93-F-1342

USAISEC

US Army Information Systems Engineering Command
Fort Huachuca, AZ 85613–5300

U.S. ARMY INSTITUTE FOR RESEARCH
IN MANAGEMENT INFORMATION,
COMMUNICATIONS, AND COMPUTER SCIENCES

Software Quality and Testing: What DoD Can Learn from Commercial Practices

ASQB–GI–92–012

31 August 1992

AIRMICS
115 O'Keefe Building
Georgia Institute of Technology
Atlanta, GA 30332–0800

X35C

Software Quality and Testing: What DoD Can Learn from Commercial Practices

Prepared for

the OASD(C3I) Director of Defense Information

and

the Deputy Undersecretary of the Army (Operations Research)

31 August 1992

by LTC Mark R. Kindl

Army Institute for Research in Management Information,

Communications, and Computer Sciences (AIRMICS)

DISCLAIMER

This research was performed by the Army Institute for Research in Management Information, Communications, and Computer Sciences (AIRMICS), the research organization of the U.S. Army Information Systems Engineering Command (USAISEC).

The findings of this technical report are not to be construed as an official Department of Defense or Department of the Army position unless so designated by other authorized documents.

The use of trade names in this document does not constitute an endorsement or approval for the use of such commercial hardware or software. This document may not be cited for the purpose of advertisement.

ACKNOWLEDGMENTS

I wish to express my appreciation to the following individuals for their valuable suggestions and assistance during the research for and preparation of this report:

Mr. John Mitchell, Director, AIRMICS
Mr. Glenn Racine, Chief, CISD, AIRMICS
LTC David S. Stevens, Computer Scientist, CISD, AIRMICS
Mr. Earl Lee, Senior Test Engineer, IBM Federal Systems Company Houston
Mr. Andrew Chruscicki, Air Force Rome Laboratory
Mr. Ray Paul, Army Operational Test and Evaluation Command

EXECUTIVE SUMMARY

Historically, software testing was the process of exercising a computer program to verify that it performed as required and expected. The strategic goal of software testing was to demonstrate correctness and quality. We now know that this view of testing is not correct. Testing cannot produce quality software, nor can it confirm correctness. Testing can only verify the presence (not the absence) of software defects. Yet, the difficulty of testing and the impracticality of correctness proof have often driven us to the dangerous perception that if testing does not find defects, then the software is correct.

In the early 1980s, software testing concepts were neither well-developed nor well-understood [1, p.39]. While testing techniques were many, supporting theories were few. Even worse, little or no guidance existed for making intelligent choices of technique(s) [2, vol. 1, p. 24]. During the 1980s, Department of Defense (DoD) and industry gathered much empirical evidence to justify many software quality and software development techniques. As a result, the scope of software testing has evolved into an integrated set of software quality activities that cover the entire life cycle [3]. Software tests now take different forms and apply to all software products including requirements, design, documentation, test plans, and code. Each test contributes to a total quality assurance plan. Quality assurance focuses on the front of the development process and emphasizes defect prevention over detection. A cost-effective prevention program first requires accurate error detection and analysis to understand where, how, and why defects are inserted. Though testing cannot prevent errors, it is the most important method for producing error data necessary to guide process improvement. However, the following extract from the 1992 Software Maintenance Technology Reference Guide [4] summarizes the difficulty of testing:

> *"Software implementation is a cozy bonfire, warm, bright, a bustle of comforting concrete activity. But beyond the flames is an immense zone of darkness. Testing is the exploration of this darkness."*

The conclusions of this report are not revolutionary, but they may be surprising. DoD knows how to produce quality software. There are a few contractors who produce quality software, (though not necessarily for DoD) using many of the policies published in DoD Standards. These documents describe the need to focus on quality activities early in the software life cycle. Developers and verifiers should identify and remove errors during requirements definition and design so that they do not enter the code, where finding and fixing defects is extremely expensive. For management information and command/control

systems this is a particularly difficult task because most requirements for these systems are based upon human demands which are highly subjective, easily influenced, and thus, very dynamic and difficult to state precisely.

Although not in common practice yet for software development, quality control methods adapted from the factory paradigm [5] may have the greatest potential to move software production from an art to a true engineering discipline [6, 7, 8, 9]. Both the products and the development process should be subjected to these procedures. To engineer quality into the software products requires that we inspect/test and remove defects from requirements, design, documentation, code, test plans, and tests. Quality control of the development process requires that we establish standard procedures to measure defects, determine their root causes, and take action to prevent future insertion. Such a process is self–correcting, and future measurements will provide convincing evidence of cost–effective improvement. In summary, software quality improvement is *evolutionary* and requires that we control, coordinate, and feedback into three concurrent processes: the software development process, the error detection process (testing life cycle), and the quality improvement process. Figure 1 depicts the relationships between the processes in the software life cycle.

