Assessing the Quality of Large, Software-Intensive Systems: A Case Study

Alan W. Brown, David J. Carney, Paul C. Clements,
B. Craig Meyers, Dennis B. Smith, Nelson H. Weiderman,
and William G. Wood

Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA 15213, USA

Abstract

This paper presents a case study in carrying out an audit of a large, software-intensive system. We discuss our experience in structuring the team for obtaining maximum effectiveness under a short deadline. We also discuss the goals of an audit, the methods of gathering and assimilating information, and specific lines of inquiry to be followed. We present observations on our approach in light of our experience and feedback from the customer.

1 Introduction

In the past decade, as engineers have attempted to build software-intensive systems of a scale not dreamed of heretofore, there have been extraordinary successes and failures. Those projects that have failed have often been spectacular and highly visible [3], particularly those commissioned with public money. Such failures do not happen all at once; like Brooks' admonition that schedules slip one day at a time [2], failures happen incrementally. The symptoms of a failing project range from the subtle (a customer's vague feelings of uneasiness) to the ridiculous (the vendor slips the schedule for the eighth time and promises that another $30 million will fix everything). A project that has passed the "failure in progress" stage and gone on to full-fledged meltdown can be spotted by one sure symptom: the funding authority curtails payment and severely slows development. When that happens, the obvious question is asked by every involved party: "What now?" The answer is often an audit.

This paper summarizes the experience of an audit undertaken by the Software Engineering Institute (SEI) in the summer of 1994 to examine a large, highly visible development effort exhibiting the meltdown symptom suggested above. The customer was a government agency in the process of procuring a large software-intensive system from a major contractor. The audit team included the authors of this paper, as well as members from other organizations. Members of the team had extensive backgrounds and expertise in software engineering, in large systems development,

To be presented at the European Conference on Software Engineering, Spain, September 1995.

and in the relevant application domain, but few had experience in conducting a thorough software audit. The deadlines of this one were inflexible, giving the team only 90 days to audit an extremely large system, and our struggle to come to terms with issues of basic approach, logistics, execution, and coordination consumed valuable time. It is the purpose of this paper to share our experiences, in the hopes that others finding themselves in a similar situation may benefit by having to spend less valuable time learning to do the job.

Section 2 of the paper provides the background of the audit, including a description of the system under development and the goals of and circumstances surrounding the audit. Section 3 describes the procedures that our audit team followed in order to meet its mandate. Section 4 presents a summary and commentary on our results, suggesting what worked well and identifying weaknesses in the approach.

2 Overview of the Audit

2.1 Goals

An audit may have many goals, and it is essential to articulate them carefully. Not all will be explicitly mentioned when the work begins. In our case, the goals (some of which only became clear after the audit was launched) were as follows, in approximate decreasing order of importance:

- Assessment of status. The customer had lost the ability to perform effective oversight, and hence could not judge whether the project was headed for fruition (albeit late and costly) or disaster.
- Assessment of salvageability. The customer needed to know whether, in order to field a high-quality system, it would be more cost-effective to perform a massive mid-course correction and press onward with the current effort, or abandon the development, write off the (quite significant) money already spent, and begin afresh.
- Satisfaction of funding authority. The funding agency (in this case, the United States Congress) was demanding that the customer produce a viable plan to bring the project to a successful conclusion; our audit was a significant part of that plan.
- Relation to a previous audit. The customer had received conflicting and incomplete information from a previous audit; our audit was to resolve some of the inconsistencies of that audit.
- Education. The customer desired to understand what went wrong in order to avoid repeating mistakes in future efforts.
- Gaining credibility. The customer had the implicit goal of adding credibility and objectivity to the development effort by employing a team of independent experts to conduct an evaluation. The development effort, were it allowed to continue, would thus receive the blessing of an outside, objective body.

- Protection of reputation: The contractor had the implicit and understandable goal of demonstrating value and quality in the existing (incomplete) work products.

2.2 General Description of the System under Audit

The target of the audit was a large, complex, real-time system of a command and control nature. The requirements of the system were stringent, having extremely precise hard real-time deadlines, with human safety being the critical basis for most timing requirements. The complexity of the system was also affected by the requirements for distribution, availability, and multi-site installation and site-specific customization. The availability requirements were very severe; to a large degree, these requirements drove the key decisions about architecture, test plans, modeling requirements, and many other aspects of the system. The system included both hardware and software, and was intended to replace an existing system.

