The economics of software quality assurance

by DAVID S. ALBERTS
The Mitre Corporation
McLean, Virginia

ABSTRACT

This paper presents an examination into the economics of software quality assurance. An analysis of the software life-cycle is performed to determine where in the cycle the application of quality assurance techniques would be most beneficial. The number and types of errors occurring at various phases of the software life-cycle are estimated. A variety of approaches in increasing software quality (including Structured Programming, Top Down Design, Programmer Management Techniques and Automated Tools) are reviewed and their potential impact on quality and costs are examined.

INTRODUCTION

Current realities of large scale computer systems have provided the impetus to undertake this examination into the need for and potential of a quality assurance program. Proponents of quality assurance claim that significant savings in both cost and time can be achieved in addition to improved system performance if a quality assurance program is implemented. The purpose of this paper is to examine these claims by addressing the economics of software quality assurance. Software rather than hardware is the subject of the analysis since the costs of software have far outstripped the costs of hardware and the trend seems to be continuing in this direction (Figure 1).

The stakes involved are high. Estimates of recent Air Force annual expenditures on software are over $1 Billion.[5,6] WWMCCS alone was estimated to involve $¾ Billion for software (about 10 times its hardware costs),[2] while major software systems also run into hundreds of millions of dollars (IBM OS/360 $200M,[11] SAGE $250M[11] and NASA manned space program $1B[12]). Indirect costs must be added to these huge sums and are by no means trivial in of themselves. For example, software delays often cause delays in reaching the operational phase of a system's life. A 6-month delay (considered almost on-time) translates into a $100M loss of services, based upon a projected 7 year operational life and a $1.4 billion project.

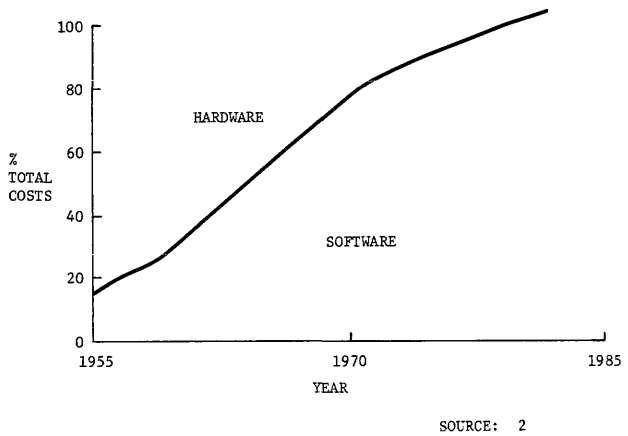

Figure 1—Importance of controlling the cost/effectiveness of S/W

The actions which could be undertaken under the umbrella of a quality assurance plan are quite diverse; so diverse that it is difficult to separate these actions from project management. However, quality assurance is only one aspect of project management. First this analysis addresses the software life cycle and the relative cost on each portion of the cycle. Next productivity is considered insofar as the reduction in errors in each portion of the cycle impacts cost. Finally a variety of methods, techniques and tools which can directly affect the error rate/severity experienced will be examined.

Two questions drive this analysis. First, can QA work? and second, Is it worth it? This paper brings together the experiences and thoughts currently in circulation and forms these into an analysis of the issues involved and presents composite estimates of the potential target of quality assurance (cost of error) and the reported experience of quality assurance programs and methods currently available. Because of the difficulty in separating QA methods from project management and the absence of good cost accounting standards, the costs of a quality assurance program are not explicitly treated in this paper.

PHASES OF THE SOFTWARE LIFE CYCLE

To ensure a complete and systematic review of the potential for a QA plan, each aspect of the software "life cycle" will be examined to determine at what point QA can support substantial improvements. Both direct and indirect costs will be considered.

Direct costs are those associated with the actual performance of the particular phase of the software life cycle under consideration while indirect costs include schedule slippages, system degradation, and errors which contribute or add to the cost of subsequent stages in the process. The software life cycle can be broken down into four phases: Conceptual, Requirement, Development and Operations. While other authors have broken this cycle down somewhat differently, either by separating *Development* into two or more separate phases or by extending *Requirements* to include part of the development phase, the categorization shown here more closely corresponds to distinct levels of effort or expenditures.

After a brief qualitative discussion of the potential role of QA in each of the four phases of the software life cycle, the amount of time and relative costs incurred in the performance of each of these phases will be reviewed. Available data on contributions of errors to cost and delay is examined later.

The conceptual phase

This phase begins with the recognition of a need for the system. The feasibility and general worthiness of a proposed system is addressed. Usually a management decision is required to move into the next phase which involves more detailed specifications of performance characteristics. This phase is typified by numerous briefings designed to establish a recognized need for and cost/effectiveness of the system vis-a-vis organizational missions and functions. Order of magnitude cost figures are the typical modus operandi.

This phase has a relatively low contribution to total cost and may last several years. The question of *software* quality assurance is essentially moot throughout this phase of the life cycle. However, the role of software as it may interface with hardware, and gross estimates of costs and schedules should be reviewed as part of a larger quality assurance effort.