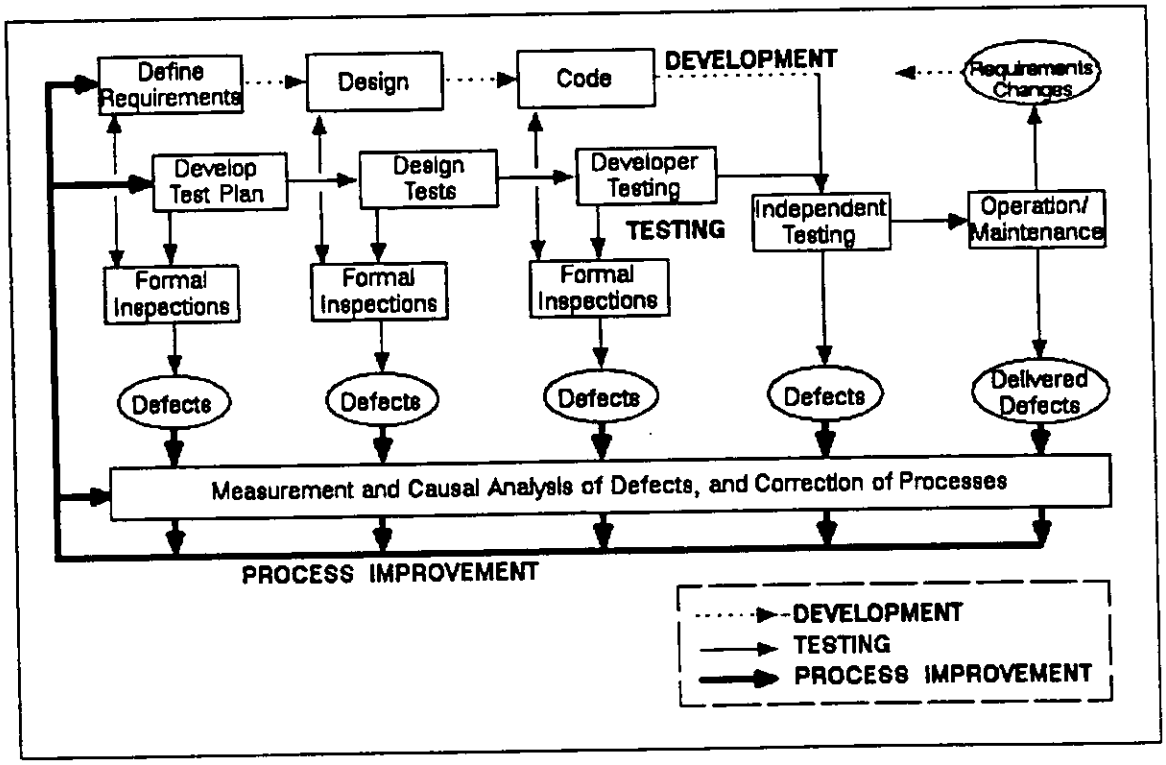

Figure 1. Software Quality Control

A few corporate organizations have successfully implemented these procedures [10, 11, 12, 13, 14, 15, 16, 17]. The common key element in these successes is organization–wide commitment to a quality attitude and disciplined life cycle procedures. However, within DoD the perception persists that such practices are not cost–effective. Simply mandating their use has not been adequate. Even if enforced, the techniques can be undermined, and neither software quality nor the perceptions will change [10]. DoD must jump–start these procedures with an active campaign to establish and nurture a quality attitude both internally and in its contractors. IBM Federal Systems Company (FSC) Houston took 15 years to refine their processes into producing high quality software. But, it also believes that other organizations can learn from their procedures without investing such time. What can make this possible is the fact that their procedures already correlate well with written DoD policies, the policies of other corporate software developers, and the recommendations of academia. The difference is that IBM has disciplined itself to practice them. DoD should take advantage of this knowledge and experience now, and adapt its own *practices* accordingly.

In order to initialize the production of higher quality software within DoD, we recommend the following actions:

(1) Actively motivate a software quality attitude in DoD and government contractors through management commitment, incentives for process improvement and quality, and technical training. Make quality as visible as the software product, its cost, and its schedule. For every change to software product, cost, or schedule, DoD project managers must give equal consideration to the corresponding cost of and effect on quality.

(2) Motivate and make standard the use of formal inspections for all software products (requirements, documentation, design, code, test plans, tests).

(3) Users, developers, and verifiers should jointly analyze requirements to ensure they are clearly documented, implementable, and testable. The formal analysis of quality objectives should be an integral part of this effort. A joint relationship should continue throughout the software life cycle. Eventually, this effort should result in documentation or data that directly cross–references test cases to requirements and code. At the same time, both developer and verifier should independently plan, design, develop, inspect, execute, and analyze the results of software tests.

(4) Measure and document errors throughout the life cycle. Establish a formal defect prevention program which empowers developers and verifiers to analyze the causes of error and enact improvements to their own local

development processes that will prevent future error insertion and enhance detection processes.

(5) Evolve Computer-Assisted Software Engineering (CASE) tools to support all aspects of software development, testing, and maintenance. DoD should permit organizations to introduce standard CASE tools gradually in piece-meal fashion. An organization should purchase, train, and employ only those tools for which its sub-processes are defined in writing. Start small and allow adequate time to learn and gain experience. Purchase and integrate a new tool only when users understand the manual procedure the tool will automate, and the benefit of automating it.

With regard to software testing in DoD, we can summarize our conclusions in two fundamental ideas. First, DoD knows how to produce quality software at low cost. This is because organizations such as DoD STEP, Army STEP, and Software Engineering Institute have already researched and documented policies for DoD. A few commercial software developers practice many of the DoD policies and directives now, and produce quality software (for example, IBM FSC Houston). Second, quality cannot be tested into software. Only a well-defined, well-disciplined process with a continuous improvement cycle can ensure software quality. However, testing cannot be underestimated. Systematic testing activities that detect error earliest in the life cycle are necessary to drive process improvement and optimize the development of quality software. Such testing methods as formal inspection find defects early. This enables cost-effective error resolution, identification and removal of defect causes, and thus, prevention of future defect insertion. If practiced with discipline, such methods can evolve a self-correcting software development process that is stable, modeled, measured, and therefore, predictable. This development process engineers quality software faster at reduced cost.

This report discusses software testing practices, and more specifically, why and how IBM's practices achieve high quality. Along the way, we will relate DoD policies, instructions, and guidance to IBM's practices. We will also discuss current initiatives within DoD which will impact software testing and quality. Finally, we present our specific recommendations for software testing and quality within DoD. We believe that these recommendations have the potential for immediate value to DoD.

Administration (NASA), currently approaches .01 errors per thousand lines of source code [10]. This figure is well below the U.S. industry average. Surprisingly enough, there is nothing new or revolutionary about the way that IBM FSC Houston develops or tests its software. Many of the same methods are used at IBM FSD Rockville, as well as at other large software development corporations. IBM FSC Houston practices basic software life cycle processes, most of which have been known for at least a decade. These include requirements analysis, formal inspections, configuration control, quality control, developmental testing, and independent verification and validation.

So, why does IBM FSC Houston produce such high quality software? The difference results from a strong *attitude* toward quality, the disciplined practice of its *basic processes*, and a commitment to *process improvement*. From manager to programmer, the entire organization strives to achieve zero–defects through *prevention*. To the classical waterfall model of software development, this organization applies basic testing processes designed to identify errors as early as possible. Once identified, defects in the software products are corrected. However, continuous measurement, causal analysis, and subsequent cause removal improves the development process and prevents future error insertion. Their techniques are very closely related to concepts of Total Quality Management (TQM) [22] and the software factory paradigm [5]. Recent empirical evidence in other organizations [10, 11, 12, 13, 14, 15, 16, 17] confirms the effectiveness of the software quality techniques practiced by IBM FSC Houston. Later, we will discuss the techniques in more detail and relate them to the DoD environment.

Critics maintain that because of fundamental differences, software techniques used to develop and test weapons systems cannot be used efficiently or effectively to produce information systems (and the reverse). IBM also believed this until the late 1980s. However, on the basis of its own success in developing high–quality flight control software, IBM FSC Houston began to develop its ground system software (essentially MIS) using the same methods. The empirical evidence speaks for itself. Error rates in delivered ground software decreased dramatically to the same levels achieved in flight software. Furthermore, this similarity in quality occurs in spite of the more extensive testing that safety critical flight software undergoes [10]. The quality achieved with early error detection and prevention techniques is largely independent of the type of software being developed.

The development of large DoD Management Information Systems (MIS) and Command/Control Systems (C2) software is costly and time–consuming. Much of this cost and time can be attributed to the identification and repair of errors.

Closely related defect repair is maintenance — re–work necessitated by changing requirements or latent defects. In such cases, software in operation must be modified to reflect new requirements or requirements that were initially ill–defined. To reduce the cost and time to produce and maintain software, DoD must avoid passing immature software to the testing phase, or worse, to the customer.

Testing is one of the most important quality tools. Properly applied, testing helps to identify one of the greatest impediments to quality — error. However, if quality software is the ultimate goal, then any discussion of effective software testing must address the entire software life cycle. This is because testing alone can neither produce nor guarantee software quality. Testing only finds faults; it cannot demonstrate (in a practical sense) that faults do not exist. What we have traditionally thought of as software testing tends to be labor–intensive, costly, and ineffective. This view of testing is a paradox. Testing is a process that instills confidence in software by cleverly plotting to undermine that confidence [23]. Nevertheless, there is empirical evidence to suggest that old concepts of quality control can counter this view. By expanding the concepts and practices of software testing to all areas of the life cycle, we can optimize test efforts, increase its effectiveness, and significantly reduce its cost. The result will be the delivery of higher quality software on schedule for less money.