The architecture of the system had a number of characteristics that distinguished it as inherently complex and largely unprecedented. The architecture was distributed and multiprocessor, using message-oriented and primarily client-server paradigms. In addition, the architecture featured redundancy in both hardware and software aspects. The system used table-driven data, with configuration of each site done through adaptation data that is read in and initialized during installation.

The audit focussed on the software components of the system, not the hardware aspects. The project was at a very late stage of development: software had been designed and developed over several years, and extensive testing had been performed and was in progress both at the development site and at a remote testing facility.

By the time of the audit the system development was several years late, millions of dollars over budget, and although close to final acceptance testing, many people at the user organization had severe doubts about the quality of the design and implementation of the system.

2.3 General Description of the Audit Process

The audit was conducted over a period of three months. The audit team included personnel drawn from two independent organizations as well as from the customer's staff. One independent organization had extensive background and expertise in the particular domain of the target system, and the other provided broad expertise in the areas of software engineering, fault-tolerant systems, and real-time software.

Organization of Team

The team included thirteen persons. One member from each independent agency performed administrative duties (although authority was always shared, neither choosing to assume full authority). The opening days of the audit provided evidence that the complexity of the system, and therefore of the audit, was such that it was not feasible for the team to function as a single entity throughout the course of the audit. The team therefore chose to divide into subteams. While there was no clear indication

of the optimal partitioning into subteams, the decision was made to divide into three broad subject areas: operational system (the system under audit); infrastructure (the set of facilities, tools, and processes that were used to build and maintain the system); and management (the practices used to monitor and control system development). There was a general agreement that the division into subteams was a convenient structuring device and nothing more. When some aspect of the audit suggested a different partitioning, then we would accommodate whatever different structure was necessary.[1]

Since the project had been monitored by various independent contractors over the course of several years, the audit team had access to some of their data and expertise. The system's contractor was also an active participant in the audit process. The contractor expressed full intention to cooperate in every way with the requests and needs of the audit team, and also expressed a deep interest in seeing that the outcome of the audit was a full and objective examination of the target system.[2]

Information Gathering

The audit team used a six-stage process to learn about the system and guide its investigation. These stages were: get information, clarify information, digest information, formulate questions, feed tentative conclusions back (to the customer and the contractor), and get additional clarification.

To establish the background and general concepts initially, we attended extensive briefings. Some of these were designed and presented by the contractor, and others by the government agency. These briefings provided the audit team with an overview of the system, its development, and its current state of readiness. We also read the overview design documents, and spot-reviewed detailed design documents.

To clarify areas where a particular sub-team lacked understanding, the team conducted follow-up meetings with the contractor, as well as interviews with persons from several other independent sources, including the customer, the testing personnel from the remote testing facility, the independent monitoring agencies, the subcontractor tasked with independent verification and validation (IV&V), and similar other sources. The audit team also interviewed personnel from the different subcontractors that had been participating in the project. As this information was digested, the sub-teams updated their models of the areas of interest to reflect the information gathered.

This process led to formulation of questions and tentative conclusions that were fed back to the customer and contractor. The process was iterated until the understanding of the issues was clear. Thus, to verify an assertion made about the system, the audit team asked the contractor to perform particular actions or demonstrations, or walk the

1. This in fact occurred when the team performed a detailed examination of the code.

2. This intention was fulfilled: the audit team was given access to any requested records or documents.

team in detail through a particular topic. The response to such a request was sometimes a prepared briefing or demonstration, sometimes a one- or two-day technical interchange between auditors and contractor personnel, and sometimes a day-long work session of team members and programmers huddled together over a workstation.

In some sub-teams, outside experts were consulted. This activity helped the team clear up remaining issues, gain perspective, and focus on where further information was needed.

It was not possible to learn all of the details, or read all of the documents, relevant to a particular topic. Some documents, such as the architectural overview documents, were available immediately; others became visible early in the audit; and still others became visible in answer to specific detailed questions.

Logistics and Mechanics

The team worked on a full-time basis throughout the three months. During the early phases of the audit the work tended to take the form of plenary meetings, often for briefings from the contractor or from other parties. During the later phases of the audit, most of the intensive work was conducted in subteam meetings. For the work of the subteams, there were always at least two persons conversant on any particular technical or programmatic issue under consideration. To prepare the final report, the team again met in full plenary sessions.

The findings of the audit were always the product of team consensus. Where differences of opinion were present, there was full debate on every point. This was often a painful and time-consuming process, but ultimately we found this to be a worthwhile (and perhaps necessary) aspect of the audit process.