Failure to adequately address these issues could result in having to incorporate into the software development functions or design features which could have been accomplished better in other ways and which restrict flexibility or increase the complexity of the software.

The requirements phase

This phase of the software life cycle refines the conceptual system, further delineating the functions and interplay between hardware, software and the user. In general, data inputs and system outputs are specified and overall load and performance characteristics are determined. In many cases, specific determinations of system hardware and user-oriented languages are made. A properly designed Request for Proposal (RFP), even if the system is to be done in-house (this step in the design process is skipped only at considerable risk), treads a thin line between over-specification and insufficient detail. The former is often caused by past contractor failures while the latter is a reflection of the fact that the user simply does not know what he really wants or needs.

To a large extent, the "die is cast" with the issuance of an RFP (or corresponding internal document). The constraints placed on system performance, hardware and software at this early stage of the life cycle can have enormous repercussions on the flexibility, reliability, maintainability and cost of the system. Implicit trade-offs between system throughput and ease and cost of use, enhancement and maintenance are often made.

Realistically one cannot expect a prospective vendor to do the necessary work required to examine and weigh each of the possible solutions to the design problem. Even with the most competent of vendors, their objective function differs from the clients. Specifically, a vendor's staff may have certain backgrounds and expertise, or his equipment characteristics more adaptable to one family of solutions than another. To save time or money a vendor may modify an already developed product or assemble a patch work of available system modules rather than seek an "optimal" solution.

Thus, quality assurance cannot begin any later than this phase without considerable risk. The phases which follow are characterized by much higher expenditures than these first two phases, with the obvious result that errors carried forward from this point are very costly.

The development phase

This phase is a transitional one bridging the gap between a well defined concept and an implementable system. The big black box between inputs and outputs has to be broken down into programmable units, logic determined and finally coded. The testing and validation tasks require the generation of test data and test parameters and the development of test tools. Documentation provides the vital link to connect the test activities to the designers, programmers and coders.

It is during this stage that a QA activity reaches its peak, for with increasing detail and concreteness comes the need for constant monitoring to assure that the system in reality is the system in concept. Quality assurance in this phase is simultaneously concerned with the correctness of (1) functional requirements, (2) detailed design, (3) program logic, and (4) code. In addition, the specificity and clarity of the documen-

tation is also a proper subject of a QA plan. The testing and validation of the system or the "quality control" function is the most viable aspect of a quality assurance plan. For many developers, all too often, it *is* the QA plan. This tendency is to become lost in code is at the risk of deviations from intended system functions. Correctness of code is not a guarantee that the code is doing what the user required, but rather that is doing what it was designed to do; quite a different matter. Errors in design are far more perfidious than errors in logic.

The operations phase

This phase bridges the gap between the developers and the users. If QA proponents are correct, some payoff attributed to QA should be noticed during the implementation part of this phase, but its greatest contribution will appear during the productive part of the life cycle which is oddly called "maintenance." This terminology may be an indication of the general lack of quality assurance which exists.

More often than expected, the implementation period becomes a "field test" with the essential aspects of the development phase extending far into the operations phase. Design or even worse functional errors are frequently uncovered which may require extensive reprogramming. The start of implementation is often merely an artificial contrivance to cover a scheduled deadline rather than at the completion of the development phase.

THE SHAPE OF THE SOFTWARE LIFE CYCLE

To place the various aspects of the quality assurance function into perspective, it is necessary to look at the relative costs and time requirements of each of the phases described in the previous section. Figure 2 represents the idealized shape of the software life cycle.[28,13,38,39,40,45] While actual project experience is difficult to come by, a search of the literature for real-world cost and time data has been sufficiently productive to enable the construction of a composite software life cycle. This composite was developed from bits and pieces of available information on different phases and parts of phases of large systems. The degree of consistency found among projects gives rise to a fair degree of confidence that this composite is a useful tool in obtaining estimates of the potential benefits of a QA plan. Using the composite life cycle concept, this section relates the time required for accomplishing each portion of the cycle to software costs expended.

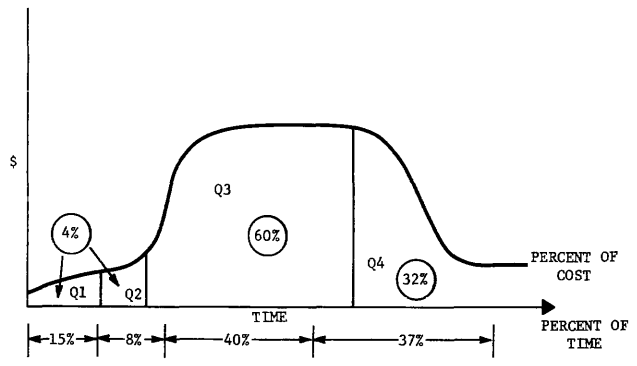

Figure 2—Idealized software life-cycle

The time axis

The percentages of time thought to be denoted to each phase of the software life cycle as implied by the shape of the idealized curve are as follows: Conceptual 15%, Requirements 8%, Development 40%, and Operations 37%. This differed from the reported experience of several large DoD projects.[33] In actual practice the conceptual phase accounted for 30% rather than 15%, while development took only 12% (compared to 40%). The requirement phase accounted for the same percentage of time in actual practice as was expected, while the operations phase (implementation and maintenance) lasted longer in actual practice (50%) than is implied by the curve (37%).

