The Economics of Software Quality Assurance: A Simulation-Based Case Study

By: Tarek K. Abdel-Hamid
Department of Administrative Sciences
Naval Postgraduate School
Monterey, California 93943

Abstract
Software quality assurance (QA) is a critical function in the successful development and maintenance of software systems. Because the QA activity adds significantly to the cost of developing software, the cost-effectiveness of QA has been a pressing concern to software quality managers. As of yet, though, this concern has not been adequately addressed in the literature.

The objective of this article is to investigate the tradeoffs between the economic benefits and costs of QA. A comprehensive system dynamics model of the software development process was developed that serves as an experimentation vehicle for QA policy. One such experiment, involving a NASA software project, is discussed in detail. In this experiment, the level of QA expenditure was found to have a significant impact on the project's total cost. The model was also used to identify the optimal QA expenditure level and its distribution throughout the project's lifecycle.

Keywords: Software quality assurance, software project management, software cost, system dynamics

ACM Categories: D.2, D.2.5, D.2.9, I.6, I.6.3

Introduction
The IEEE standard P730 defines software quality assurance (QA) as "a planned and systematic pattern of all actions necessary to provide adequate confidence that the software conforms to established technical requirements" (Buckley and Poston, 1984). QA not only holds the key to customer satisfaction, but as more and more software managers are starting to realize, it has a direct impact on the cost and the scheduling of a project. "Failure to pay attention to QA has often resulted in budget overruns, schedule delays, and failure to meet the needs of the customer" (Chow, 1985).

The utilization of QA tools and techniques does, however, add significantly to the cost of developing software. For example, man-hours are needed for developing and running test cases and conducting structured walkthroughs. These added costs are a source of concern to everyone associated with the QA program, particularly the program manager and the customer.

A pressing concern to the software quality manager is how cost efficient are the QA operations during the development cycle. The QA organization, just as all elements of the development process, will and should be subject to detailed and continuing scrutiny regarding the cost of doing business (Knight, 1979).

Furthermore, QA staffs are continually being challenged to translate the cost of quality into dollars and cents terms. They are realizing that such reinterpretation of quality is essential for gaining the involvement and support of top management (Riggs, 1983).

These "pressing concerns" shared by QA managers and their customers are the focus of this article. Specifically, the article investigates the tradeoffs between the economic benefits and costs of the QA effort. Such considerations obviously lie at the heart of any QA planning process.

A Case Study: The DE-A Software Project
Consider the case of a software project conducted to develop a software system for a space application. The development and target machines were the IBM S/360-95 and -75. The programming language was FORTRAN. The system was estimated to be 16,000 delivered source instructions (DSI) in size; require 1,100 man-days for development and testing; and be completed in 320 working days. How much should management allocate to QA?
The above project is not a hypothetical scenario. Indeed, it is a real project that was conducted at the Systems Development Section of NASA’s Goddard Space Flight Center at Greenbelt, Maryland. The basic requirements for the project were to design, implement, and test a software system for processing telemetry data and for providing attitude determination and control for the DE-A satellite.

As is typically the case, resources for QA were allocated as a function of the project’s total development effort. In this case, approximately 30% of the DE-A project’s development resources were allocated to QA (e.g., for walkthroughs and reviews)—a level that is significantly higher than the industry norm (Boehm, 1981).

Upon completion, the DE-A project’s statistics were:

- Project size: 24,000 DSI
- Development cost: 2,200 man-days
- Completion time: 380 working days

Given these statistics, was the QA effort optimal? If not, what were the impacts on the project’s cost and schedule? Such issues are obviously of great interest to management. They constitute the primary research issues that will be addressed in the following sections.

The Need for an Experimentation Vehicle

In software engineering it is remarkably easy to propose hypotheses and remarkably difficult to test them. Controlled experiments have proven to be too costly and time consuming (Myers, 1976). Furthermore, even when affordable, the isolation of the effect and the evaluation of the impact of any given practice within a large, complex, and dynamic project environment can be exceedingly difficult (Glass, 1982). Accordingly, it is useful to seek other methods for testing software engineering hypotheses.

Simulation modeling provides a viable laboratory tool for such a task. In addition to permitting less costly and less time-consuming experimentation, simulation-type models make "perfectly" controlled experimentation possible. Indeed:

The effects of different assumptions and environmental factors can be tested. In the model system, unlike the real systems, the effect of changing one factor can be observed while all other factors are held unchanged. Such experimentation will yield new insights into the characteristics of the system that the model represents.