3 Description of the Audit

In performing the audit, our point of departure was the three subteams (System, Infrastructure, and Management). The system subteam considered the operational system itself; the infrastructure subteam considered all supporting material used to build, test, and maintain the system; the management subteam considered all elements and activities that governed the creation of the system and its infrastructure. The following are the areas considered.

The system was examined with respect to

- documentation
- code quality
- performance
- fault tolerance
- system maintainability

The infrastructure was examined with respect to

- the development and maintenance environments
- configuration management

Management was considered with respect to

- problem trouble reports (PTRs)
- quality of process definitions and management

The audit of the system occupied the greatest amount of time, and those results occupied the largest part of the final audit report.

In this section, we provide a summary of the major issues that arose in performing the audit. Note that in this paper we are not providing the results of the audit, but rather the issues that arose while conducting it.

3.1 Code Quality of the System

In assessing the code of a large and complex system, there are several issues that are pertinent. The first issue concerns coverage of the audit: can the code be inspected in its entirety? And if not, what is the strategy for performing an audit that will provide a meaningful assessment of the code?

The code of the system approached one million lines, and given the schedule constraints we faced, it was impossible to fully examine all of it. The team therefore adopted a dual strategy. On one hand, we performed some investigations over the entire body of code. On the other we chose a few subsystems for close and deep examination. By mixing global searches with “slicing” the large system in this manner we were able to perform a reasonable code analysis, although in-depth analysis took place on a subset of the system.

A second key issue is that the code must be examined in the context of its requirements and design: does the code do what it is supposed to do? And does it do it in terms of the stated design? The former issue relates to verifying that the system’s requirements have been met, while the latter has additional importance for the evolution and maintenance of the code.

In assessing the system with respect to this issue, the team took the stated requirements that applied to a randomly selected subsystem and traced those requirements through the design documents to the code modules. This identified those requirements that were met, those that were not, how they were met, how the actual code conformed to the stated design, and so forth. It also contributed to the audit of the documentation of the system, since this portion of the audit uncovered inconsistences both in style and substance between the requirements documents, the design documents, and the code.

A third issue for a code audit is that the code must be examined with respect to some objective measures of code quality: does the code follow sound software engineering practices? These practices are at both a relatively high level (e.g., Does the code

exhibit a good degree of data abstraction?) and a relatively low level (e.g., Have the features of the target programming language been used properly and effectively?). One difficulty in this area is that there is often considerable disagreement about an objective measure of “sound practice” or “effective use.” The results of an audit in this area are often a set of observations, and possibly qualitative assertions, rather than an assessment in any hard metric sense.

We therefore examined the code with respect to the aggregate software engineering experience of the audit team. The results of this examination produced a list of observations and assertions about coding practices on many levels: modularization, data abstraction, exception handling, naming conventions, and so forth.

Finally, a related but distinct issue to the previous one is that, presuming that some project-wide style guide and conventions exist, does the code adhere to them? This is a distinct issue inasmuch as the project style guide may embody some questionable practices. Thus, an audit should reveal information on: how the code is (or is not) well-engineered, whether the project style guide has been followed; and whether the style guide itself mandates sound engineering practices.

In this respect the audit team compared the code with the recommended practices and conventions found in the project style guide, and identified the discrepancies between it and the code. At the same time, the audit team evaluated the style guide itself against the team’s shared experience in software engineering; this provided not only an assessment of the code’s conformance, but also some recommendations concerning revision of the style guide itself.

3.2 Performance of the System

It is beyond the scope or capability of a small, short-lived audit team to verify that performance requirements are sufficient and have been successfully met. Instead, the audit team can assess the developer’s treatment of performance during development, to try to establish whether or not sufficient attention has been paid and whether sound engineering practices have been followed. In order to assess these conditions, we addressed the following issues:

- system requirements
- system architecture and design
- performance drivers
- performance modeling activities
- verification of performance requirements

System Requirements

The auditor must understand the system’s requirements that affect performance. These include how performance requirements are partitioned onto different system components, the rationale for quantification of performance requirements, and whether performance requirements are critical to the system.

System Architecture and Design

Most significant for an audit is the rationale for the system architecture and design, and the degree to which it has been thought out and evaluated. By a combination of detailed reading of documentation, interviews with designers, and analysis of code samples we were able to assess these aspects. To focus our analysis we posed a number of questions that we then sought to answer. High-level considerations included

- To what degree has performance been a driver of the system architecture and design?
- How have performance considerations driven the system architecture and design?
- Has any prototyping work been done to assess design alternatives that may affect performance?