In absolute terms, these projects spanned 16 years from inception to termination. The percentages translate into a conceptual phase of 4½-5 years; a requirements phase of about 1.5 years (these two were actually performed simultaneously for about 6 months); 2 years for development and 8 years for operations. The requirements phase consists of the preparation of specifications, drafting an RFP and the evaluation and selection of a vendor. About half (3½% of Total Life Cycle Time) the time was devoted to specifications. The RFP's took slightly less (2½% TLCT) with about 4-5 months (2%) devoted to review, evaluation and selection. The components of the development phase (2 years) are more difficult to characterize by time, since the steps within are either overlapping (requirements analysis and design) or simultaneous (code, test, document).

Relative cost of software life cycle phases

The relative costs of each of the four phases of the software life cycle can also be inferred from the shape of curve presented in Figure 2. To verify these inferences, data from several studies are pieced together and a composite software life cycle (Figure 4) is constructed and presented in a following section. The shape is compared to the idealized versions found in the literature. It should be remembered that the purpose in developing this composite is to obtain estimates of the relative costs of each phase to use in the determination of the potential effects of instituting various

forms of quality assurance. Therefore, the cost balance between development and maintenance as well as among steps in the development phase were of greatest concern.

Operations vs. development

The balance between development and operation depends primarily upon the length of the maintenance period used in the calculations. To standardize these calculations for comparison purposes a maintenance time period equal to 50% of the total life cycle (or 8 yrs) will be used. A study[9] which monitored costs fairly closely from requirements through *one year* of maintenance reported expenditures (in terms of man years) for Requirements, Development and Operations.

These figures were weighted (3 for management; 2 for programmer and 1 for staff support) to determine costs incurred. Assuming a negligible cost for the conceptual phase, say 1% and an operational life of 8 years, the percentage of total costs incurred by each of the four phases of the life cycle were calculated as follows: Requirements 1.5%, Development 51.3% and Operations 46.2%. The ratio of Development to Operations (Implementation and Maintenance) in this case would be 1 to 1.1.

Implementation is difficult to separate from development and maintenance since it in reality is a transitional period between the two. For this reason data about implementation is hard to find and interpret. This being the case the remainder of this paper treats operations as essentially equivalent to maintenance.

A look at cost data available for development vs. maintenance costs for OS releases 18, 19, and 20.0[13] are even more heavily weighted toward maintenance with ratios of 3 to 1 for OS 18 with only two years of maintenance included and 1.25 to 1 for all three releases with only one year of maintenance included. The experiences reported on in this section with respect to the balance between development and maintenance costs show that the costs of maintenance consistently exceed costs of development. Since QA would be expected to have the greatest impact upon costs in the operations phase, a conservative cost equation (conceptual cost+requirements cost+development cost=operations cost) will be used to *minimize* the estimated potential for QA.

Relative costs within the development phase

The activities undertaken during the development phase can be grouped into (1) analysis and design, (2) coding and debugging and (3) testing or validation.

The ratio of the cost of these activities to one another is often thought to be a function of the complexity of the system to be developed. That is, a non-linear (exponential) relationship is said to exist between complexity and the cost of testing. Testing costs are highly related to the number and severity of errors to be discovered and fixed, the number of which is related to system complexity. Proponents of QA will argue that this exponential relationship need not be the case if proper management (including a good QA plan) is exercised. Since the success of QA is directly related to error rates and error rates are the underlying causes of the cost relationships among the activities undertaken during development, this section will concentrate on the ratio of testing (or validation) to the total of development costs.

	Analysis and Design	*Coding and Debugging*	*Validation*
SAGE	39%	14%	47%
NTDS	30%	20%	50%
GEMINI	36%	17%	47%
SATURN V	32%	24%	44%
OS/360	33%	17%	50%
AVERAGE	34%	18%	48%

SOURCE: 14

Figure 3—Breakdown of development costs for selected systems

A study[8] which looked at the relative costs of design, coding and debugging in relationship to validation reported that the ratio of validation (testing) costs to the total development effort ran between ⅓ to ½.

Figure 3 gives a breakdown of the development phases of five large projects. The results[14] are very consistent from project to project and in the range of the results of the first study referenced. The range ⅓-½ also includes the experience from ALPHA-6[9] reported on earlier in this chapter.