By using a model of a complex system, more can be learned about internal interactions than would ever be possible through manipulation of the real system. Internally, the model provides complete control of the system’s organizational structure, its policies, and its sensitivities to various events (Forrester, 1961, p. 55).

The next section proposes a system dynamics-based simulation approach to the study of the software development process in general and the economics of QA in particular. First, an overview of the model’s structure is given, next more details of the model’s QA sector are discussed and the validity of the model is tested, and finally the model’s experimental results pertaining to the economics of QA are presented.

A System Dynamics Model of Software Development

Model structure

It is important to note that this research work on the economics of QA is being conducted within the context of a much broader research effort to study, gain insight into, and make predictions about the dynamics of the entire software development process. A major part of this effort is devoted to the development of a comprehensive system dynamics computer model of software development. The model is currently being used in several research capacities, one of which is to serve as a laboratory vehicle for conducting experimentation in the area of QA, the topic of this article.

The model was developed on the basis of field interviews of software project managers in five organizations, complemented by an extensive database of empirical findings from the literature. The model integrates the multiple functions of the software development process, including both the management-type functions (e.g., planning, controlling, and staffing) as well as the software production-type activities (e.g., designing, coding, reviewing, and testing). Figure 1 depicts an overview of the model’s four major subsystems: (1) the human resource management subsystem; (2) the software production subsystem; (3) the controlling subsystem; and (4) the planning subsystem. Figure 1 also illustrates some of the interrelations among the four subsystems.

Because the model is quite comprehensive and highly detailed, it is infeasible to fully explain it in the limited space of this article. Therefore, the description is limited to a high-level overview of the
four subsystems. The model's QA component (which is part of the software production subsystem) is, however, discussed in more detail in a later section. (The interested reader can refer to Abdel-Hamid, 1984 or Abdel-Hamid and Madnick, 1988a; 1988b for a full description of the model's structure and its mathematical formulation.)

The human resource management subsystem captures the hiring, training, and transfer of the human resource. Such actions are not carried out in a vacuum, but are affected by the other subsystems; for example, the hiring rate is a function of the workforce level needed to complete the project by a given (planned) date. Similarly, the available workforce has a direct bearing on the allocation of manpower among the different production activities.

The development lifecycle phases incorporated in the software production subsystem include the designing, coding, and testing phases. The initial requirements definition phase is excluded for two reasons: (1) primarily because this study focuses on the "endogenous" software development organization, i.e., the project managers and the software development professionals, and how their policies, decisions, actions, etc. affect the success/failure of software development. In many environments the definition of user requirements is not totally within the control of the software development group (McGowan and McHenry, 1980); (2) "Analysis to determine requirements is distinguished as an activity apart from software development. Technically, the product of analysis is non-procedural (i.e., the focus is functional)" (McGowan and McHenry, 1980).

As the software is developed, it is also reviewed, using quality assurance activities such as structured walkthroughs to detect any errors. Errors detected through such activities are reworked. However, some "escape" detection until the testing phase.

As progress is made, it is reported. A comparison of the degree of project progress to the planned schedule is captured within the control subsystem. Once an assessment of the project's status is made, it becomes an important input to the planning function.

In the planning subsystem, initial project estimates are made and then revised, when necessary, throughout the project's life. For example, to handle a project that is behind schedule, plans can be revised to (among other things) hire more people, extend the schedule, or do both.

In addition to integrating the managerial and technical aspects of software development, the modeling approach in this study has a second important feature that should be noted. The feedback principles of the system dynamics methodology are utilized to structure and clarify the complex web of dynamically interacting variables. Feedback is the process in which an action taken by a person or thing will eventually affect that person or thing. The significance and applicability of the feedback systems concept to managerial systems has been substantiated by a large number of studies (Roberts, 1981). For example, Weick (1979) observes that:

The cause-effect relationships that exist in organizations are dense and often circular. Sometimes these causal circuits cancel the influences of one variable on another, and sometimes they amplify the effects of one variable on another. It is the network of causal relationships that impose many of the controls in organizations and that stabilize or disrupt the organization. It is the patterns of these causal links that account for much of what happens in organizations. Though not directly visible, these causal patterns account for more of what happens in organizations than do some of the more visible elements such as machinery, timeclocks, ... (p. 7).

It is no wonder, then, that many software managers get into trouble because they forget to think in circles. This is meant literally. Managerial problems persist because managers continue to believe that there are such things as unilateral
causation, independent and dependent variables, origins, and terminations.