At a lower design level, the following questions were considered:

- Is the system characterized by independent entities that may be scheduled, or is there a central model for scheduling of activities?
- Do the architecture and design include the use of priorities? If so, what are they and what is the rationale, and how is the problem of priority inversion handled?
- How is the treatment of shared resources handled?
- Where appropriate, have recognized engineering methods (e.g., schedulability analysis) been applied?

Performance Drivers

We considered the following performance drivers as most significant:

- *Hardware*. One key question was the rationale for the selection of each hardware component. Other questions were how performance requirements may have influenced such decisions, and whether prototyping had been performed as part of the selection of hardware components.
- *Compiler and runtime system*. The large body of work related to performance assessment of compilers and runtime systems often embodies benchmarks. Hence, the audit questioned the rationale for the choice of the compiler and runtime system, whether and which benchmarks were used, and whether prototyping had been done in the selection of the compiler and runtime system.
- *Non-developed components.* This refers to hardware or software items that are part of the delivered system, but not developed by the contractor.[1] Examples of non-

1. Such components are often referred to as NDI (non-developed items). This term is more familiar in government acquisitions than commercial acquisitions. COTS (commercial off-the-shelf) is another familiar term.

developed components include databases, networking components, and implementations of interfaces that are based on standards. The key questions for NDI concerned whether management has specified that the system shall incorporate non-developed components and if so, whether processes and criteria exist for making such decisions and what verification procedures exist for such components.

- *Integration*. An integration perspective must evaluate a given performance driver in a particular context. For example, the assessment of the compiler and runtime system are done in a hardware context. So the question asked was why were this compiler *and* this hardware platform chosen?

Performance Modeling Activities

Analytic methods represent one form of modeling activity that may be applied to assess system performance issues; some widely-used techniques in this area include *schedulability analysis* and simulation. These techniques may be particularly useful in considering proposed system modifications. The issues we considered included

- What aspects have been modeled and what was the rationale?
- How many of the performance requirements have been included in the model?
- If a quantitative modeling approach was used, how robust is the model (i.e., how believable are the input parameters) and what level of detail has been included in the model?
- How are the results of the modeling used? For example, are they used to influence the design or implementation?

Analysis of System Performance With Respect to Requirements

Validation that system performance requirements are being met is achieved through testing (unit, integration, or system). In our audit we considered whether there was a sufficient test for each performance requirement, what testing mechanism were used for each performance requirement, how the performance requirements were found to be satisfied, and whether test generators were used to help generate tests and verify results (especially important in large systems).

3.3 Fault Tolerance

One of the most intrusive requirements of the system was fault tolerance. The availability required was regarded to be so restrictive that conventional hot-standby approaches would not work, since there was the possibility of common mode failures, and a likelihood of overloading the network with communications messages during fault recovery. Hence, a standby data management approach was taken, since this was the most likely way to achieve the desired availability. The audit team assessed the developer's approach to fault tolerance from architecture through implementation and testing to try to judge whether or not they would be successful in meeting their very stringent requirements.

The documents that were used as the basis for the investigation are listed below.

- engineering analysis documents written early in the contract to provide rationale that the system would satisfy the high availability requirements. These included documents describing experiments and analysis of the results of these experiments to demonstrate that the design approaches were sufficiently robust to meet the required availability. Too many studies had been produced to enable us to read them all, and therefore spot checking of a few studies was carried out.
- architectural documents and published papers explaining the hardware and software architectures, and the protocols used to achieve high availability. These formed the basis for the fault tolerance in the system; however, many of the important details were missing. These formed the basis for the initial understanding of the system (and follow-up questions).
- descriptions of the architecture of the fault tolerant operating system processes. This included a number of templates for various types of fault tolerant processes, state-machine like definitions of how the processing should be accomplished, and the message types and interfaces involved. Each application program had to conform to one of the design templates, depending on its characteristics.
- descriptions of the interfaces provided by the infrastructure to provide the services required for fault tolerance. These documents were quite lengthy, and were spot checked.
- various white papers written justifying specific design decisions and parameters, and describing the rationale for these decisions. These white papers were usually given to us in response to questions as they arose. The fact that the analysis had been done increased our confidence in the capabilities, and the documents were spot checked also.
- software code listings, and the results of static analysis of the code produced by independent contractors. Some code was inspected by team members, and the contractors walked team member through other parts of the code. The goal of the walkthroughs was to check conformance to the design templates.