One reason for general difficulty in testing software appears to stem from differences of testing models conceived in the minds of users, managers, developers, analysts, and testers [3]. Without common accepted concepts, all vital communication in large software development projects will amount to assumptions and guesswork in the best case. Therefore, in order to clarify further discussion, we summarize several fundamental definitions from the ANSI/IEEE *Glossary of Software Engineering Terminology* [24], considered industry standards:

error – a discrepancy in implementing requirements or design specifications. An error *may* manifest itself as incorrect or undesired results.

fault – a defect in code that has the potential to cause (possibly visible) incorrect or unexpected results. Faults are also known as *bugs*. Faults in code usually result from errors.

debugging – the process of locating, analyzing, and correcting suspected faults.

failure – the execution of software fault or defect that manifests itself as incorrect or undesired results.

testing – the process of exercising or evaluating a system or system components by manual or automated means to verify that it satisfies specified requirements or to identify differences between expected and actual results.

dynamic analysis – testing by executing code.

static analysis – the process of evaluating a computer program without executing it; e.g. review, desk check, inspection, walk–through.

correctness – use of this term usually means the composite extent to which:
(1) design and code are free from faults
(2) software meets specified requirements
(3) software meets user expectations

verification –
(1) the process of determining whether or not the products of a given phase of the software development cycle fulfill the requirements established during the previous phase.
(2) formal proof of program correctness.
(3) the act of reviewing, inspecting, testing, checking, auditing, or otherwise establishing and documenting whether or not items, processes, services, or documents conform to specified requirements.

validation – the process of evaluating software at the end of the software development process to ensure compliance with software requirements.

Several of these terms have subtle relationships and differences in meaning. It is important to recognize that errors relate to early phases of the life cycle — requirements definition and design specification. An error in requirements or design causes the insertion of a fault into code. However, a fault may not be visible during code execution, whether during testing or operation. If a fault is executed, then it may result in a visible failure (but not necessarily). Programmers debug code to correct faults by using visible failures as a guide. However, the lack of failures cannot guarantee the absence of faults. Even if the fault executes, it may not be visible as output. Furthermore, fault correction does not necessarily imply that the error(s) that induced the fault has been corrected.

From the above discussions, one should conclude that effective software testing cannot be limited to code. It must address all products of the software life cycle. The definition implies that testing demonstrates:

(1) that the code satisfies a specific requirement.
(2) whether faults exist in the code.

However, these are only the ideal goals of testing. In practice, they cannot be achieved in the absolute sense. Furthermore, these goals are not necessarily mutually exclusive. Code can often satisfy user requirements (as defined) and still contain faults. The definition can easily convey the erroneous perception that testing can verify correctness. Correctness is a major factor in software quality, and by definition, relates to code, requirements, and user expectations. But, testing code only verifies the presence (not absence) of faults in code, and cannot verify correctness or ensure quality. Testing code can verify the presence of requirements only if they are defined precisely as test cases. Developers and users do not normally view requirements in this manner. Effective testing identifies errors before they become code faults, and therefore, must apply to the entire life cycle.

Since the 1980s, the scope of software testing has expanded to cover the entire life cycle [3]. Empirical data from software projects in the last decade provides convincing evidence that testing in this context can significantly improve software quality. In its current model, software testing has a variety of forms that apply to a range of products including requirements, design specifications, documentation, test plans, as well as code. These techniques must be coordinated, disciplined, and integrated throughout the entire life cycle to effectively impact on quality. We will make the case that to have maximum positive effect on a large software project, testers must participate in development and gain a broad understanding of the software requirements and design. Therefore, in the remainder of this report we will refer to software testing professionals as verifiers to highlight their expanded roles consistent with the definitions above.

3. Involve Verifiers in the Entire Development Life Cycle

DoD STEP reports [25, vol. 3] indicate that the most successful DoD software projects established independent test and evaluation organizations. Sometimes these organizations were separate independent contractors. Other times they were sub–organizations under the prime contractor, but having an independent chain of command. This is an effective strategy which is in common practice to help ensure objective, impartial, and unbiased testing. DoD directives provide for such independent testing activities. Each military service has its own independent test and evaluation organization. General IBM testing policies also define the need for such. Both IBM FSD Rockville and IBM FSC Houston have independent verification organizations within their respective projects.

The advantages of independent testing should not overshadow the need for communication and coordination between verifier, user, and developer. While verifiers should plan, design, implement, and analyze software tests

independently, they should not do so in isolation. Verifiers who must design and perform operational tests cannot gain adequate understanding of the requirements of a large software system by studying the system documentation after development. They must take an active role in the requirements definition and system design phases.

The biggest mistakes in software are almost always made early during requirements definition and design [26]. Empirical evidence indicates that the cost of fixing errors versus time in development is an exponentially rising curve. IBM FSC Houston data shows that average error repair costs increase 10 times in each successive phase of the life cycle [10]. As a result of such data, in the mid-1980s, IBM FSC Houston decided to move 30% of its resources used in testing of code to assist in the requirements definition and design phases. This decision resulted in a significant increase in software quality. Furthermore, this shift resulted in a net decrease in total cost. Shell Research reported similar results [12]. The conclusion is obvious — verifiers should participate in requirements analysis, definition, and design. It is far cheaper to find and fix errors before they become faults in the code.

DoD STEP identified the need for early test and evaluation activities in software development. It also identified the need for integration of independent verification organizations. One result of the DoD STEP recommendations is that DoD Instruction 5000.2 states "Both developmental and operational testers shall be involved early..." Army STEP has further defined procedures for close coordination between verifiers, users, and developers. The new DA Pam 73-1 Volume 6, *Software Test and Evaluation Guidelines* [27] describes how software testing and evaluation activities relate to each phase of the software life cycle. The adoption of all or portions of DA Pam 73-1 into DoD instructions and directives could reinforce and more precisely define the communications that should occur among verifiers, developers, and the customer.

The Air Force Standard Systems Center (SSC) at Gunter Air Force Base in Alabama, takes customer involvement seriously. Some of their standard information systems development work is contracted to local software firms. However, as dictated by the terms of the contracts, Government personnel are participating members of contractor developer and verifier teams. While this has caused a few unusual and difficult situations, the overall strategy appears to work. SSC anticipates that the result of these contracts will be well-defined requirements and design, better quality software, and systems that are more easily maintained after acceptance [28]. This SSC practice could be a model for DoD contracted software development.

DoD Standard 2167A [19] clearly requires traceable and testable requirements. Traceable requirements are defined and formulated such that direct cross-referencing exists among requirements, design specifications, code, and test cases. Traceability also implies that each requirement can be implemented in both design and code. A requirement is testable if and only if it is written so that developers and verifiers can prepare specific test cases that can clearly confirm satisfaction of the specific requirement.