A composite software life cycle, based upon a 16 year length (50% operational life) and the relative costs for the four phases given in Figure 3, is presented in Figure 4. The shape is far more leptokurtic than the

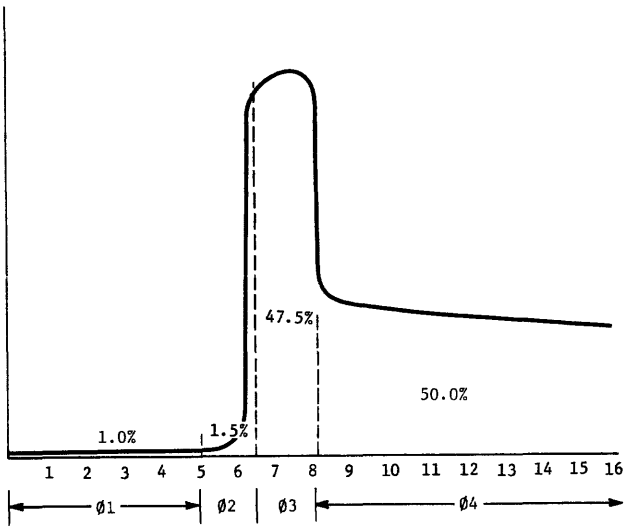

Figure 4—Composite software life-cycle

"idealized" curves found in the literature with the length of the maintenance tail spreading its significant costs over many years. This is due in larger part to a contraction of the development phase. The visual impact of the meager resources applied to the Requirement phase also represents a departure from the idealized shape.

THE COST OF ERROR

A measure of system quality is the number of errors which occur. Hence, ratios of one kind of error to another have been proposed[22] as indicators of quality software. It is taken on faith that well designed systems can be put together with little resultant error, and for those errors which occur, the mean age of the errors becomes a vital statistic with which to judge software.

This section is devoted to estimating the source, kind, type and severity of errors generated during development. It would be of interest to examine the requirements stage to place a value on the "errors" which originate there and trace their impact throughout the rest of the life cycle; but aside from intuitive feelings about their impact no real data appears to be available.

Frequency and severity of errors

No two researchers group errors in quite the same way. As a result, the available information on software errors had to be interpreted and classified based upon the explanations provided in individual studies. Errors are classified in this paper as either design, logic or syntax. These categories are sufficient for the purpose at hand. Design errors are those which require changes in the specifications used by the programmers. Usually they represent a lack of understanding (or proper communication) of a computation or process, which results in the wrong "problem" being solved. Logic errors occur when the system design is translated into programmable form (detailed flow charts). Syntax errors are self-explanatory. Few of the studies of software errors present actual data pertaining to frequency and severity. Taken together[1,4,6,8,47] those that present some data all report design error as occurring most frequently. Ranging from a high of 64%[1] to 46%.[6] Syntax errors were reported to be about 15% of the known errors. Logic errors ranged from 21% to 38%. The significant point to note is the large percentage of design errors.

Available data on cost of detection and correction reveals that design errors cost the most to diagnose and fix. Syntax errors are reported to be more of a nuisance than a significant cost particularly with the use of automated precompiler processing.

Origin and detection of errors

Where errors originate as well as when and how they are discovered are important inputs to the design of an effective QA plan. Syntax errors originate, surface and are resolved within a brief period of time and for all intents and purposes can be considered totally encompassed within the process of coding. Such is not the case with design and logic errors. Design errors can be caught during a design review (if there is one), during preparation of detailed flow charts or occasionally during coding. Simple logic errors (process before read) can be caught at compilation time or during program testing. Because of the numerous paths in any program which can be tested many logic errors are not observed until the validation, implementation or maintenance stages. A study of a large software development effort[1] found that 54% of the errors were not caught until acceptance testing or presumably until after development was complete. To make matters worse, the overwhelming proportion of these were design errors. Reported figures indicated that 70% of the design errors were not caught at earlier stages while by contrast 80% of the programming or logic errors were caught during development. If the mean age of error were calculated for this case it would be quite high due to the high percentage of design errors involved.

Estimation of the costs of errors

Using the three categories of error (design, logic and syntax) it appears that design errors account for at least 80% of the *total cost of error*. This percentage is arrived at by noting that about ⅔ of all errors caught are design errors; with logic and syntax errors making up about equal proportions of the remaining ⅓. Compared to the cost of tracking down and correcting coding errors, the cost of syntax mishaps is small. However, the cost of design errors is more than double (2½ times) that of coding error. The calculation of the contribution of design error to the total cost of error consists of taking the weighted (expected) cost of an error [% design errors × 2½ + % coding + syntax errors × 1] and dividing it into the contribution of design [% design × 2½] using the percentage given above 83⅓% of the total cost of error can be attributed to design errors. This relatively large contribution to total error cost should play an important role in the design of a QA plan and will be used as an input in the calculation of the potential effectiveness of quality assurance.

The next calculation which is required for the assessment of the potential of quality assurance is the percentage of total life cycle cost which can be attributed to error. Once this percentage is obtained, an estimate of the benefits of a QA plan can be developed based on a "tool by tool" analysis of the kind of error it ad-

dresses (design, logic or syntax) and the percent error reduction claimed or experienced. These calculated benefits can then be compared to the cost associated with these components of a QA plan for a final assessment of the economics of quality assurance.

To estimate the cost of error the following method was used. For a slightly conservative estimate, it was assumed that (i) all costs in design (ii) coding and (iii) documentation to be non-error related. All checkout and validation costs (recognizing that only some of these costs can be reduced by reducing the number of errors, since some costs are fixed) were attributed to error. From the ALPHA-6 data then 47% of development cost (assuming the code, test, and document costs were equal) could be traced to errors. Further data on developments costs for several large systems (given in Figure 3) averaged almost exactly the same percentage (48%).