At the completion of the process, the fault tolerance issues were organized into categories addressing the fault-tolerant infrastructure of the system, the fault-tolerant services made available to applications, the design and use of fault-tolerance templates, and the appropriate use of the fault-tolerant services by the applications. A list of outstanding risks with the fault-tolerance mechanisms was also developed.

3.4 Maintainability of the System

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 (e.g., [4]), and the system we were auditing promised to fall squarely into this class. Therefore, it was important to address the ability of the system to accommodate change and evolution.

We assessed maintainability by investigating several areas, including

- documentation, through which a maintainer learns which parts of the system are (and are not) affected by a proposed modification
- architecture and high-level design, which embody the major design decisions
- low-level design (e.g., code modularization strategy), which embodies the majority of the encapsulation and information-hiding decisions
- implementation, which determines how well the promises made by the documentation are kept, and how well the policies dictated by the design are followed

Maintainability is also a function of the appropriateness of the maintenance environment, which we addressed elsewhere in our audit (see Section 3.5).

Rather than just try to assign a scalar maintainability metric, which in our opinion is without operational meaning, we assessed maintainability in the context of likely life-cycle evolution scenarios. Our audit procedure took into account the domain-specific, project-specific, and organization-specific aspects of maintainability. It proceeded as follows:

- We enumerated the quality attributes that were 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 in order to accommodate a contingency plan to field a scaled-down version of the system.
- We enumerated a set of change classes likely to occur to the system over its lifetime. These change classes may come from anticipated requirements for this system, or domain knowledge about changes made to legacy systems of the same genre. The system under audit was an embedded, real-time, reactive, user-in-the-loop, safety-critical system operating at the edge of its performance envelope. Systems of this genre typically undergo the following classes of change:
  - replacement of hardware: computers, display devices, networks, input devices, etc.
  - replacement of system-level software: operating system or network controller upgrades, new compiler releases, etc.
  - incorporation of third-party components: a commercial display driver, employing elements of a reuse library, etc.
  - changing the so-called “quality attribute” requirements: performance, availability, safety, security, etc.
  - adding, deleting, or changing functionality: changing the display symbology, the input language, the layout and/or contents of a display, adding new information to a display, etc.
  - making the system interoperable with new systems in its environment

- We made sure that our list of change classes covered each of the quality attributes listed above (e.g., increase the system's availability requirement), as well as covering the system at the architectural level (i.e., affect its highest-level components), the module level, and the code level.[1]
- 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 inputs the system was required to monitor.
- 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 low-level code modules) and documentation that would be affected by the change. The result was a set of *active design reviews* [5] in which the participants were pro-active, each in his or her own area.

The purpose of the change scenarios was 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.

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 specific 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 straightforwardly modified with respect to normal life-cycle evolutionary pressures. One aspect of this "unguided" part of the investigation is to inquire after the design rationale to see what information was encapsulated in each component, whether at the system-level, module-level, or package level. This encapsulation implies a set of changes that the designers had in mind, explicitly or implicitly, against which the resulting system is insulated. This step includes the use of standard code quality metrics, as well as traditional documentation inspection and quality assessments.

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

3.5 Development and Maintenance Environments

Existing approaches to evaluating software development and maintenance environments fall into one of two categories: a technology-oriented view that concentrates on the selection of individual computer-aided software engineering (CASE) tools; and a more process-oriented view that concentrates on assessing the practices used during development and maintenance. We incorporated both of these views in our approach, since we believe that tools, techniques, and processes should not be considered in isolation. Rather, our notion of an "environment" is the combination of all three, each providing context for the others. 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.

We also found it essential to concentrate attention on how the development and maintenance environments specifically apply to the current system being maintained. In particular, this attention included examining the goals of the organization that developed, is maintaining, and is using the system in question. These considerations led us to assess the software development and maintenance environments by performing the following analyses:

- comparing the development and maintenance environments
- evaluating the plan for transition of responsibility from development to maintenance
- assessing the key maintenance practices
- examining the organization's maintenance of other systems

Comparing the Development and Maintenance Environments

The development environment leaves a legacy of documents, data, and knowledge concerning the system that must be brought forward into maintenance. The accessibility of these artifacts are strongly impacted 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 impossible. We therefore compared the two environments by focussing on four key questions:

- When is the maintenance environment instantiated?
- 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 adequate for the maintenance personnel to carry out their task?

