gratefully acknowledge the work and good ideas of our colleagues on the audit team.

The SEI is sponsored by the U.S. Department of Defense.

6 References

1. Barry Boehm, *Software Engineering Economics*, Prentice-Hall, Englewood Cliffs, NJ, 1981.

2. Don Coleman, Dan Ash, Bruce Lowther, and Paul Oman, “Using Metrics to Evaluate Software System Maintainability,” *IEEE Computer*, Vol. 27, No. 8, August 1994, pp. 44-49.

3. IEEE Recommended Practice for the Evaluation and Selection of CASE Tools, The Institute of Electrical and Electronics Engineers, Inc. (IEEE), 345 East 47th Street, New York, NY 10017, 1992. ANSI / IEEE Std. 1209-1992.

4. David Lanning and Taghi Khoshgoftaar, “Modeling the Relationship Between Source Code Complexity and Maintenance Difficulty,” *IEEE Computer*, Vol. 27, No. 9, September 1994, pp. 35-41.

5. Paulk, M.C., Curtis, B., & Chrissis, M.B. *Capability Maturity Model for Software*. Technical Report CMU / SEI-91-TR-24, ADA240603, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, August 1991.

6. M. Pickard and B. Carter, “Maintainability: What Is It and How Do We Measure It?” *ACM SIGSOFT Software Engineering Notes*, vol. 18, no. 3, July 1993, pp. A36-9.

7. Charles H. Schumauch, *ISO 9000 for Software Developers*, ASQC Quality Press, 1994.

8. S. Wake and S. Henry, “A Model Based on Software Quality Factors Which Predicts Maintainability,” *Proceedings of the Conference on Software Maintenance*, Scottsdale, AZ, 1988, pp. 382-387.

9. David Weiss and David Parnas, “Active Design Reviews: Principles and Practices,” *Proceedings, Eighth International Conference on Software Engineering*, 1985, pp. 132-136.

10. W. M. Zage and D. M. Zage, “Evaluating Design Metrics on Large-Scale Software,” *IEEE Software*, Vol. 10, No.4, July 1993, pp. 75-80

Table 1: Change Scenarios and Their Scope

	Design Level Affected			Quality Attribute Affected		
Change Scenario	**Architecture**	**Design**	**Code**	**Availability**	**Performance**	**Subset**
Modify the user interface to the Monitor & Control console position’s user interface		✔	✔			
Import third-party-developed applications as major components, testing the system’s openness	✔			✔	✔	
Increase the system’s maximum capacity of flight tracks by 50%		✔	✔		✔	
Add new output displays to the system		✔	✔	✔		
Delete the requirement to support electronic flight strips by the system						✔
Upgrade to a higher-performance communication network	✔				✔	
Upgrade to a faster processor	✔				✔	
Migrate to X-Window System	✔	✔	✔		✔	