Maintenance costs can be attributed to correcting errors and to enhancements, but "enhancements" often result from initial design errors. For the sake of discussion assume that half can be directly related to error. This amounts to the conservative estimate of almost half of the total life cycle costs (47.6%) being directly tied to error (see Figure 5). On the cost basis of a large system, the total cost of error is in the hundreds of millions. If quality assurance methods can reduce error by even small amounts, they would appear to be worthy of serious consideration. For example, a 10% reduction in error (⅔ Design, ⅓ logic and syntax) as they have been reported in the studies reviewed would represent a saving of almost $25 million based upon a relatively large effort (cost= $½ billion over the 16 year cycle). A five percentage error reduction (only ½ design) would result in a savings of over $10 million.

THE EFFECTIVENESS OF THE TECHNIQUES AND TOOLS OF QUALITY ASSURANCE

The assurance of quality can be brought about by any number of different approaches which have been suggested. These range from essentially project management techniques to methodologies of design to syntax checking tools. Many of the methods which will be discussed in this section can be expected to have a much broader impact on design, development and implementation than is pertinent to a discussion of quality assurance. This section will address the impact these methods have on error rate and error-related productivity.

In some cases, their contribution to quality is rather straightforward, particularly for error detection tools. However, for those which promise the most sweeping reforms, essentially those dealing with management or design effectiveness, measurement is difficult and little concrete information is available.

It is the purpose of this section to analyze based upon available data the potential of quality assurance in terms of the cost of error, development productivity and the cost of quality assurance. In the following paragraphs, some of the most widely discussed techniques and tools will be reviewed.

Error Type	*% Total Errors*	*Relative Severity*	*% Total Cost of Error*
Design	2/3	2.5	83+%
Logic	1/6	1.0	8+%
Syntax	1/6	1.0	8+%
	Development Phase	*Operations Phase*	*Both*
% Total Life Cycle Cost	47.5%	50%	97.5%
% Cost Due to Error	48%	50%	—
% Total Life Cycle Cost Attributed to Error	22.6%	25%	47.6%

Figure 5—Error and software life cycle costs

Structured programming

The advantages touted for Structured Programs range from improved program design to improved documentation. Improved design is linked to fewer design errors and fewer logic/programming errors. Fewer statement types are linked to fewer syntax errors and an almost self-documenting program. Fewer errors imply greater productivity during development and reduced operations costs. Further, the streamlined design is claimed to be easier to upgrade and enhance. Finally, the planning and conceptualization required by Structured Programming is said to enhance the performance of project management.

Reported increases in error free productivity ranged from 50%[25] to 125%[47] with the introduction of Structured Programming while error reductions of between 30%-90% were reported by another study.[17] Quantitative results of ease of enhancements were not found, however, a study of the development cycle[56] estimated 25% reduction in the elapsed time from requirements to implementation, from 6 years to 4.5 years.

Top-down development

The essence of Top-Down Development is simultaneous systems integration and development which results in a viable, executable, if rather skeletal system at an early state. This development approach amounts to an ordering of the sequence of system decomposition decisions beginning with a simple description of the entire system or process and continuing with successive refinements until a programmable design is reached. Top-Down Development is a natural companion of Structured Programming, so much so that

the two concepts are often confused. The claimed advantages of this approach include the increased ease of implementing a QA plan with a resultant reduction in design error and productivity improvements associated with the systems integration and testing efforts during the development phase.

Perhaps the most significant advantage claimed from a QA perspective, is the early existence of a complete system's design replete with the design specification of system components and interfaces. Not only does such a document enhance the changes of a coherent and consistent design, but it also serves as a vehicle for establishing a correspondence with "user" oriented functional specs. The system components are placed into perspective for all to see and comment upon. Misunderstandings that often were not surfaced until acceptance testing can be resolved at this time. Design problems often not found until systems integration may be corrected reducing the high cost currently associated with these problems.

The incorporation of the testing function throughout development, made possible by the continual existence of a testable system, offers QA with an opportunity to be more of a pro-active force in development.

There are recognized pitfalls as well. Care must be taken to ensure design feasibility in terms of existing software and hardware, since actual coding is significantly delayed.

Holistic design is difficult to achieve and false starts are likely. However, when weighted against the known shortcomings of bottom-up design there is little question that a Top Down approach when combined with some common sense offers substantial advantages to both developer and user.

Hard estimates of the reduction in error and increases in productivity from the use of this approach alone are not readily available. However, when used in conjunction with Structural Programming and a Development Support Library,[25] a productivity improvement of over 300% (when compared to a system using a Development Support Library alone) was experienced. With Structural Programming alone, productivity gains of 50%-100% were experienced; thus, the addition of a Top Down Design approach seems to further enhance performance significantly.

For the purposes of this analysis, the expected performance of this approach will be conservatively bounded from above. In terms of development productivity, a very conservative range which includes gains made by reduced systems integration and testing, and by better manpower and computer time scheduling would be between a 5-10% improvement in productivity. This improvement could be reasonably expected from the savings in the integration step alone.