At both IBM FSD Rockville and IBM FSC Houston, developers and verifiers work together to ensure that requirements are both traceable and testable when defined. In fact, test engineers for the Advanced Automation System (AAS) project built and use a software tool which automatically maintains the relationships between requirements and test cases. This tool is essentially a specialized database management system that assists developers and verifiers in test management and configuration control. While such tools help to manage the relationships and maintain the consistency of the software products once developed, they cannot replace the difficult work required beforehand to ensure traceability and testability. As practiced by IBM, success in this work is a direct result of close communications and coordination among the users, developers, and verifiers. IBM describes the relationship between its developers and verifiers as *friendly-adversarial*. This means that both groups work together with the customer toward a mutual understanding of the product requirements and design and the early identification of errors. Finding and preventing errors are considered primary job responsibilities for both developers and verifiers. At the same time, each group independently designs its respective test plans and cases for later verification and validation.

4. Formally Inspect All Software Products

Empirical evidence indicates that dynamic software testing (i.e. execution of code) alone cannot ensure quality and is not cost-effective. Yet, dynamic testing is essential to confirm software quality. Dynamic testing should be planned at the same time that requirements are analyzed and defined, and then executed systematically as planned. However, if quality is the objective, then verification cannot wait for code. Early detection techniques must be applied extensively to all software products so that dynamic testing can be a cost-efficient and graceful confirmation of functionality and quality.

The analyses necessary to define implementable, traceable, and testable requirements helps to avoid errors. However, one of the most effective early detection methods is the *formal inspection* [29, 30] (also referred to as the *Fagan Inspection* [31]). Developed by Dr. Michael Fagan in 1976, the formal inspection

is a general–purpose verification method. It is product–independent and can be employed to identify errors in requirements, design, documentation, test plans, or code. Thus, it has the potential to identify and permit removal of errors very early during the software life cycle. In fact, IBM FSC Houston reports that their application of formal inspections accounts for the identification of 80% of the errors in the U.S Space Shuttle flight software. Even so, the acceptance of formal inspection into general practical use has been slow for several possible reasons. The technique has a reputation for being "low–tech." It requires a fair amount of intensive, detailed work [15], although it does appear that automated tools could enhance some of its procedures. The availability of good empirical data verifying its cost–effectiveness has not been available until the last several years. Even now, published results are not prevalent. At least one software corporation considers its use of formal inspection procedures as a competitive advantage, and thus, declined to divulge their procedures [14].

A formal inspection is essentially a testing technique in which a software product is formally examined by a team of experts. These experts include the author of the product and several of his/her peers. Depending upon the product, the team may also include a customer representative and a verifier. The primary objective of the team is to find as many errors as possible. In such a situation, finding errors must be considered in a positive sense, i.e. the team intensively scrutinizes the product (code or documentation), *not* the author's abilities. The team's responsibility is to help the author(s) by identifying mistakes, thus preventing their entry into the next phase of the life cycle. This is done by paraphrasing lines or portions of the software product at a slightly higher level of abstraction or from a different perspective (such as from the verifier's view). The error detection efficiency of this process results from its formality and intensity. The procedures are defined and repeatable. Standard checklists ensure that common mistakes are not overlooked.

As practiced by IBM FSC Houston, the formal inspection is the cornerstone of software verification and process improvement. All software products must submit to and pass a formal inspection prior to acceptance into configuration control or submission for execution testing. Each product is examined by an inspection team tailored to that product. For example, the inspection team for a requirements definition document will include the customer, a requirements analyst, a verifier, a programmer, as well as the author. The inspection team for the independent verifier's test plan will include the customer, a requirements analyst, and several verifiers. The inspection team for the developer's test plans will include several requirements analysts and programmer's. These particular examples illustrate how the tailoring of inspection teams establishes a cooperative yet independent relationship between developer and verifier. Each inspection

team includes a senior peer who acts as the moderator. He must foster cooperation and focus on the objective — to find errors. Management strongly supports formal inspections, but does not participate in them. This ensures that inspection results are used to rate the effectiveness of the technique and not the performance of individuals. Inspection teams record errors identified, and subsequently, require authors to correct them. The requirement for re-inspection depends upon the severity and number of errors recorded. Error statistics from inspections of all products and phases of the software life cycle are collected to measure process effectiveness.

The advantages of formal inspections can be significant. Since formal inspections are reported to detect 80% of all errors, subsequent dynamic testing of code becomes more efficient. Fewer execution failures cause fewer interruptions. This translates to additional time for more thorough testing, and possibly less time required for regression testing. Formal inspections and dynamic testing techniques compliment each other. Each can detect flaws that the other cannot [14, 15]. Execution testing detects faults and failures, the manifestation of errors. On the other hand, formal inspections detect the errors which potentially cause faults and failures.

Besides enabling early and effective error detection for a range of software products, there are several indirect advantages of formal inspections. At IBM they encourage the *friendly-adversarial* relationship between developers and verifiers through teamwork, cooperation, distributed risk, and consensus. Developers, verifiers, and the customer tend to focus effort on the most important aspects of software development — requirements and design. Time and cost required for testing and repair are diminished [32]. Formal inspections foster understandability and standardization in all software products. They provide excellent on-the-job-training for all participants since they teach technical standards and organizational culture [12]. Furthermore, formal inspections proliferate good ideas and eliminate bad approaches [31].

Formal inspections can require from 15% to 25% of total development time [32], so DoD developers may be reluctant to expend limited resources to support them. However, the resources necessary to implement them are not as great as those necessary to find and fix errors later [10, 11, 12, 13, 14, 15]. A cost-benefit analysis at Shell Research [12] reported an average 30 hours of repair and maintenance time saved for every hour of inspection time invested. Bell-Northern Research [15] reported a 33:1 return. Other organizations have reported more conservative returns of 2:1, 6:1, and 10:1 [14]. Note that these estimates are based entirely on direct costs. They do not include other possible

savings from indirect costs related to customer confidence and the avoidance of the consequences of operational failure.

DoD Instruction 5000.2 specifically states that DoD contractors should practice walk-throughs, inspections, or reviews of requirements, documents, design, and code [18]. Of these, the formal inspection is the most painstaking and work-intensive technique. Reviews and walk-throughs are also useful, but they have other goals, so they are less effective for detecting errors [32]. While use of formal inspections has demonstrated the production of high software quality at overall reduced cost and time [31], the earlier investment in cost and time can easily drive a decision not to employ them. This is apparently because the consequences of quality are not as visible as those of cost and schedule in the early phases of the life cycle. We will discuss more about this later. The fact is that the resources expended to implement formal inspections can pay for themselves in a short time by removing more expensive testing and maintenance costs.

Of the techniques we discuss in this report, formal inspection appears to be the basis for the others. This technique stimulates, coordinates, and checks the developer/verifier coordinated requirements definition process. It does this by promoting teamwork and shared responsibility for quality. It also produces early defect data necessary to measure, feed, and guide process improvement. In addition to IBM, several other companies have described their experiences with the successful introduction of formal inspections. A few offer tips for overcoming the difficulties of instituting them [12, 15]. We summarize these tips as follows:

(1) There must exist a *belief* that formal inspections will be effective. Dynamic code testing will always seem to be faster and more effective, but this is not true [15]. To change this mind-set will require an active campaign to *sell* the efficiency of formal inspections to all levels of the organization. Circulating reports of success and training programs can accomplish this.