Consider, as an example, the feedback loop of Figure 2a. It portrays how project work is accomplished through the utilization (1) of project resources (manpower, facilities, equipment). As (2) work is accomplished on the project, it is reported (3) through a project control system. Such reports cumulate and are processed to create the project's forecast completion time, i.e., adding to the current date the indicated time remaining on the job. The feedback loop is completed (closed) if the difference, if any, between the scheduled completion date and the forecast completion date causes adjustments in the magnitude or allocation of the project's resources (4).

The feedback loop of Figure 2a provides only a very high-level overview of the project control process. At a more detailed level, a far more complex conglomerate of interconnected feedback loops exists. As an example, consider the feedback loop of Figure 2b, which portrays some of the dynamic forces directly impacting the QA activity. The loop shows how schedule pressures, which arise when a software project falls behind schedule, can lead to a higher error generation rate. As more errors are committed, a larger chunk of the available manpower is diverted from development work and devoted instead to error correction and rework duties. As this happens, the project's progress rate drops further, leading to even greater schedule pressures and necessitating another pass around this "vicious cycle."

However, project managers do have "escape" mechanisms to break loose from this positive feedback loop. For example, as schedule pressures persist (e.g., after several passes around the loop), project managers can add more people, extend the schedule, trim project deliverables, etc.

The details of error generation, detection, and correction

Figure 3 details the model's structure for the generation, detection, and correction of errors. This component of the model together with two others—software development and system testing—constitute the software production subsystem.

The schematic conventions used in Figure 3 are the standard conventions used in system dynamics models. All the quantities appearing in such models can be classified into two broad groups: constants (whose values cannot change at all in the course of a simulation), and variables (whose values can change). The symbol for a constant is shown below:

\[
\text{CONSTANT}
\]

Model variables are one of three types: level, rate, and auxiliary. A level is an accumulation, or an integration, over time of flows or changes that come into and go out of the level. The flows increasing and decreasing a level are called rates. Thus, DETECTED ERRORS is a level of errors that is increased by the ERROR DETECTION RATE and decreased by the REWORK RATE.

Rates and levels are shown below:

The cloud-like symbols represent sources and sinks for the "stuff" that flows into and out of levels. The flows that are controlled by the rates are either information flows or physical flows. The two types of arrow designators used are:

INFORMATION FLOWS

OTHER FLOWS (e.g., People)

In principle, levels and rates are sufficient to represent all variables in a system dynamics model. Usually, however, it is very difficult to write a rate equation without first doing some (often complex) algebraic computations. These additional algebraic computations are termed auxiliaries. Thus, auxiliary variables, as their name implies, aid in the formulation of rate equations. Auxiliary variables are represented by a circular symbol.

Finally, variables that are defined in sectors of the model other than the one(s) diagrammed are represented by enclosing the variable name in parentheses.
Figure 2. Example Feedback Loops
Error Generation Factors: Three sets of factors affect the error generation rate in a software project. The first set includes organizational factors (e.g., the organization's use of structured techniques (Alberts, 1976), the overall quality of the staff (Belford, et al., 1977)). A second set includes project-specific factors (e.g., project complexity, system size, programming language). Even though these two sets of factors differ from organization to organization and from one project to another, they tend to remain constant throughout the development lifecycle of any single project. This means that in studying the dynamics of software quality assurance during the lifecycle of a particular software project (the concern in this study), the above variables can be assumed to remain constant. In the model, such factors are captured through the model's nominal error parameters.

The NOMINAL NUMBER OF ERRORS COMMITTED PER TASK is such a parameter. It captures the error generation characteristics of a particular project environment, i.e., the software product's characteristics as well as those of the organization in which it is developed. Thus, its value requires modification only when modeling different organizational settings or different projects, but not while experimenting on a particular software project (the scenario in this study). A task is a unit of work such as a software module, a page of documentation, etc. in the DE-A project, for example, it is defined as a 40 DSI software module.

In order to capture the generation of different error types, the NOMINAL NUMBER OF ERRORS COMMITTED PER TASK is not formulated as a scalar, but rather as a continuous function that changes in value as the project progresses through its lifecycle (i.e., as a function of the % OF JOB WORKED). The formulation of the NOMINAL NUMBER OF ERRORS COMMITTED PER TASK serves two purposes. First, its absolute value reflects the particular error generation characteristics of a specific project environment. Second, its shape over the project's lifecycle reflects the relative generation rates of different error types throughout the life of a project. (As an illustration, the parameter profile characterizing the DE-A software project is presented in the Appendix.)