As far as design errors are concerned, the increased attention to overall design could be expected to reduce configuration and architecture errors significantly and virtually eliminate errors in the specification of offered system functions. One study[6] showed that machine configuration and architecture errors accounted for just over 20% of all design errors while errors in the functions offered accounted for about 25% of the design erorrs. Both are susceptible to being caught early. An examination of specs by others not involved in their formulation resulted in the detection of between 30%-40% of these errors. An increase from this to a 50% rate of error detection might realistically be achieved by the use of Top Down Design.

Other recent innovations

In addition to Structured Programming and Top Down Development, a number of other approaches to improving software quality and productivity have been advanced. Among these are the techniques of the Chief Programmer Teams, Egoless Programming, and automatic or semi-automatic tools ranging from Design Assertion Consistency Checkers to Automated Test Case Generators.

The management oriented techniques are aimed at achieving increased communications and coordination while the automated tools seek to provide complete, systematic and low cost verification. This section will briefly explain some of these innovations concentrating on the contribution or impact likely on the performance of the QA functions.

Programming organizations

In this section, the effects of the Chief Programmer Team, Egoless Programming and Democratic Team Organization on the performance of the QA function will be addressed. Egoless Programming and Democratic Teams are essentially loosely structured programming environments in direct contrast to the Chief Programmer approach which is highly structured. It is interesting that the changes from current practice being advanced to improve software quality are in opposite directions. Both approaches, however, take aim at the individualist who becomes ego-involved with code to the extent that error detection is thwarted. The loosely structured approaches attack this problem directly by eliminating "ownership" of code to reduce defensiveness. The Chief Programmer Team approach is meant to be employed in conjunction with Structured Programming and Top Down Design which systematically eliminates tricks and gimmicks in programming and imposes ridged forms. Users of both types of approaches claim better communication leading to reduced misunderstandings and error rates. On the one hand, the Chief Programmer Team approach is criticized for being too authoritarian while the other approaches are said to tend to alleviate the individualist and require more sophisticated management techniques. Experience indicates that managing bright

Error Type	Methods of Detection: Manual Inspection	Formal Methods (Simulation, etc.)	Tests Runs	
Design	45%	20%	35%	100%
Logic/ Coding	24%	22%	54%	100%

Figure 6—Error detection for design and logic/coding error types

creative staff is no mean task regardless of the techniques employed. The key seems to be in the actions taken to increase the understanding and clarity of assignments not in what abstract management philosophy is employed.

Automated tools for quality asssurance

The literature contains countless tools developed to check out design, flow charts, code and even documentation systematically and quickly. Their performance can more easily be measured than the techniques previously discussed, but their contribution to the potential of an overall QA plan is limited. Their very nature (highly specified and deterministic) limits their effectiveness in dealing with other than highly structured situations. Thus, these tools are most applicable to the detection of errors in code and simple sorts of logic errors rather than major flaws in program logic or design approach. Nevertheless, they can significantly contribute to increased productivity, earlier detection and hence some reduction of the "ripple" effect (19% of the errors introduced as a result of error correction[55]). An analysis of error types and means of detection[6] showed that (See Figure 6) manual inspection uncovered only 24% of logic and coding errors compared to 45% of design errors indicating the potential for the use of automated tools. Such tools could have an impact in reducing the percentage of logic and coding errors (54%) not caught until testing. One study gave evidence to support this feeling[21]. The use of automated instruction and path checkers (ASSIST and NODAL) reportedly catch between 67%-100% of the errors and at between 2-5 months earlier than they would have otherwise been detected. Automated error checking is currently at the state of development where it is either language or application specific and it would probably be of marginal value to develop such a tool for a specific project.

Figure 7 summarizes the results with respect to the reported effectiveness of quality assurance methods and shows the dollar impact that improvements in development productivity can have based upon a project whose total life cycle costs equal $.5 billion.

SUMMARY AND CONCLUSIONS

This section places the relevant estimates developed during this report in perspective and highlights important aspects in the assessment of the economics of Software Quality Assurance. This paper first addressed the software life cycle to identify the areas which could be improved by a QA plan. Second, an examination of the frequency of software error, its sources or origins, methods of detection and associated costs was presented. This was followed by an examination of some of the methods and techniques suggested for quality assurance. Highlights of these examinations and analyses follow.

Summary of findings

The examination of the software life cycle revealed that costs were concentrated in the Development and Operations phases. The typical Development Phase accounted for just under 50% of the total costs while lasting about 2 years (12% of a 16 yr. cycle). About half of the development costs were spent on check-out and testing activities in contrast to about 1/3 for analysis and design and 1/6 for actual coding. The Operations phase while consuming just under 50% of the total life cycle costs was spread over an eight year period.

Errors were classified into three types (design, programming/logic, syntax). The last accounting for some 15% of all errors. Design errors outpaced program/logic errors by a little less than 2 to 1 accounting for a little more than half of all errors. Program/logic errors ran about one-third of the total.