(2) Everyone must clearly understand formal inspection procedures. They are not informal. They are not cursory reviews, audits, or walk-throughs. Formal inspections are manual, intensive, detailed, and painstaking. Education and training are the best ways to prepare.

(3) It is essential to have management support. Management must be decisive and committed to the belief that formal inspections will pay off. This requires that the cost of formal inspections be quantified, and resources be allocated to accommodate them into the schedule. Also, organizations must anticipate adjustments to the procedures as they adapt inspections to their own local environments.

(4) Early successes are critical, but also difficult to achieve. Start by inspecting only one or two types of documents (for example, requirements definition). The first products inspected may be riddled with defects. Early inspections can easily become muddled in details until problems with standards and procedures are resolved. Therefore, good moderators who can maintain group momentum are essential in the early stages.

(5) Keep detailed statistics on defect identification and associated actions. This data feeds process improvement and provides clear evidence of effectiveness. It will confirm belief in the process and strengthen commitment to it.

(6) The best training for inspections is on–the–job training. However, continued formal training of inspection team moderators is particularly important. Otherwise, as the effectiveness of the process becomes apparent to all, the amount of materials and the number of required inspections can overwhelm the best–planned schedules.

(7) The local development process must be well–defined and understood by the participants. Otherwise, formal inspections will be ineffective. [31]

5. Use Error Data to Guide Defect Prevention and Process Improvement

Early identification and correction of errors is critical to software product correctness and quality. Correcting errors in software is a fix, but not a solution. Software errors are often the symptoms of a more fundamental process defect. Typical process defects might be failure to follow a standard practice, misunderstanding of a critical process step, or lack of adequate training. In the software factory paradigm [5], the software development process is a special manufacturing system to which many traditional quality control principles apply. The developers and verifiers themselves (owners of the process) use error data to measure and improve the process, until it reaches a repeatable, predictable steady state. Based on principles of Total Quality Management (TQM), formal process improvement implements error prevention by removing the causes of errors within the development process and the causes of not finding these errors earlier in the detection processes.

IBM FSC Houston practices process improvement. Developers and verifiers form small process evaluation teams (in TQM terms, process action teams) to analyze defects and identify their causes. These teams also determine how to remove defect cause, and subsequently implement required process changes. The

effectiveness of these teams is rooted in TQM. Team members are the analysts, programmers, and verifiers whose primary daily responsibilities are software development. Therefore, those who execute the development process also execute process improvement. The key to success is total management support and encouragement. The responsibility to analyze and execute rests with the developers and verifiers. The responsibility to allocate resources and make decisions that support process improvement rests with the managers.

The practice of process improvement has a number of positive outcomes. A process that is partially or totally undefined will have to be defined in writing in order to subject it to process improvement. This further stabilizes the process and tends to make it repeatable. The continued practice of improvement defines clear procedures for change and enables gradual technology insertion. There is less resistance to new technology, because the implementors of change are the same people who suggest it. At the very least, there will be a willingness to try new ideas.

Another important advantage of process improvement is its built-in on-the-job training environment. Membership on a process evaluation team is an excellent first assignment for new personnel. This responsibility encourages immediate participation and teamwork, teaches the process definition and its change procedures, and stimulates creative thinking in the form of improvements. New personnel are generally enthusiastic about contributing and bring fresh ideas into the organization.

The long-term benefits of process improvement are also significant. The procedures make the development process self-correcting. Therefore, over time, the number of errors inserted during each phase of software development decreases. This translates to a decrease in re-work and greater efficiency for all sub-processes. For example, verifiers may experience fewer problems during dynamic testing, because fewer (if any) serious errors exist that could interrupt, delay, or prevent test completion.

Process improvement techniques are not new. As previously mentioned, they are essentially TQM techniques applied to the software development process. The SEI Capability Maturity Model (CMM) for the software development process contains procedures for process improvement [33]. Furthermore, the SEI Quality Subgroup of the Software Metrics Definition Working Group and the Software Process Measurement Project Team have developed a draft framework for documenting software problems [34]. Such a standard collection mechanism can ultimately measure progress, enable estimations, and guide process improvement. Dr. Michael Fagan, who originally developed formal inspection procedures [29], now trains developers and managers to improve their software productivity and quality with a three-step process — formal process definition,

formal inspection of all software products, and continuous process improvement (through defect cause removal) [31].

DoD could achieve significant efficiencies in both its software development process and its software quality by applying process improvement. Current DoD emphasis on TQM will help to encourage the incorporation of software process improvement techniques. Past and current DoD practices rely on developers to learn from their mistakes [2, vol. 1]. However, as this learning is lost through personnel and project turnover, organizations are doomed to repeat their mistakes. Formal process improvement eliminates the causes of error, documents learning, and hardens solutions against erosion by time. Process improvement has been proven effective in several software development environments [5]. It can move a software development process from one that is highly reactive and ad hoc to one that is statistically stable, predictable, repeatable, and efficient.

6. Actively Motivate a Quality Attitude

IBM FSC Houston's empirical evidence is convincing and its recipe is simple: Quality software results from basic disciplined processes, supported by a genuine quality attitude [10]. But, the greatest impediment to implementation facing DoD appears to be the lack of a software quality attitude. This is not to say that DoD does not care about producing quality software. Rather, cost, schedule, and the product take greater priority because they have the highest visibility during the early phases of software development. Until the testing phase, quality is essentially an unknown or invisible. However, by this time, cost and schedule drive the software through some ad hoc forms of testing (because, if the software is riddled with defects, dynamic testing will be extremely time-consuming, costly, and difficult). Once the software is in customer hands, poor quality will be highly visible because the cost to fix it is high on the exponential scale (independent of the cost of damage to customer confidence!).

The implementation of early defect identification, defect prevention, and process improvement in large software development environments is apparently difficult [12, 15]. They are perceived as labor-intensive work with little short-term payoff. Investment returns are not realized until late in the software life cycle — during system testing, operation and maintenance. Within DoD this is a particularly critical problem. DoD program managers are generally not rewarded for delivering systems with good operation and maintenance records. These managers usually control the development phases of the program. Thus, costs, schedules, and product functionality drive their decisions. But, the success of defect identification, defect preventions, and process improvement demands an

early and continuous commitment. This means that managers must be willing to adjust cost, schedule, and resource allocation to support this process.

We have made the case that DoD knows how to produce quality in software. Then, why have current DoD instructions and directives been unable to make this happen? Basic processes are easy to mandate; belief in their long-term payoff is not. Unfortunately, without the *belief* that they work and an associated *commitment* at all levels to support them, it is possible to merely satisfy the requirements, and thus, render them ineffective [10]. The solution is to make quality more visible than cost, schedule, or product. The effects of poor quality must be quantified early in a project. Program managers must compare cost of these effects plus the cost of re-work against the cost to implement formal inspections and process improvement. To motivate this, quality must be made equivalent to cost and schedule for judging program success.