In addition to the organization- and project-specific factors discussed above, a third set of factors affecting error generation includes the work-force mix and schedule pressures. Unlike the factors in the first two sets, these two variables cannot be preset for a particular project environment. Instead, these factors acquire their values dynamically depending on how the project is being conducted.

Consider the impact of work-force mix on error generation. The work-force level in the model is segregated into two types of employees—newly hired and experienced. Newly hired project members typically pass through a project orientation period when they are less than fully productive. The orientation process educates them through training that covers both the social as well as the technical environments of the project (Couger and Zawacki, 1980). During this training period, newly hired employees tend to be more error-prone than their experienced counterparts (Endres, 1975; Myers, 1976). It is assumed in the model, based on the research findings reported in Abdel-Hamid (1984), that a newly hired employee is, on the average, twice as error-prone as an experienced employee.

The second dynamic variable that can drive error generation is schedule pressure (Mills, 1983; Putnam and Fitzsimmons, 1979; Radice, 1982). According to DeMarco (1982):

People under time pressure don't work better, they just work faster ... In the struggle to deliver any software at all, the first casualty has been consideration of the quality of the software delivered.

Two explanations have been proposed in the literature to explain why schedule pressures cause more errors to be generated. Schneiderman (1980) suggests that schedule pressures increase the "anxiety levels" of programmers. A high anxiety level, then

...interfaces (with performance), probably by reducing the size of the short-term memory available. When programmers become more anxious as deadlines approach, they tend to make even more errors ... (Schneiderman, 1980).

A second explanation is provided by Thibodeau and Dodson (1980). They suggest that schedule pressures often result in the unintended overlapping of activities, which, in turn, can significantly increase the chance of errors. For example,

When coding has begun before the completion of design, the designers are required to communicate their results to the programmers in a raw, unqualified state, hence significantly increasing the chance of design errors ... This is not to suggest that systems cannot be developed with overlapping activities. Many systems have distinct parts that can be coded before the entire design is
completed ... We are concerned here with the situation where the press of the development schedule or the slippage of preceding activities results in overlapping activities that would have been accomplished better sequentially (Thibodeau and Dodson, 1980).

Based on the above, the ERROR GENERATION RATE can now be determined. First, a nominal error generation rate is computed as the product of the NOMINAL NUMBER OF ERRORS COMMITTED PER TASK and the SOFTWARE DEVELOPMENT RATE (i.e., the number of tasks developed per unit of time). Then, the two dynamic factors % OF WORK FORCE EXPERIENCED and SCHEDULE PRESSURE adjust this nominal rate upwards or downwards, depending on the state of the project, to arrive at the actual ERROR GENERATION RATE.

The errors generated as the software is developed remain as POTENTIALLY DETECTABLE ERRORS until the software is reviewed and tested. Any detected errors are then reworked. Usually, though, some errors will "escape" and pass undetected into subsequent phases, where detection (if any) is at a much higher cost.

**Error Detection Factors:** The objective of the software quality assurance (QA) activity is to detect the software errors that have been generated. The QA RATE shown in Figure 3 has a rather non-characteristic mathematical formulation (with its special schematic representation), known as an exponential delay. The "characteristic" way is to formulate the rate of accomplishing some activity as a product of the effort allocated to the activity and the productivity at which this effort is utilized. However, our field studies indicate that the QA rate tends to be independent of the allocated QA effort and its productivity!

In five software-producing organizations studied in this research effort, QA effort is planned and allocated as a fixed schedule of periodic group-type functions. For example, two-hour walkthroughs for project members are scheduled once a week. During these periodic "QA windows," all tasks developed since the previous one are supposed to be processed. A surprising finding showed that all completed tasks, irrespective of how many there were, were always indeed "processed." No backlogs, therefore, develop in the QA pipeline even when QA activities are suspended temporarily because of schedule pressures. For example, when walkthroughs are suspended on a project, the requirement to review the affected tasks is bypassed, not postponed. (This behavior was also reported by others in the literature, e.g., Hart (1982) and Mitchell (1980)).

Since the objective of the QA activity is to detect errors and since undetected errors are by their very nature invisible, it is almost impossible to tell whether an adequate QA job was done (except much later in the lifecycle). Under such circumstances it is easy to rationalize both to oneself and to management that the QA job that was "convenient" to do, was not insufficient.