Evaluating the Plan for Transition of Responsibility

For many projects, 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 the system 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. We considered the following documents to be essential:

- a high-level overview of the architecture of the system that establishes the major design requirements for the system, the implementation choices made to meet those requirements, and the typical operation of the system
- a detailed transition plan for moving the system to maintenance that defines the tools, techniques, and practices to be used in maintenance, the responsibilities and expectations of all participants, and so on

Assessing the Key Maintenance Practices

The key practices that take place during maintenance parallel the key practices that occur during development. But while most software projects consider these as critical aspects of the development phases, they are often severely neglected when establishing the maintenance environment. We examined a number of key maintenance practices to ensure that the practices were well-defined, adequately documented, and had been agreed to by all relevant organizations. The key process areas that we examined included the code inspection processes, the code bug-fix process, the integration test process, the system build and release practices, and the system change request procedures.

Examining Maintenance of Other Systems

Most organizations simultaneously maintain many large systems. Hence, the maintenance of one large system cannot be considered in isolation; many decisions must take a wider picture of maintenance that provides consistent maintenance practices across the organization as a whole. Of these decisions, we considered four to be paramount.

First, since the system may interface with a number of existing or planned future systems, many decisions (e.g., system interfaces) may have been fixed. This provides substantial design challenges during maintenance. For example, the system we examined interfaced to a wide range of systems constructed over a 25-year period. The continued correct operation of these systems was paramount in any proposed enhancements to the target system.

Second, maintenance engineers have an existing technology base for maintaining systems. The maintenance environment for a new system must harmonize with this existing environment. We examined the planned maintenance environment in the context of the existing maintenance activities of the organization.

Third, the recent climate of systems development toward the use of commercial off-the-shelf (COTS) components provides significant maintenance challenges: large parts of the system are maintained by COTS vendors, access to detailed information on the operation of COTS software is often severely limited, new releases of COTS software occur at the choice of the vendor, and tracing errors can be problematic in systems that include COTS components. In our study we paid particular attention to maintenance activities for COTS components of the system, and examined contingency plans for events such as new releases of COTS software, tracing bugs in COTS software, and actions to be taken if the COTS vendor went out of business.

Fourth, maintenance is required not just for the operational system, but also for all of the software needed for development, testing, maintenance, and release. In comparison with the operational software, the support software in most large systems can be more extensive, in multiple languages, and poorly documented. We attempted to ascertain which support software was essential to the ongoing operation and maintenance of the system (e.g., database systems for data entry and manipulation, assembler code for network support, test scripts written in a high-level scripting language), and to ensure plans were in place to maintain and evolve the information needed to maintain them. This included maintaining large amounts of documentation, data (e.g., test data), and administrative information on support system configurations using during development.

3.6 Documentation

During any large software project there will be a large amount of documentation generated. This is particularly true in projects such as the one discussed in this paper that take place using a variant of the Department of Defense standard development approach, Mil-Std-2167A. In this approach there are a number of points at which detailed documentation is produced that is used as the basis for project reviews.

As part of the audit we spent considerable time examining this and other documentation. Initially, we examined the documentation in order that we ourselves could obtain an understanding of the system. Later our examination was based on the quality of that documentation as it applied to

- others interested in finding out about the system
- software developers as they sought guidance on technical questions concerning how to implement parts of the system
- system maintainers as they attempted to fix and evolve the system
- end users as they tried to operate the system in the field

For each of these classes of users there are clearly different sets of documents that are of interest, and different document qualities that are of importance. Hence, we attempted to consider the major needs of each of these users and to consider whether the documentation was adequate for those needs.

Additionally, we considered some generic qualities of the documentation that we believe are fundamental to good software engineering practice. Namely, that the documentation be well-written, comprehensive, internally and mutually consistent, and readily accessible. To do this we carried out a number of simple analyses based on realistic scenarios. For example, we selected a number of requirements, tried to find where they were documented, attempted to trace these requirements to documents containing the key design decisions that they influenced, and eventually to pieces of code that implemented them. Such scenarios proved very valuable in revealing whether the documents could be easily navigated, contained accurate information, and captured the information that is needed by practicing software engineers.

3.7 PTR Analysis

Problem Trouble Reports (PTRs) give valuable insight into both the software product and the ongoing software process. In the case of the software product, the problem discovery rates and the problem fix rates give some indication of product volatility and how close the product is to completion. In the case of the software process, the management and control of problem handling is one indication of the overall level of management and control of the entire software process. The PTR process is appealing as an indicator for a software audit because it is discrete and easily analyzed within a well-defined time frame.

Problems are not restricted to “bugs” in the program under construction, but are defined more broadly. PTRs are initiated for performance enhancements, or to accommodate external changes (e.g., in the system’s operating environment). They are also opened to report bugs in support software or commercially available software, or to report documentation errors. In the audit conducted by our team, the PTR database contained more than 25,000 records with each record containing over 200 fields of information.

