WORKING PAPER
ALFRED P. SLOAN SCHOOL OF MANAGEMENT

THE ELUSIVE SILVER LINING: HOW WE FAIL TO LEARN FROM FAILURE IN SOFTWARE DEVELOPMENT

Tarek K. Abdel-Hamid Stuart E. Madnick

Sloan School of Management WP # 3180-90-MS
May 31, 1990

MASSACHUSETTS
INSTITUTE OF TECHNOLOGY
50 MEMORIAL DRIVE
CAMBRIDGE, MASSACHUSETTS 02139

THE ELUSIVE SILVER LINING: HOW WE FAIL TO LEARN FROM FAILURE IN SOFTWARE DEVELOPMENT

Tarek K. Abdel-Hamid Stuart E. Madnick
Sloan School of Management WP # 3180-90-MS
May 31, 1990

THE ELUSIVE SILVER LINING:

HOW WE FAIL TO LEARN FROM FAILURE IN SOFTWARE DEVELOPMENT

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

Stuart E. Madnick
Center for Information Systems Research
Sloan School of Management
Massachusetts Institute of Technology
Cambridge, MA 02139

May 31, 1990

Accepted for publication to the Sloan Management Review

THE ELUSIVE SILVER LINING:

HOW WE FAIL TO LEARN FROM FAILURE IN SOFTWARE DEVELOPMENT

ABSTRACT

As modern organizations struggle with increasing complexity and difficult challenges, we argue that every experience --- win, lose, or draw --- is a valuable organizational asset that must be fully exploited. First, this requires an organization willing to view failures as opportunities to learn something rather than as embarrassing moments to be quickly forgotten. Second, and often missing, organizations need a formal postmortem diagnostic tool that can reliably discern what worked on a project and what did not. Without such a tool, it is possible that the wrong lessons will be learned.

We describe our development of and experiences with a system dynamics based tool for postmortem project diagnosis. In this paper we focus our attention on software development projects since they have become critical to the successful operation and strategy of many organizations, yet have experienced serious problems. As an example, when applied to a NASA software development project, three types of insights were gained. First, we show how intuition alone may not be sufficient to handle the complex and dynamic interactions characterizing the software project environment, and may indeed mislead us in deriving the wrong lesson regarding staffing policy. Second, we show how the experimentation capability of the model may be utilized to derive additional learning "dividends" from what continues to be a seriously under-exploited software project lesson regarding undersizing. Third, the model uncovers a well-disguised lesson, a case where too much of a good thing, such as quality assurance, was bad.

INTRODUCTION

There is a silver lining to every failure. For it is only through costly experience and errors that managers can develop effective intuitive judgement. "Good judgement is usually the result of experience. And experience is frequently the result of bad judgement" (Neustadt and May, 1986). And, not unlike sticking one's hand into the fire, what we learn from our mistakes is often more indelible.

Why do we fail to learn from project failures? There are at least two reasons. First, we rarely try. Generally, mistakes are hidden rather than reported and evaluated. Second,

and often missed, the important lessons to be learned are rarely conveniently packaged for easy picking, rather they often need to be dug out from deep within the project experience. A primary objective of this article is to show how and why the silver lining often eludes us. To illustrate this we will focus on the information technology field and use a case study of a real software project.

Software technology is playing an ever larger role in formulating business strategy, in determining how an organization operates, how it creates its products, and indeed in reshaping the product itself [(Porter and Millar, 1985) and (Bales, 1988)]. Yet, to the dismay of not only system professionals but business executives as well, this formidable dependance on software technology has not been matched by a corresponding maturity in the capability to manage it. We continue to produce too many project failures, marked by cost overruns, late deliveries, poor reliability, and users' dissatisfaction (Newport, 1986).

Failure to learn from mistakes has been a major obstacle to improving software project management:

> We talk about software engineering but reject one of the most basic engineering practices: identifying and learning from our mistakes. Errors made while building one system appear in the next one. What we need to remember is the attention given to failures in the more established branches of engineering (Boddie, 1987).

We will demonstrate the utility of a system dynamics based tool for conducting a postmortem diagnostic analysis of what worked on a software project and what did not.

THE CHALLENGE OF SOFTWARE DEVELOPMENT

One measure of the impact of software is on the pocketbook. It has been estimated that U.S. expenditures for software development and maintenance will grow to more than $225 billion by 1995 in the U.S. and more than $450 billion worldwide (Boehm, 1987).