The severity of errors, as measured by the cost of detection and correction, was found to be higher for

Technique	Error Reduction	Productivity: Range	Productivity: Mid-Point	$ Impact of 1% Improvement In Development	Potential Impact
Structured Programming	30-90%	50%-100+%	75%		$175 Million
Top-Down Design	Substantial	10%-200%	100%		$250 Million
Management Organization	—	—	—	$2,375,000	—
Automated Tools	Caught earlier	Up to 25%	10%		$ 25 Million

Figure 7—Performance of quality assurance techniques

design errors than program/logic or syntax errors (least costly). Weighted by costs it was calculated that design errors accounted for just over 80% of the total cost of error. In terms of the total software life cycle then, with 47.5% of its costs in development and 50% in maintenance, the cost of error could easily run over 50% of the total software life cycle cost.

To combat error and improve software quality a variety of methods have been suggested. Preliminary reports have been encouraging in both the areas of productivity improvement and error reduction.

Conclusions

While the data drawn upon comes from a large variety of sources (different systems, different environments and from studies using different definitions and analysis methodologies), the experiences reported were so compatible that, while more detailed data is necessary for the actual development of a QA plan specific to a given set of system and organizational circumstances, the conclusion that QA can be cost effective is inescapable.

From the analysis presented in this paper, the development of a QA plan should concentrate on techniques and methods for the early detection and elimination of design errors. The researchers reporting on the development of ALPHA-6[9] indicated that if more resources were applied during design, it would have resulted in substantial savings in the costs of testing and maintenance. An extrapolation of the data they presented gives a multiplicative factor of 5; that is, a dollar more spent in design would have saved 5 dollars spent on testing and maintenance. While this example may be unusual, it, together with the fact that a significant portion of total system cost can be attributed to error point to the cost impact that a QA function can provide.

A parameterization of the impact that error and productivity improvements have on total software system costs based upon a $\$1\frac{1}{2}$ billion total life cycle cost (about $250 million for S/W Development) has been made. For each 1% of error reduction ($\frac{1}{2}$ coding $+\frac{1}{2}$ design) a savings of just over $\$1\frac{1}{2}$ million could be expected. For each 1% improvement in Development productivity a saving of $2,375,000 could be expected. It should be noted that Design errors have more than double the impact than do coding errors.

Thus the leverage of QA in large programs is significantly high to warrant serious consideration. The costs of developing and implementing a QA plan are difficult to specify for a given organization, especially in light of their management considerations. However, even with the additional expense QA still promises to be cost-effective. For example, if management overhead for software development is approximately 5% of development costs and a QA plan increased this overhead by $\frac{1}{4}$, then a reduction of error by approximately 1% (coding) alone could offset these additional costs.

REFERENCES

1. Boehm, B. W., et al, "Some Experince With Automated Aids to the Design of Large-Scale Reliable Software," *IEEE Transactions on S/W*, TRW, March 1975.
2. Boehm, B. W., "Software and Its Impact—A Quantitative Assessment," *Datamation*, TRW, May 1973.
3. Brown, J. R., et al, "Evaluating the Effectiveness of Software Verification—Practical Experience With an Automated Tool," *AFIPS Fall Joint Computer Conference*, December 1972.
4. Shooman, M. L., et al, "Types, Distribution, and Test and Correction Times for Programming Errors," *IEEE Transactions*, Bell Labs, March 1975.
5. Schneidewind, N. F., "Analysis of Error Processes in Computer Software," *IEEE Transactions*, Naval Postgraduate School, March 1975.
6. Endres, A., "An Analysis of Errors and Their Causes in System Programs," *IEEE Transactions*, IBM Lab, Germany, March 1975.
7. Rubey, R. J., et al, *Comparative Evaluation of PL/1*, USAF ESD-TR-68-150, April 1968.
8. Rubey, R. J., "Quantitative Aspects of Software Validation," *IEEE Transactions*, LOGICON, March 1975.
9. Buda, A. O., et al, "Implementation of the ALPHA-6 Programming System, IEEE Transactions," *Academy of Sciences* USSR, March 1975.
10. Ramamoorthy, C. V., et al, "Testing Large Software Evaluation Systems," *IEEE Transactions*, CSD, ERL, University of California, Berkeley, March 1975.
11. Alexander, T., "Computers Can't Solve Everything," *Fortune*, May 1969.
12. Boehm, B. W., "System Design," *Planning Community Information Utilities* (ED) H. Sackman, AFIPS Press, 1972.
13. Barry, B., et al, *Software Life Cycle Considerations*, IBM, January 1974.
14. Boehm, B. W., "Some Information Processing Implications of Air Force Space Missions: 1970-1980," *Astronautics and Aeronautics*, January 1971.
15. Vick, C. R., *Specification for Reliable Software*, EASCON, 1974.
16. Brown, J. R., et al, "Evaluating the Effectiveness of Software Verification—Practical Experience with an Automated Tool," *AFIPS Conf.*, 1972.
17. Cammack, W. B., et al, *Improving the Programming Process*, IBM SDD TR002483, October 1973.
18. Cheng, L. and J. E. Sullivan, *Case Studies in Software Design*, MTR-2874, Volume I, June 1974.
19. Boden, W. H., "Designing for LCC," *EASCON 74*, pp. 624-29.
20. Knight, C. R., "Warranties as a Life-Cycle-Cost Management Tool," *EASCON 74*, pp. 621-623.
21. Mangold, E. R., "Software Error Analysis and Software Policy Implications," *EASCON 74*, pp. 123-27.
22. Mills, H. D., "How to Buy Quality Software," *EASCON 74*, pp. 120-22.
23. Nashman, A. E., "Software Development Management: The Key to Quality Software Products," *EASON 74*, pp. 31-35.
24. Oliver, P., "Observations on Software Reliability," *EASCON 74*, pp. 126-29.
25. Baker, F. T., "Structured Programming in a Production Programming Environment," *IEEE Transactions on S/W Rel.* 75, pp. 172-185.
26. Rain, M., *Two Unusual Methods for Debugging S/W Software Practice and Experience* 3, pp. 61-63.