During the course of the audit, the software audit team conducted the following activities:

- interviewed members of the problem management group
- reviewed PTR process documents and the PTR model
- reviewed PTR status reports and graphs
- sat in on a PTR Review Board (PRB) meeting
- reviewed a random sample of individual PTRs

The purpose of the initial interviews with the problem management group was to gather information about the definition of PTRs and to gain an understanding of the overall PTR process, including the life cycle of a PTR. Along with initial interviews

came documentation of the process that could then be studied off-line. Later we met with this group again when we were in a position to ask more probing questions. We also studied status reports and graphs of PTRs over a several year period showing various metrics including the discovery and closure rates broken down by types of problem, type of module, and severity of problem. We examined the mathematical model for predicting PTR activity based on historical data and life cycle stage.

The PTR Review Board is a group that reviews incoming PTRs and assigns them to be fixed by a certain group by a certain time. By sitting in on a regular meeting, an audit team is less likely to be manipulated by selective information disclosure. In our case we attempted to sit quietly to the side while the meeting was conducted in the usual manner. Finally, we inspected a random sample of actual PTR records in order to discover the integrity, consistency, and completeness of the database.

Among the questions that can be answered during a software audit by a study of PTRs and the PTR process are the following:

- Is sufficient information collected on each PTR, including its status, its history, and its criticality?
- Is the change management process well documented?
- Are the documented change management procedures followed?
- Does the PTR model accurately predict the PTR discovery and fix rates?
- Are PTRs entered during the analysis, design, and implementation stages or only during integration and testing?
- Are the fields in the database complete and consistent so that they produce accurate reports?
- Are PTR reports produced at regular intervals and analyzed consistently based on consistent definitions?
- Is management using PTR data to find the root causes of problems and to identify parts of the software and parts of the organization that are causing high error rates?
- Is number of outstanding PTRs decreasing or increasing?

It is important to understand that PTRs must be viewed in the context of the current stage of the life cycle. It is normal for PTRs to increase during integration and test. However a sure sign of trouble is when PTR levels stay the same or increase over long periods of time. This is one indication of volatility in requirements. The PTR model used by the contractor in our audit projected that a mere 7% addition of new code from one build to the next would preclude any diminution in the number of outstanding PTRs over time.

3.8 Management Issues

Although the audit focused primarily on the current technical software product, the technical issues often led back to management issues. Deficiencies in the product

provided inferences of a chaotic management process on the side of both the government and the contractor. Our approach was a departure from other approaches, which tend to focus exclusively on either the product or the process. By using product deficiencies to point to process and management issues, our conclusions had an empirical grounding, and provided a balance between product and process.

Product-oriented audit investigations revealed major problems in areas of code quality and documentation. The lack of quality in the product led us to investigate how well processes had been defined and management-level enforcement of these defined processes in areas such as code inspections, system testing, and software quality assurance. Some of these processes had been defined, but their sustained use appeared to have been inconsistent.

In particular, our audit of management practices and procedures of the contractor addressed the following areas of concern:

- Was there a significant attempt to learn from the past through root cause analysis of problems? A symptom of a problem in this area was code that had been developed with errors that should have been caught at earlier stages of development.
- Did the contractor manage shifting requirements effectively? Were requirements changes accepted without sufficient analysis for cost, schedule, and impact on the rest of the system?
- Did management tend to rely on a small group of experts, as opposed to a stable, strong process? If the latter, is it championed by a strong leader?
- Overall, was quality or expediency the more highly valued goal? What are the developer's quality improvement and quality assurance plans and processes?
- Were problems ever addressed frankly by the contractor management in the months before the audit, when trouble was brewing? This might be evidenced by such actions as internal audits, identification of process deficiencies, development of a process improvement plan, etc., and the existence of engineering teams to implement the improvements and dedicated management teams to monitor them.