IBM FSC Houston has a well-defined motive for producing quality software — astronaut lives depend upon it! This forces quality to be the priority objective early and continuously. However, such a motive for quality is not common to all software. MIS software is generally not safety critical. Command and control software failures may have some safety implications, but these are generally indirect — the result of human decisions based on faulty information. However, DoD might learn from one Japanese corporation — Fujitsu. In 1965, Fujitsu accepted a contract stating that "if a customer receives any damage due to malfunction of Fujitsu equipment, Fujitsu will compensate any damage unlimitedly." [35] While not a software contract, this example does illustrate that a quality guarantee based on consequence can create high visibility. This may be one way to help stimulate the use of proven quality techniques. Unfortunately, most (if not all) U.S. software firms take quite the opposite approach. Rather than guarantee compensation for poor quality, they issue statements of limited liability for the software they create. The widespread issuance of such statements indicates a low level of confidence by software makers in the quality of their own products. Furthermore, this may also indicate a general unwillingness to sign such contracts.

In contrast to IBM FSC Houston, quality in DoD MIS/C2 software is not clearly defined, and therefore, is rarely a motive. However, there are ways to establish quality objectives. In 1980, as part of a joint Air Force Rome Labs and AIRMICS project, General Electric initially developed a method to plan, insert, and measure quality requirements in software. Subsequent contractors for Rome Labs have improved this method. In 1985, Boeing Aerospace Company finalized a detailed framework for establishing software quality objectives and requirements from organizational needs [36]. The resulting reports provide guidance for measuring and evaluating requirements satisfaction throughout the

life cycle. Included are evaluation checklists that are similar to those used by IBM in their formal inspections. The ideas in this framework have appeared in other DoD manuals, but only as guidelines. For example, they are contained in *Draft Army Technical Bulletin 18-102-2 (1985), and U.S. Army Information Systems Software Center Pamphlet 25-1 (1990), Software Quality Engineering Handbook* [37]. In contrast to well-published results of formal inspections, Rome Labs' Software Quality Framework has received less visibility. However, the adoption and extensive use of very similar quality methods by NEC Corporation [38] and Metriqs, Inc. is evidence of its potential value [39].

DoD appears to have a more positive, pro-active attitude toward software quality. We believe that the establishment and support of the Army Software Test and Evaluation Panel (Army STEP) is very significant. The mandate to employ the Army STEP Metrics may be the first high-level action taken to implement software quality practices in the military. The Army's serious attention to metrics represents a significant shift by Army management toward software development as an engineering discipline. This also indicates management willingness to expend the resources for early measurements to gain control of quality. We believe that DoD should take this opportunity to encourage, support, and motivate these efforts. Management supported metrics are a positive first step. However, these should not be collected for the sake of project management alone. Quality requirements should be formulated during the requirements definition phase. This could be accomplished using the Rome Labs Software Quality Framework. Once established, quality requirements should be measured through standard metrics and checked in detail through formal inspections using the checklists associated with the requirements.

7. Introduce CASE Tools to Support Well-Defined Sub-Processes

Because Computer Assisted Software Engineering (CASE) tools apply to all aspects of software development including testing, and because we have expanded the view of testing to encompass the entire life cycle, we discuss here the potential impact of CASE technology on DoD, and its relationship to the techniques presented thus far.

The activities of software engineering which most impact software quality (coordinated planning, formal inspection, configuration control, verification testing, and process improvement) should be repeatable, and yet, adjustable if they are to be effective. Often, the most efficient means of standardizing a process and making it repeatable is by automating it. Computer Assisted Software Engineering (CASE) tools do this for software development and testing processes. However, automating any process first requires that its procedures be

well–defined, well–understood, and practiced. CASE technology cannot impose a methodology on an ad hoc software development environment [40]. Applying CASE technology in order to structure a manual process that is not working simply exacerbates a poor process. For example, without a well–defined, well–understood manual procedure for configuration management, an organization should not expect to effectively control software configuration using a CASE tool. Automating a bad process only escalates a bad situation. Initially, DoD organizations should use CASE tools only for those processes which are well–defined and practiced.

There are many examples of organizations which have adopted CASE tools only to abandon them because the anticipated improvements never materialized. One reason for this was described above. Another reason is an underestimation of the CASE tool training requirements [41]. Because CASE tools support methods, but do not impose them, an organization must recognize the difference between learning the tool and learning the method the tool supports (especially if the supported method is not currently practiced!). There are learning curves associated with each [41]. If the method is understood and practiced, then only the tool presents a learning shortfall. However, if the tool supports a new method unfamiliar to the developers, then two shortfalls exist. The training and time to overcome such may add significantly to CASE tool investment. The compound training requirement (for both method and tool) will also extend the time required to show a return on the investment. DoD should sensitize management to the large investment required to train personnel in both the new tools and, as necessary, the associated new methods.

The adoption of CASE technology within DoD should be an evolutionary process that begins small and grows gradually. Controlled institution of CASE tools has a greater potential for immediate success and visible investment return than a massive, overwhelming introduction. The adoption of standard software process metrics gives DoD the means to establish the value of CASE tools. DoD should not force its managers to overreact. Rather, it should make the time and financial resources available for managers to adequately train and methodically insert standard CASE tools into practical use. Both CASE tool users and management must restrain expectations of immediate results. They must anticipate the learning curve(s), measure progress, and continue with process improvement to insert additional CASE technology.

Our observations of and discussions with IBM FSC Houston strongly support this approach to introduction of CASE tools. Developers at IBM FSC Houston have produced and continue to produce high quality software using manual processes (supported by non–integrated databases and word processing tools).

CASE tools are just now being considered. However, deployment of CASE tools will be closely monitored and controlled through formal process improvement [42]. This will ensure that the adoption of tools will properly enhance their own methods. CASE technology will probably cause changes (improvements) in their current methods, but only through the formal improvement process. We highly recommend that DoD consider process improvement as the technology insertion mechanism for instituting CASE tools.

8. Conclusions and Recommendations

DoD knows how to produce quality software. The Army has recognized the critical relationship between measurement and quality control, and is now implementing mechanisms that can decrease the cost and increase the quality of software production. Software testing is related to these mechanisms as a key data-gathering technique. However, quality cannot be tested into software. Quality must be designed through engineering. Engineering requires an expanded view of testing to maximize the effectiveness and reduce the cost of verification and acceptance testing. DoD testing activities should begin during requirements definition and should influence the entire software development life cycle.

The general principles of engineering applied by IBM and other commercial companies to produce quality MIS, C2, or MSCR software are the same — modeling, standardization, measurement, and process control. However, the empirical evidence has come from environments in which the developing/ testing/maintaining organization, the software, and the customer have been relatively constant for a long period of time. For example, IBM FSC Houston has developed and tested the Space Shuttle flight and ground software for NASA for over 15 years, adequate time to stabilize their development process. However, real cost savings have been reported even in the case these procedures were initiated during the verification phase of a project [31]. Nonetheless, our recommendations should be viewed in the context of the DoD environment. There exists a variety of contractors, customers, software, and relationships among them. Detailed standardization of software engineering activities would be too restrictive and probably counter-productive. Instead, DoD guidelines should be standardized locally through detailed, written procedures.