27. Katzenelson, J., *Documentation and Management of Software Project,* SP & E 1, 2, pp. 147-157.
28. Peadi, P., *Quality Control for Computer Programming: A Final Report on an Initial Study,* SDC, Santa Monica, California, September 1965.
29. ———, *QC for Systems and Programming, A Survey of the Literature,* SDC, March 1965.
30. Connolly, J. T., *Software Acquisition Management Guidebook: Regulation, Specifications and Standards,* MTR-3080, The MITRE Corporation, June 1975.
31. Clapp, J. A., *Major Contributions to Software Engineering in the 1980's,* MTR-2791, The MITRE Corporation, January 1974.
32. ———, *A Software Error Classification Methodology,* MTR-2648, Volume VII, The MITRE Corporation, June 1973.
33. Reifer, D. J., "Automated Aids for Reliable Software," *IEEE Transactions on S/W Reliability,* March 1975, pp. 131-42.
34. Wulf, W. A., "Reliable Hardware-Software Architecture," *IEEE Transactions on Software Reliability,* March 1975, pp. 122-30.
35. Cicu, A., et al, "Organizing Tests During Software Evolution," *IEEE Transaction on Software Reliability,* March 1975, pp. 43-50.
36. Williams, R. D., "Managing the Development of Reliable Software," *IEEE Transactions on Software Reliability,* March 1975, pp. 43-50.
36. Williams, R. D., "Managing the Development of Reliable Software," *IEEE Transactions on Software Reliability,* March 1975, pp. 3-8.
37. Ceoff, N. S., "Development Project Costs," *Journal of Systems Management,* September 1974, pp. 14-17.
38. Aron, J. D., *Characteristics of the Program System Development Life-Cycle,* IBM FSD 74-0180.
39. Pietrasanta, A. M., "Resource Analysis of Computer Program System Development," *On The Management of Computer Programming,* G. F. Weinwurm, (Editor): Auerbach Publishers, 1970.
40. Brooks, F. P., Jr., "Why is Software Late," *Data Management,* Volume 9/8, August 1971.
41. Clapp, J. A. and J. E. Sullivan, *SIMON: Finding the Answers to Software Development Problems,* MTP-159, The MITRE Corporation, May 1974.
42. Cheng, L. L., *Some Case Studies in Structured Programming,* MTR-2648, Volume VI, The MITRE Corporation, June 1973.
43. Fleischer, R. J., *Effects of Management Philosophy on Software Production,* MTR-2648, Volume II, The MITRE Corporation, June 1973.
44. Corrigan, A. E., *Results of an Experiment in the Application of Software Quality Principles,* MTR-2874, Volume III, The MITRE Corporation, June 1974.
45. Schiff, J. D., "An Overview of the Software Life-Cycle Process," *Proceedings of the Aeronautical System Software Workshop,* Dayton, Ohio, April 1974, p. 108.
46. Prywes, N. S., *Research on Automatic Program Generation,* Report 74-05 University of Pennsylvania Moore School of Electrical Engineering, January 1974.
47. Boles, S. J. and J. D. Gould, *A Behavioral Analysis of Programming—On the Frequency of Syntactical Errors,* IBM RC 3907, June 1972.
48. McGonagle, J. D., *A Study of a Software Development Project,* J. P. Anderson and Company, September 1971.
49. Nichols, B. S., *Practical Experience With Structured Programming,* Bell Systems, November 1973.
50. Mill, H. D., "Top Down Programming in Large System," *Debugging Techniques in Large Systems,* R. Rustin (ED) Prentice Hall, 1970.
51. Baker, F. T., "Chief Programmer Team Management of Production Programming," *IBM Systems Journal* 11, 1972.
52. Sackman, A., *Man-Computer Problem Solving,* Auerbach Publishers, 1970.
53. Weinberg, G. M., *The Psychology of Computer Programming,* Van Nostrand Reinhold, New York, 1971.
54. Baker, F. T., "System Quality Through Structured Programming," *AFIPS Conference Proceedings,* 1972, pp. 339-343.
55. McGonagle, J. D., *A Study of a Software Development Project,* James P. Anderson and Company, Los Angeles, California, 1971.
56. Haile, A., *Command and Control Information Processing in the 1980's* (USAF-CCIP-85) Presentation in DoD Computer Institute Seminar IX, November 1972.
57. Asch, A., et al, *DoD Weapon Systems Software Acquisition and Management Study,* Vols. I and II, The MITRE Corporation, MTR-6908, 1975.