Furthermore, the QA effort that is convenient to expend (given scheduling considerations) is usually never exceeded even when more effort is needed. There seem to be no significant incentives to do otherwise. First, at a psychological level, there are actually disincentives for working harder at QA, since it only "exposes" more of one's mistakes (Weinberg, 1971). Second, at the organizational level, there are seldom any real reward mechanisms in place to promote quality or quality-related activities (Cooper and Fisher, 1979).

The formulation of the QA RATE as an exponential delay provides a good approximation for this "Parkinsonian execution" of the QA activity. It says that software tasks that are developed will always be quality analyzed (or, more accurately, considered quality analyzed) after a certain delay, which is independent of the actual QA effort allocated.

However, the effectiveness of QA, obviously, depends on that effort. That is, the amount of errors that are detected will necessarily be a function of the amount of QA effort allocated. This is evident in Figure 3, where the ERROR DETECTION RATE equals DAILY MANPOWER FOR QA divided by QA MANPOWER NEEDED TO DETECT AN ERROR.

As is the case with the ERROR GENERATION RATE, the QA MANPOWER NEEDED TO DETECT AN ERROR is a function of organization-type factors such as the overall quality of the staff, as well as project-specific factors such as project complexity and programming language. As explained before, all such factors are assumed to remain constant during the lifecycle of any single software project. Such factors are captured in the model through the NOMINAL QA MANPOWER NEEDED PER ERROR. Because different error types differ in how costly they are to detect, this nominal parameter is not a scalar. It is a continuous function that changes as the project progresses through its lifecycle. Specifically, design-type errors are not only generated at a higher rate, as was discussed above, but are also more costly to detect than coding-type errors (Alberts, 1976; Boehm, et al., 1975; Myers, 1976).

MIS Quarterly/September 1988 401
Figure 3. Model Structure for the Generation, Detection, and Correction of Errors
The (actual) QA MANPOWER NEEDED TO DETECT AN ERROR, in addition to being a function of error-type, also depends on the efficiency of how people work. Man-hours are lost on communication and other non-project activities (e.g., personal business, coffee breaks, etc.). These types of losses are captured in the model's MULTIPLIER TO PRODUCTIVITY DUE TO COMMUNICATION AND MOTIVATION LOSSES, which represents the average productive fraction of a man-day. In other words, if the communication and motivation losses amount to a 4 man-hour loss per day (for the average project member), then the value of the multiplier would be 0.5 (assuming an 8-hour work day).

Finally, there is the effect of error density on the error detection activity. At any point in time, the set of POTENTIALLY DETECTABLE ERRORS constitutes a hierarchy of errors, in which some are more subtle, and therefore more expensive to detect than others. Empirical results reported by Basili and Weiss (1982) suggest that the distribution is pyramid-like, with the majority of errors requiring a few hours to detect, a few errors requiring approximately a day to detect, and still fewer errors requiring more than a day to detect.

The author assumes in the model that as QA activities are performed, the more obvious errors will be detected first. As they are detected, it becomes increasingly expensive to uncover the remaining, more elusive (although less pervasive) errors. At high error densities (greater than 10 errors/KDSI), this factor assumes a neutral role. But as the "obvious" errors are detected and error density decreases, its impact increases in an exponential fashion. For example, when error density decreases to a level as low as 1-2 errors/KDSI, the lingering elusive errors are an order of magnitude more expensive to detect.

To recapitulate, the QA MANPOWER NEEDED TO DETECT AN ERROR is a function of error-type, work efficiency, and error density. Because manpower allocations to QA are often modest and because some errors are simply too difficult to detect during the development phase, not all errors generated will be detected (Shooman, 1983). Inevitably, some errors will escape and pass undetected into the testing phase (see Figure 3).

**Error Correction Factors:** Those errors that are detected through QA are reworked. The REWORK RATE is a function of how much effort is allocated to the rework activity (DAILY MANPOWER FOR REWORK), and the ACTUAL REWORK MANPOWER NEEDED PER ERROR. For example, if the project members commit 10 man-days per week to rework detected errors, and the ACTUAL REWORK MANPOWER NEEDED PER ERROR is 1 man-day, then errors will be reworked at the rate of 10 per week.

The ACTUAL REWORK MANPOWER NEEDED TO CORRECT AN ERROR also depends on the efficiency of the employees. That is, the communication and motivation losses need to be accounted for. For example, if the MULTIPLIER TO PRODUCTIVITY DUE TO COMMUNICATION AND MOTIVATION LOSSES is 0.5, then the actual rework manpower needed to correct an error becomes twice the nominal.