It must also be recognized that improvement takes time and requires monitoring long after the audit team has disbanded. One result of such an audit may be a list of areas that the customer must monitor in the future. They may include monitoring items such as

- progress in meeting the objectives of the process improvement plan
- evidence that a viable inspection process has been successful
- use of the program trouble report database for root cause analysis and management decision making
- development of an independent quality assurance function
- verification of successful schedule and project planning

- initiation of risk mitigation activities
- establishment of a metrics program for management decision making
- schedules and milestones for fixing problems addressed in this audit

On the customer's side, we tried to assess whether their management of the developer and its sub-contractors was strong, reasonable, and consistent. Areas we investigated included

- the number of requirements changes and versions mandated by the customer
- the ability (or inability) of the customer to control the expectations of users;
- degree of visibility into, and monitoring of, contractor work;
- quality control of contractor products
- quality or existence of joint risk management processes;
- whether schedules were driven by management or political needs, rather than technical reality
- degree of effective use of information provided by support contractors (e.g., failure to raise program trouble reports based on information they provide)

4 Comments and Summary

Performing a software audit is typically a stressful task undertaken under great pressure to produce results in a relatively short time. The audit discussed in this paper was typical in this regard: it took place with minimum time for preparation, and using personnel that were largely inexperienced in carrying out such audits. However, we believe that in many regards the audit was successful. We

- documented the current system in a form that highlighted its major characteristics and identified areas of major technical risk
- provided an assessment of many of the aspects of the system that have a direct relevance to the quality of the overall design of the system
- examined the major documentation describing the system and its implementation, and provided many suggestions for improvement
- looked at parts of the system implementation to assess the fidelity of the implementation to the design, and to ensure that the implementation was of high quality
- considered the engineering environment and practices being used to complete and maintain the system to ensure that they were adequate for the predicted life of the system

As a result, the information we provided enabled the customer to make appropriate decisions concerning the future of the program being audited. In retrospect, we are able to identify a number of factors that we believe contributed substantially to the success of the audit. These included

- *Personnel*. The audit team consisted of experienced software engineers with a range of technical skills that matched the fundamental characteristics of the system we examined (e.g., experienced in fault-tolerance, performance, and distributed real-time systems).
- *Customer interactions*. We began by negotiating a clear statement of work with the customer that included a well-defined scope and set of objectives for the audit. As the audit progressed, the pressures to amend these goals and objectives had to be rigorously resisted.
- *Contractor interactions*. Early in the audit we established a good relationship with the contractor by demonstrating our technical capabilities, and by being clear that our role was to provide technical data to the customer, and not to offer rash opinions based on little information.
- *Internal organization*. During the period of the audit we received hundreds of documents amounting to many thousands of pages of text, we attended dozens of meetings, and we communicated frequently with the contractor and the customer. A dedicated, responsible support person was essential to the project to manage this information and to organize and distribute relevant material, plan travel and meeting details, and ensure relevant electronic communication mechanisms existed between the team members.

We also recognize a number of criticisms of our audit approach. These include

- *Narrow focus.* Some members of the customer organization were disappointed that we did not address the cost and resource implications of the problems we identified. The customer had a number of support contractors whose role on the project was to plan and monitor such aspects, and our approach was always to refer the customer to these support contractors.
- *Fixed audit scope*. During the audit the customer made decisions concerning the future of the program that led to some of the findings at the end of the audit being irrelevant to the customer. We were also unable to say how some of our findings related to this new strategy, as the strategy was not considered during the audit.
- *Lack of quantitative data*. Wherever possible we justified our comments with facts and data from the system. However, the customer had expected quantitative data that objectively measured many quality attributes of the system. We had to explain that metrics to measure these attributes do not exist.
- *Avoidance of "finger-pointing"*. We decided that the outcome of our audit would not focus on assessing blame. This is seldom a useful exercise, since interpretations and excuses can always be found to contradict any assertion of blame. Also, the potential candidates for blame were on all sides of the question; selecting a scapegoat from these possible candidates would not be valuable to anyone.

Apart from the value to the customer, the team itself found the experience of performing a post-mortem software audit to be an interesting and valuable one. Many

important lessons were learned concerning the state of the practice in developing large, complex software systems, and the difficulties of project management and monitoring. These have only briefly been discussed in this paper. We are currently in the process of documenting the lessons learned from our audit experiences, and hope to provide a set of more detailed guidelines for audit teams faced with a similar situation to ours.

Acknowledgments

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

While responsibility for this report lies with the stated authors, we gratefully acknowledge the contributions made by other members of the audit team to the design and execution of this audit.

References

1. *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.

2. Fred Brooks, *The Mythical Man Month,* Addison Wesley, 1975.

3. *Scientific American*, "Software's Chronic Crisis," September 1994.

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

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

6. Alan W. Brown, David J. Carney, Paul C. Clements, "A Case Study in Assessing the Maintainability of a Large, Software-Intensive System," *Proceedings of the International Symposium on Software Engineering of Computer Based Systems*, Tucson, AZ., IEEE Computer Society, March 1995.