On the basis of our discussions and conclusions, we recommend the following actions:

(1) Actively motivate a software quality attitude in DoD and government contractors. Implement by adopting and training a process (such as the Air Force

Rome Labs Software Quality Framework [36]) to establish quality objectives within software requirements. This will increase the importance and visibility of software quality by providing clear motivation. Thus, quality will balance with product functionality, cost, and schedule.

(2) Make the use of formal inspections standard for all software products (requirements, documentation, design, code, test plans, tests). For this, DoD can provide high-level guidance in the form of generic checklists. Examples are already published in the military [37]. From these generalized lists, more specific local standards can be developed.

(3) Users, developers, and verifiers should participate jointly in requirements analysis and definition. They should be mutually responsible for ensuring that requirements are clearly documented, implementable, and testable. They should also define quality objectives. Their goal should be to produce clear documentation which directly cross-references test cases to requirements and code. Employ formal inspections to enable joint participation throughout the software life cycle. At the same time, the developer (for development testing) and verifier (for operational testing) should independently plan, design, develop, inspect, execute, and analyze the results of software tests.

(4) Standardize the measurement and documentation of errors throughout the software life cycle. Establish a formal defect prevention program which empowers developers and verifiers to analyze the causes of error and enact improvements to the development process that will prevent future error insertion and enhance detection processes.

(5) Encourage evolutionary introduction of standard CASE tools for all software life cycle sub-processes including development, testing, and maintenance. Each organization should purchase, train, and employ only those tools for which corresponding processes have been defined. Allow adequate time to learn and gain experience with both the method and the tool. Purchase and integrate new tools only when users understand the supported process, recognize the benefit of automating it, and are ready to be trained. Use formal process improvement as the technology transfer mechanism.

A recent issue of *Army Research, Development & Acquisition Bulletin*, contained a short article reprinted from a 42-year-old issue of the *Proceedings of the Institute of Radio Engineers* [43]. The article, entitled "Quality in Engineering," emphasizes the importance of quality in the engineering of electronic equipment. The following is an extract of this article:

"Quality is never an accident. It is always the result of high intentions, sincere effort, intelligent direction, and skillful execution. Quality cannot be inspected in ... quality must be designed in!"

This passage is as true today of software as it was then of hardware. Quality cannot be tested into software. Furthermore, inspection alone is ineffective. Quality results from belief in and commitment to quality objectives, well-defined processes, continuous measurement supported by formal inspection, and process improvement.

REFERENCES

1. Buckley, F.J. and Poston, R., "Software Quality Assurance," *IEEE Transactions on Software Engineering*, vol. SE-10, no. 1, Jan 84, pp. 36–41.

2. DoD Software Test and Evaluation Project (STEP) Volumes 1–6, 1983.
   (1) Final Report and Recommendations
   (2) Software Test and Evaluation: State-of-the-Art Overview
   (3) Software Test and Evaluation: Current Defense Practices Overview
   (4) Transcript of STEP Workshop, Mar 82
   (5) Report of Expert Panel on Software Test and Evaluation
   (6) Tactical Computer System Applicability Study

3. Gelperin, D., and Hetzel, B., "The Growth of Software Testing," *Communications of the ACM*, vol. 31, no. 6, Jun 88, pp. 687–695.

4. *Software Maintenance Technology Reference Guide*, Software Maintenance News, Inc., 1992.

5. Humphrey, W.S., "Software and the Factory Paradigm," *Software Engineering Journal*, Sep 91, pp. 370–376.

6. Fernstrom, C., Kjell-Hakan, N., and Ohlsson, L., "Software Factory Principles, Architecture, and Experiments," *IEEE Software*, Mar 92, pp. 36–44.

7. Basili, V.R., and Musa, J.D., "The Future Engineering of Software: A Management Perspective," *IEEE Computer*, Sep 91, pp. 90–96.

8. Dunham, J.R., "V&V in the Next Decade," *IEEE Software*, May 89, pp. 47–53.

9. Martin, R.J., "The Application of the Principles of Total Quality Control to Mission-Critical Software," ISYE 6301, 28 Nov 84.

10. IBM FSC Houston Software Process Showcase, IBM Federal Systems Company, Houston, TX, 2–3 Apr 92.

11. Dichter, C.R., "Two Sets of Eyes: How Code Inspections Improve Software Quality and Save Money," *Unix Review*, vol. 10, no. 1, Jan 92, pp. 19–23.

12. Doolan, E.P. (Shell Research), "Experience with Fagan's Inspection Method," *Software — Practice and Experience*, vol. 22(2), Feb 92, pp. 173–182.

13. Royce, W., "Pragmatic Quality Metrics for Evolutionary Software Development Models," TRW-TS-91-01, TRW Systems Engineering & Development Division, Jan 91.

14. Ackerman, A.F., "Software Inspections: An Effective Verification Process," *IEEE Software*, May 89, pp. 31–36.

15. Russell, G.W. (Bell–Northern Research), "Experiences with Inspections in Ultralarge–Scale Developments," *IEEE Software*, Jan 91, pp. 25–31.

16. Blakely, F.W., and Boles, M.E., "A Case Study of Code Inspections," *Hewlett–Packard Journal*, vol. 42, Oct 91, pp. 58–63.

17. Cavano, J.P., and LaMonica, F.S., "Quality Assurances In Future Development Environments," *IEEE Software*, Sep 87, pp. 26–33.

18. *Defense Acquisition Management Policies and Procedures*, Department of Defense Instruction 5000.2, 23 Feb 91.

19. *Defense System Software Development*, DOD–STD–2167A, 29 Feb 88.

20. *Defense System Software Quality Management Program*, MIL–STD–2168A (Draft), 6 Aug 91.

21. Final Report (Draft) of the Software Test and Evaluation Panel (STEP) by the Standards and Regulations Implementation Team (SRIT), 29 Jan 92.

22. Demming, W.E., "Out of the Crisis," MIT Center for Advanced Engineering Study, Cambridge, MA, 1982.

23. Graham, D.R., "Test is a Four–Letter Word: The Psychology of Defects and Detection," *CrossTalk: The Journal of Defense Software Engineering*, No. 35, Aug 92, pp. 2–7.

24. *ANSI/IEEE Standard Glossary of Software Engineering Terminology*, ANSI/IEEE Std 729–1983, IEEE, 1983.

25. DoD Software Test and Evaluation Project (STEP), Software Test and Evaluation Manual Volumes 1–3.
(1) Guidelines for the Treatment of Software Test and Evaluation Master Plans, Oct 85.
(2) Guidelines for Software Test and Evaluation in DoD, 25 Feb 87.
(3) Good Examples of Software Testing in DoD, 1 Oct 86.

26. Williamson, M., "Beyond Rube Goldberg," *CIO*, Mar 91, pp. 54–59.

27. *Software Test and Evaluation Guidelines*, DA Pamhlet 73–1 (Draft), vol. 6, 15 Jun 92.