This growth in demand for software has not, however, been painless. The record shows that the software industry has been marked by cost overruns, late deliveries, poor reliability, and users' dissatisfaction, collectively referred to as the "software crisis".

As early as November 9, 1979, a report to Congress by the Comptroller General cited the dimensions of the "software crisis" within the federal government. The report's title summarizes the issue: "Contracting for Computer Software Development --- Serious Problems Require Management Attention to Avoid Wasting Additional Millions." The report concludes, "The government got for its money less than 2 percent of the total value of the contracts."

More than a decade later, the problems persisted . An article in the December 18, 1989 issue of *Defense News* described the software problems with the Peace Shield project which was then four years behind schedule and estimated to be up to $300 million over budget (Baker and Silverberg, 1989).

Big as the direct costs of the"software crisis" are, the indirect costs can be even bigger, because software is often on the critical path in overall system development. That is, any slippages in the software schedule translate directly into slippages in the overall delivery schedule of the system (e.g. a new approach to customer order entry or manufacturing scheduling).

Although many of the largest and most completely documented examples are found in military projects, the "software crisis" is by no means confined to projects developed by or for the federal government. It is similarly prevalent within private sector organizations (Zmud, 1980). For example, DeMarco noted:

- Fifteen percent of all software projects never deliver anything; that is, they fail utterly to achieve their established goals.

- Overruns of one hundred to two hundred percent are common in software projects.

So many software projects fail in some major way that we have had to redefine "Success" to keep everyone from becoming despondent. Software projects are sometimes considered successful when the overruns are held to thirty percent or

> when the user only junks a quarter of the result. Software people are often willing to call such efforts successes, but members of our user community are less forgiving. They know failure when they see it (DeMarco, 1987).

Personal computer software development is not immune to these problems either. The headline in the May 11, 1990 issue of *The Wall Street Journal* says it explicitly: "Creating New Software was Agonizing Task for Mitch Kapor Firm" (Carroll, 1990). The article was sub-titled: "Despite Expert's Experience Job Repeatedly Overran Time and Cost Forecasts." The issues cited in that article (e.g., "... programmers spend 90% of their time on the first 80% of a project and 90% of their time on the final 20%") have been repeated for decades in both public and private organizations, large and small companies (Kapor's company had less than 30 people), and computers from mainframes to PC's.

Due to the embarrassment and bad publicity associated with such problems, it is likely that only a small portion are ever publicly reported. Mitch Kapor, the founder of Lotus Development Corporation, agreed to describe the experiences in his new company ON Technology Inc. "because he believes that software design must be improved and the development process better understood." (Carroll, 1990) That nicely sums up a goal of this paper. Disclosing the failure is only the first step, learning from the failure is the critical next step.

A CASE-STUDY

Consider the case of NASA's DE-A software project. The project was conducted at the Systems Development Section of the Goddard Space Flight Center (GSFC) in Greenbelt, Maryland to design, implement, and test a software system for processing telemetry data and providing attitude determination and control for the DE-A satellite. In planning and managing this project, approximately 85% of the total project cost was allocated to development (design and coding) and the remaining 15% to testing. Furthermore, a relatively large portion (30%) of the development effort was allocated to

quality assurance (QA), a level that is significantly higher than the industry norm (Boehm, 1981).

Initially, the project was estimated to be 16,000 delivered source instructions (DSI) in size, and its cost and schedule were estimated to be 1,100 man-days and 320 working days, respectively. Figure 1 depicts the values of these and other DE-A project variables over the duration of the project. Because NASA's launch of the DE-A satellite was tied to the completion of the DE-A software, serious schedule slippages could not be tolerated. Specifically, all software was required to be accepted and frozen three months before launch. As the project slipped and this date approached, management reacted by adding new people to the project to meet the strict launch deadline ... as evidenced by the rising workforce pattern curve in the final stages of the project seen in Figure 1. The actual final results were: 24,400 DSI, 2,200 man-days, and 380 days.

Compared with its original estimates, obviously the DE-A project is not a total success. The project overshot its schedule by 20% and its cost by 100%. On the positive side, the end product was reported to be of high quality i.e., was reliable, stable, and easy to maintain (NASA, 1983).

Relying upon "conventional wisdom", what lessons might NASA have learned from the DE-A project?