As further demonstrated in Figure 3, the reworking of software errors is not, itself, an errorless activity:

- Human tendency is to consider the 'fix,' or correction, to a problem to be error-free itself. Unfortunately, this is all too frequently untrue in the case of fixes to errors found by inspections and by testing (Fagan, 1976).

The problem of bad-fixes is widely documented in the literature (Endres, 1975; Fagan, 1976; Jones, 1978; Myers, 1976; Shooman, 1983). Shooman and Natarajan (1977) suggest some ways in which bad-fixes may be generated:

1. The correction is based upon faulty analysis, thus complete bug removal is not accomplished.
2. The corrections of a bug may work locally only (i.e., the global aspects of the error still remain).
3. The correction is accomplished, however, it is accomplished by the creation of a new error.

Thus, as detected errors are reworked, some fraction of the corrections will be bad-fixes. The detection and correction of such bad-fixes, together with errors that escape QA detection during the project's development phases, are activities that are captured in the model's system testing sector.
Model validation

The process of judging the validity of a system dynamics model includes a number of objective tests (Richardson and Pugh, 1981). They include:

— **Face validity.** To test the fit between the rate/level/feedback structure of the model and the essential characteristics of the real system. This fit was confirmed by the software project managers involved in the study.

— **Replication of reference modes.** To test whether the model can endogenously reproduce the various reference behavior modes characterizing the real system. Reference modes produced by the model included a diverse set of behavior patterns both observed in the organizations studied as well as reported in the literature (e.g., diminishing returns of QA effort explained below).

— **Extreme condition simulations.** To test whether the model behaves reasonably under extreme conditions or extreme policies. A model that does not behave reasonably under extreme conditions (e.g., zero error density) is suspect, because one may not be certain when aspects of extreme conditions may occur in ordinary runs.

— **Case study.** The DE-A project case study which was conducted after the model was completely developed, constituted an important element in validating model behavior. (NASA was not one of the five organizations studied during model development.)

Any one of these tests by itself is certainly inadequate as an indicator of model validity. Taken together (however), they are a formidable filter (Richardson and Pugh, 1981).

Model experimentation

Figure 4 depicts the model's simulation of the DE-A software project. The model's results conformed quite accurately to the project's actual behavior (represented by the O points in the Figure).

The figure shows that project DE-A's management was inclined not to adjust the project’s “Estimated Schedule in Days” during most of the development phase of the project. Adjustments in the earlier phases of the project were made instead to the project's work-force level. This behavior is not atypical. It arises, according to DeMarco (1982), because of political reasons:

Once an original estimate is made, it's all too tempting to pass up subsequent opportunities to estimate by simple sticking with your previous numbers. This often happens even when you know your old estimates are substantially off. There are a few different possible explanations for this effect: It’s too early to show slip... If I re-estimate now, I risk having to do it again later (and looking bad twice) ... As you can see, all such reasons are political in nature.

The project's work-force pattern, on the other hand, does not conform to the typical pattern where the work-force level rises, peaks, and then drops back to lower levels as the project proceeds toward the system testing phase (Boehm, 1981). Because NASA's launch of the DE-A satellite was tied to the completion of the DE-A software, serious schedule slippages were not tolerated. Specifically, all software was required to be accepted and frozen 90 days before launch. As the deadline approached, pressures developed that overrode normal work-force stability considerations. That is, project management became increasingly willing to "pay any price" necessary to avoid overshooting the 90-day-before-launch date. Therefore, as Figure 4 indicates, management was increasingly willing to add more people. (Abdel-Hamid (1988) investigates whether such a staffing policy does or does not contribute to the project's late completion.)

The remaining parts of this section present a series of simulation experiments that investigate the tradeoffs between the economic costs and benefits of QA.

**Experiment 1: How Much QA**

Figure 5a plots the model's results of simulating the DE-A project using different QA expenditure levels. The figure shows the impact of different QA expenditure levels (defined as a percentage of total man-days) on the project's total cost in man-days. The optimal level of QA as a percentage of total development man-days is 15%. This result is based on the assumption that the QA effort is uniformly distributed throughout the project's lifecycle.

Two important conclusions can be drawn from Figure 5a. The first, more generalizable conclusion is that QA policy does have a significant impact on total project cost. As the figure shows, project cost ranges from a low of 1,648 man-days, to 5,650 man-days, i.e., a value that is approximately 3.5 times higher.