28. Informal discussions with Air Force Standard Systems Center, Gunter Air Force Base, AL, Mar 92.

29. Fagan, M.E., "Design and Code Inspections to Reduce Errors in Program Development," *IBM Systems Journal*, vol. 15, no. 3, 1976, pp. 182–211.

30. Fagan, M.E., "Advances in Software Inspections," *IEEE Transactions on Software Engineering*, Jul 86, pp. 744–751.

31. Fagan, M., "Productivity Improvement through Defect-Free Software Development – An Executive Overview," Michael Fagan Associates, slide presentation by Dr. Fagan to Army Strategic Defense Command, Huntsville, AL, 31 Aug 92.

32. Brykczynski, B., "Software Inspections," *SDIO/SDA Software Engineering Newsletter*, vol. 1, no. 2, May 92.

33. Paulk, M.C., Curtis, B., and Chrissis, M.B., "Capability Maturity Model for Software," CMU/SEI-91-TR-24 and ESD-91-TR-24, Software Engineering Institute, Carnegie-Mellon University, Aug 91.

34. Florac, W.A., "Software Quality Measurement: A Framework for Counting Problems, Failures, and Faults (Draft)," The Quality Subgroup of the Software Metrics Definition Working Group and the Software Process Measurement Project Team, Software Engineering Institute, Carnegie-Mellon University, May 92.

35. Ochi, Hiroyuki, "Japanese Software Quality", Briefing notes delivered at 4th Annual Software Technology Conference, Salt Lake City, UT, 13–17 Apr 92.

36. Bowen, T.P., Wigle, G.B., and Tsai J.T., "Specification of Software Quality Attributes: Software Quality Evaluation Guidebook," RADC-TR-85-37 Volume 3 of 3, Rome Air Development Center, Feb 85.

37. *Software Quality Engineering Handbook*, Draft U.S. Army Technical Bulletin (TB) 18-102-2, Army Automation, 25 Mar 85 (also U.S. Army Information Systems Software Center Pamphlet 25-1, Feb 90).

38. Sunazuka, T., Azuma, M., and Yamagishi, N., "Software Quality Assessment Technology," *Proceedings of the 8th International Conference on Software Engineering*, IEEE, 1985.

39. Murine, G.E., "Integrating Software Quality Metrics with Software QA," *Quality Progress*, Nov 88.

40. Jones, C., "CASE'S Missing Elements," *IEEE Spectrum*, Jun 92, pp. 38–41.

41. Kemerer, C.F., "How the Learning Curve Affects CASE Tool Adoption," *IEEE Software*, May 92, pp. 23–28.

42. Lee, Earl, Senior Test Engineer for IBM Federal Systems Company Houston, informal discussions in Jun 92.

43. Dumont, A.B., "Quality in Engineering," *Proceedings of the Institute of Radio Engineers*, Nov 50.

BIBLIOGRAPHY

1. Bergman, M., "The Evolution of Software Testing Automation," *Proceedings of the 8th International Conference on Testing Computer Software*, International Test and Evaluation Association, 17–20 Jun 91, pp. 1–10.

2. Deimel, L.E., *Scenes of Software Inspections*, CMU/SEI–91–EM–5, Software Engineering Institute, Carnegie–Mellon University, May 91.

3. DeMillo, R.A., McCracken, W.M., Martin, R.J., and Passafiume, J.F., *Software Testing and Evaluation*, Benjamin–Cummings Publishing Co., 1987.

4. Dyer, M., *The Cleanroom Approach to Quality Software Development*, Wiley and Sons, Inc., 1992.

5. Fear, H.S., "Creating A Test Process," *Proceedings of the 8th International Conference on Testing Computer Software*, International Test and Evaluation Association, 17–20 Jun 91, pp. 35–43.

6. Fewster, M.A., "The Managing Director Wants 100% Automated Testing: A Case History," *Proceedings of the 8th International Conference on Testing Computer Software*, International Test and Evaluation Association, 17–20 Jun 91, pp. 59–70.

7. Kelly, J.C., Sherif, J.S., and Hops, J., "An Analysis Of Defect Densities Found During Software Inspections," *Journal of Systems and Software*, vol. 17, Feb 92, pp. 111–117.

8. Martin, R.J., "The Challenge of Software Engineering Project Management: Ten Years Later", SERC–TR–73–P, Software Engineering Research Center, Purdue University, May 90.

9. Mays, R.G., Jones, C.L., Holloway, G.J., and Studinski, D.P., "Experiences with Defect Prevention," *IBM Systems Journal*, vol. 29, no. 1, 1990, pp. 4–32.

10. O'Neill, D., "What Is the Standard of Excellence? New Views of Mature Ideas on Software Quality Productivity," *IEEE Software*, May 91, pp. 109–111.

11. Parnas, D.L., van Schouwen, A.J., and Kwan, S.P., "Evaluation of Safety–Critical Software," *Communications of the ACM*, vol. 33, no. 6, Jun 90, pp. 636–648.

12. Schneck, P.A., "Virtually Defect–Free Code as a Direct Result of a Well–Defined Comprehensive Testing Method," *Proceedings of 8th International Conference on Testing Computer Software*, International Test and Evaluation Association, 17–20 Jun 91, pp. 133–141.

13. Thebaut, S.M., and Martin, R.J., "SERC Affiliate Current Practices," SERC-TR-32-P, Software Engineering Research Center, Purdue University, 1989.

14. *Draft DoD Software Technology Strategy*, DoD Software Technology Working Group, Dec 91.

15. Presentations of the 1990 Multiservice Software Test and Evaluation Symposium, Reston, VA, 30 Oct - 1 Nov 90.

16. *Quality Program*, U.S. Army Technical Bulletin (TB) 18-102, Army Automation, Mar 84.

17. Report of Task Force on Military Software, Defense Science Board, 1 Jul 87.

18. Software Test Tools Report, U.S. Air Force Software Technology Support Center (STSC), Hill AFB, Utah, 1991.

19. *Testing of Computer Software Systems*, U.S. Army Technical Bulletin (TB) 18-104, Army Automation, 20 Aug 82.

20. U.S. Army Software T&E Standards and Regulations Reference Guide (Draft).

21. *Using Test Data Generators to Reduce Software Development Costs*, U.S. Army Technical Bulletin (TB) 18-22, Management Information Systems, 10 May 74.

APPENDIX – List of Acronyms

AAS	Advanced Automation System
AIRMICS	Army Institute for Research in Management Information, Communications, and Computer Sciences
ANSI	American National Standards Institute
Army STEP	Army Software Test and Evaluation Panel
C2	Command and Control
C3I	Command, Control, Communications, and Intelligence
CASE	Computer Assisted Software Engineering
CMM	Capability Maturity Model
DoD	Department of Defense
DoD STEP	DoD Software Test and Evaluation Project
FSC	Federal Systems Company
FSD	Federal Sector Division
IBM	International Business Machines Corporation
IEEE	Institute of Electrical and Electronics Engineers
IV&V	Independent Verification and Validation
MIS	Management Information System
MSCR	Materiel System Computer Resource
NASA	National Aeronautic and Space Administration
OASD	Office of the Assistant Secretary of Defense
SEI	Software Engineering Institute
SSC	Standard Systems Center
TQM	Total Quality Management