Staffing. Obviously, the DE-A staffing policy of continuing to add people late into the project is not cost effective. Indeed, Brooks (1975) suggests that by adding new people to the late DE-A project, management actually delayed it further! Thus, in the future they should limit hiring to the early phases only.

Undersizing. Another obvious culprit is the initial 35% underestimation of the product's size. Schedule estimation models are garbage in-garbage out devices: when poor sizing data is input in one side, poor schedule estimates come out the other side. On the DE-A project, an initial 35% underestimation of project size led to an underestimate of the project's man-day and time requirements. Thus, in the future if faced with a similar project,

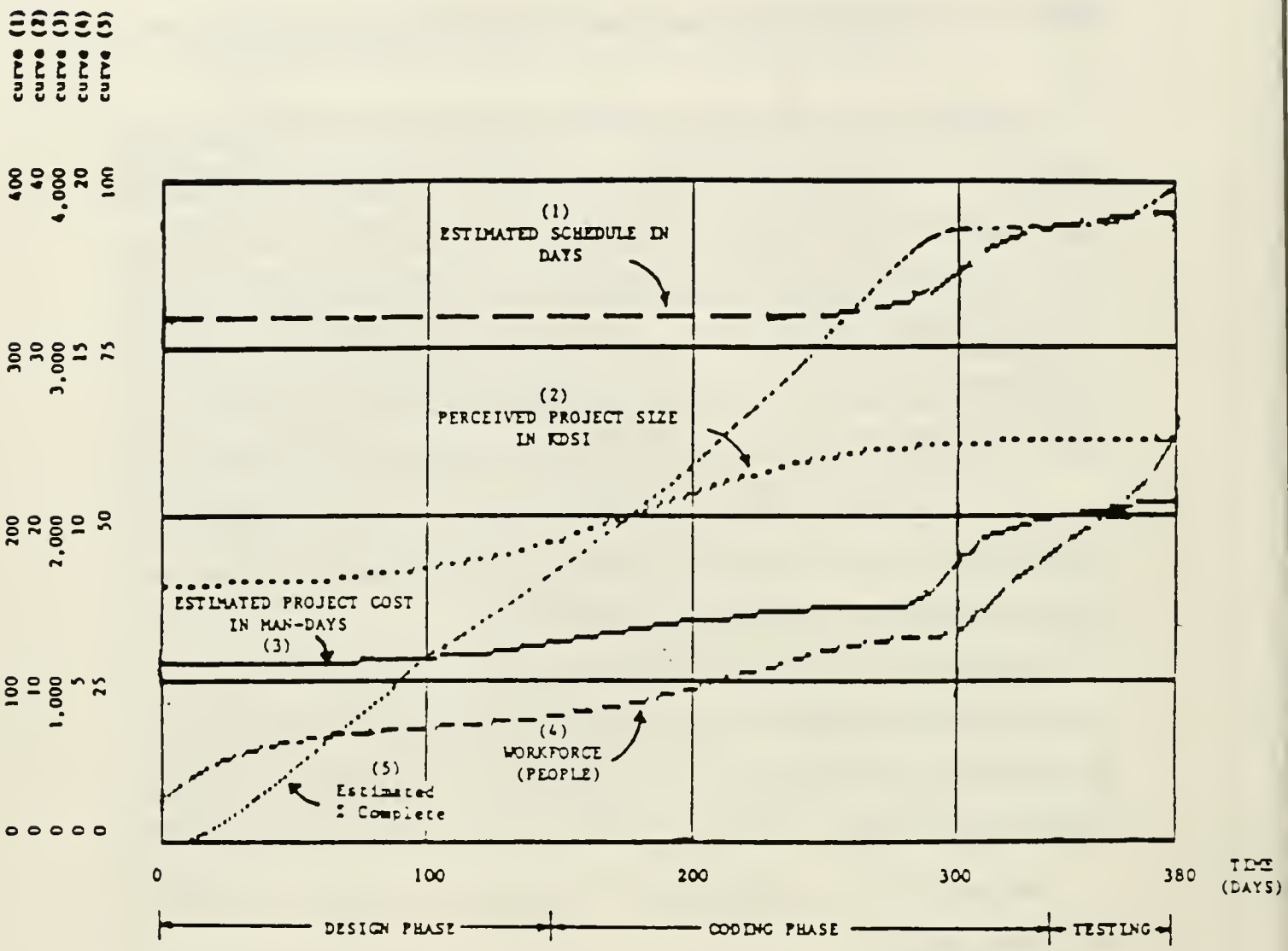


FIGURE 1
THE BEHAVIOR OVER TIME OF KEY DE-A PROJECT VARIABLES

the budgeted project size, man-days, and time requirements should be all increased to match the actual results of this project.