At low values of QA expenditures, the increase in cost results from the high cost of the testing phase. On the other hand, at high values of QA ex-
penditures, the excessive QA expenditures are themselves the culprit. The relationship between QA effort and the percentage of errors detected is shown in Figure 5b. Notice the "diminishing returns" of QA exhibited as QA expenditures extend beyond 10-15% of development effort. This type of behavior has been observed by others in the literature (Boehm, 1981; Shooman, 1983).

The second important conclusion concerns the 15% value for the optimal QA expenditure level. The significance of this result is not its particular value, since this cannot be generalized beyond the DE-A software project, but rather the process of deriving it (using this article’s integrative system dynamics simulation approach). Beyond controlled experimentation (which would be too costly and time-consuming to be practical), as far as the author knows, this model provides the first capability to quantitatively analyze the costs/benefits of QA policy for software production. It is encouraging to note that the model can be customized for

Figure 4. Simulation Run of DE-A Project
Figure 5a. Impact of Different QA Expenditure Levels on Project Cost

Figure 5b. Impact of Different QA Expenditure Levels on % of Errors Detected
different software development environments to derive environment-specific optimality conditions.

**Experiment 2: Distribution of the QA Effort**

As mentioned above, the optimal level for QA was derived under the assumption that QA effort was uniformly distributed. This is not necessarily the most effective distribution.

To identify a more cost-effective distribution, we started with the policy of 15% uniformly distributed QA effort. Areas in the project's lifecycle where such a level is not cost effective were then searched out. This was done by conducting simulation runs to test the impact of negative impulses in the QA level (see Figure 6). The effect of such an impulse is to decrease QA by 50% (i.e., from 15% to 7.5%) for a small interval of time. If such an impulse leads to a decrease in the project's total cost, then this would indicate that a QA level of 15% is too high at this point in the project's lifecycle, and vice versa. Figure 7 summarizes the results obtained from a series of simulation runs in which negative impulses were applied at different stages of the lifecycle. The results show that the simplistic uniform distribution policy under-spends on QA in the early phases of the project and over-spends in the middle and final stages.

By reiterating through the above experimentation strategy, the model can be used to derive a more cost-effective QA distribution. Such a refined distribution (called policy (R) here) is shown in Figure 8. This figure also shows the 15% uniform QA policy as well as the policy that was actually employed by NASA on the DE-A project. The impacts of these three QA policies on the project's overall cost are:

<table>
<thead>
<tr>
<th>QA Policy</th>
<th>Actual DE-A</th>
<th>15% Uniform Policy (R)</th>
</tr>
</thead>
<tbody>
<tr>
<td>QA Cost (man-days)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>524.0</td>
<td>203.6</td>
<td>161.9</td>
</tr>
<tr>
<td>Total Project Cost (man-days)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2,093.0</td>
<td>1,648.6</td>
<td>1,524.5</td>
</tr>
</tbody>
</table>

The cost improvements gained with the refined QA policy (R) are achieved largely by overspending on QA in the very early stages of the project. This allows early detection of design errors which, when left undetected, instigate many more errors in the later phases.

As Figure 8 shows, the QA level increases toward the end of the project. This proves to be cost effective because coding errors are relatively inexpensive to detect and correct. The QA effort at this stage is, therefore, highly productive. Further-

---

**Figure 6. Example Negative Impulse**
Figure 7. Where More/Less QA Would be More Cost Effective

Figure 8. Three QA Policies
more, if left undetected, such coding errors can cause additional errors in the system’s documentation and users’ manuals, which are typically quite expensive to rectify.

It is important to note that under the three different QA policies, the quality of the software product at the end of the project is assumed to be the same. When using the model, it is assumed that all errors not detected during the development stages of the project through QA will be detected and corrected at the system testing stage. Even though in practice some errors often remain in a software product after system testing is completed, all such errors are excluded from this analysis primarily because the generation, detection, and correction of such errors are maintenance-type issues that are beyond the boundary of this model. Additionally, errors that escape detection at the system testing phase are generally a small fraction of all the errors handled at that phase (Deutsch, 1979). This assertion may sound surprising to many, since it is commonly assumed that the maintenance activity is costly primarily because of the expense incurred in handling such lingering errors. Empirical results have shown, however, that corrections of such errors consume only a small portion of the software maintenance activity (Lientz and Swanson, 1981). The major portion of software maintenance is, instead, devoted to updating, enhancing, and perfecting the software system.

**Summary**

The QA function has, in recent years, become recognized as a critical factor in the successful development of software systems. However, because the utilization of QA tools and techniques does tend to add significantly to the cost of developing software, the cost-effectiveness of QA has been a significant concern to the software quality manager. As of yet this concern has not been adequately addressed in the literature.

The objective of this article was to investigate the tradeoffs between the economic benefits and costs of QA. To accomplish this, an integrative system dynamics model of the software development process was developed. The model is comprehensive because it integrates the multiple functions of the software development process, including both management-type functions and software production-type activities. The model also captures the dynamics of error generation as well as the QA activities of error detection and correction.

An important utility of the model is to serve as a laboratory vehicle to conduct controlled experiments on QA policy. Experimental results reported in this article show that QA policy has a significant impact on project costs. For the specific example analyzed, the optimal QA effort was 15% of the total development effort. Although this particular value applies only to this case-study project, the system dynamics-based simulation technique used can be adapted to model other software project environments.

**Acknowledgements**

The contributions of each of the individuals in the organizations providing perspectives and data to this research effort are greatly appreciated. I especially thank Mr. Frank E. McGarry of NASA’s Goddard Space Flight Center, for his time and effort in conducting the DE-A case study. My thanks are also extended to the anonymous reviewers, whose suggestions have improved this article’s readability. Work described herein was supported, in part, by NASA research grant NAGW-448.

**References**


About the Author

Tarek K. Abdel-Hamid is assistant professor of information systems in the Department of Administrative Sciences at the Naval Postgraduate School. Prior to joining NPS, he spent two and a half years at the Stanford Research Institute advising the senior management of multinational corporations on information systems issues. He has been an advisor to NASA's Jet Propulsion Lab since 1986, working on the development of computer-based tools for software project management. He received the B.S. degree in aeronautical engineering from Cairo University, Cairo, Egypt, in 1972, and a Ph.D. in management information systems from MIT in 1984. His research interests focus on software project management, system dynamics, expert simulators, and management information systems. He has authored or coauthored more than 20 papers and technical reports on these topics. His papers were published in journals such as the Communications of the ACM, Journal of Systems & Software, Information & Management, and IEEE Software, and he is the coauthor of a forthcoming Prentice-Hall book on software project management. Dr. Abdel-Hamid is a member of the ACM, SIM, and the IEEE-CS.

Appendix

QA Parameter Profile for the DE-A Software Project

Table equations represent a simple way of expressing relationships, particularly non-linear relations, between variables in a system dynamics model. Table equations have the following format:

\[ Y\text{-variable} = \text{TABLE} (\text{Table-name}, \text{X-variable}, \text{L}, \text{H}, \text{I}) \]

The above equation indicates a functional relationship between an independent X-variable and a dependent Y-variable. L, H, and I describe the low-end L, high-end H, and interval between points in a set of values of the independent X-variable. Table-name is the name of an associated table, or set of constant values, of the dependent Y-variable that corresponds to each of the values of the X-variable. Thus,

\[ Y = \text{TABLE} (\text{Table-1}, \text{X}, 0, 5, 1) \]

\[ \text{Table-1} = 3/7/9/11/13/14 \]

would represent the following functional relationship:

<table>
<thead>
<tr>
<th>X</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Y</td>
<td>3</td>
<td>7</td>
<td>9</td>
<td>11</td>
<td>13</td>
<td>14</td>
</tr>
</tbody>
</table>

Such table functions are used, as is shown below, to characterize the QA parameter profile of the DE-A project:

1. NOMINAL NUMBER OF ERRORS COMMITTED PER TASK (NERPK)

\[ \text{NERPK} = \text{TABLE} (\text{Table-1}, \text{"\% OF JOB WORKED"}, 0, 100, 20) \]

\[ \text{Table-1} = 24/22.9/20.75/15.25/13.1/12 \text{ ERRORS/KDSI} \]

2. NOMINAL QA MANPOWER NEEDED PER ERROR (NQAMPE)

\[ \text{NQAMPE} = \text{TABLE} (\text{Table-2}, \text{"\% OF JOB WORKED"}, 0, 100, 10) \]


\[ \text{Man-Days/ERROR} \]

3. NOMINAL REWORK MANPOWER NEEDED PER ERROR (NRWMPE)

\[ \text{NRWMPE} = \text{TABLE} (\text{Table-3}, \text{"\% OF JOB WORKED"}, 0, 100, 20) \]

\[ \text{Table-3} = .6/.575/.5/.4/.325/.3 \text{ Man-Days/ERROR} \]