Quality Assurance. We also want to learn from success as well as from failure. The high level of quality assurance activity appears to have been a favorable policy and should be continued in the future.

Evaluation of "Conventional Wisdom". The reader of this paper may or may not have suggested the three specific lessons listed above. The point is rather that professionals in the field and other published articles advocate such views. In the remainder of this paper we will elaborate upon the normal logic behind these views, a new systematic way to study these issues, and the deficiencies in relying upon traditional, and simplistic, "conventional wisdom".

ARGUMENTS FOR A FORMAL DIAGNOSTIC TOOL

Did in fact DE-A's aggressive staffing policy contribute to the project's schedule delays? To most students of the DE-A project, the answer is: "Yes ... obviously." Brooks' law, which states that adding manpower to a late software project makes it later, is intuitively quite palatable since we know that new hires are not immediately productive and, furthermore, consume considerable time of the experienced people to gain training and familiarity with the project. Since its publication, Brooks' law has been widely endorsed in the literature for all kinds of project environments e.g., systems programming-type projects as well as applications-type projects, both large and small (Pressman, 1982). This, in spite the fact that it has not been formally tested.

In (Abdel-Hamid, 1989a) we tested the applicability of Brooks law to the specific DE-A project. Our results indicate that while adding people late in the lifecycle did cause the project to become more costly it did not, however, cause it to be completed later. A result that proved unsettling to many.

A closer look, however, reveals that our result is not necessarily as counter-intuitive as it might first appear. In fact, Brooks (1975) made no claim to the universality of Brooks' law. On the contrary, Brooks was quite explicit in specifying the domain of applicability of his insights, namely, to what he called "jumbo systems programming projects." Such projects are significantly more complex to develop and manage than are smaller application-type projects (like the DE-A project).

The cost increases that projects (such as DE-A) often incur when staff size is increased is a result of the decrease in the team's average productivity and which in turn is a result of the increase in the training and communication overheads on the project. However, for a project's schedule to also suffer from a staff increase the drop in productivity must be large enough to render each additional person's net cumulative contribution to, in effect, be a negative contribution. We need to calculate the net contribution because an additional person's contribution to useful project work must be balanced against the losses incurred as a result of diverting experienced staff members from direct project work to the training of and communicating with the new staff member(s). And we need to calculate the cumulative contribution because while a new hiree's net contribution might be negative initially, as training takes place and the new hiree's productivity increases, the net contribution becomes less and less negative, and eventually (given enough experience on the project) the new person starts contributing positively to the project. Only when the net cumulative impact is a negative one will the addition of the new staff member(s) translate into a longer project completion time.

Obviously, the earlier in the lifecycle that people are added and/or the shorter the training period needed the more likely that the net cumulative contribution will turn positive. Both such conditions did apply in the DE-A project. Notice from Figure 1 that the steep rise in the workforce level really commenced early in the coding phase. Also, the training and assimilation period needed to bring new hirees up to speed was relatively low on the DE-A project. There are two reasons for this. First, the software developed was

similar to the telemetry software developed by the GSFC organization for previous NASA satellites. The second reason can be attributed to a task order arrangement that NASA had with the Computer Sciences Corporation (CSC), through which a pool of CSC software professionals was made available for recurrent work assignments on NASA's projects. This tapped pool of software professionals has, over the years, gained a lot of experience with the NASA project environment, and as a result, when recruited on a new project, they are brought up to speed relatively quickly.

The above observations, then, demonstrate the real dangers of over-generalizing in software project management. Specifically, the danger here is in deriving the wrong lesson from a DE-A postmortem analysis of staffing policy i.e., in concluding that DE-A's particularly aggressive staffing policy did contribute to the project's schedule delays (when it did not). But, more generally, our results suggest that the impact of adding staff to a late software project will depend on the particular characteristics of the project, the staff, and when in the lifecycle the staff additions are made. Because these project characteristics do dynamically interact in a complex non-linear fashion (Abdel-Hamid, 1989a), relying on intuition alone to do all the necessary bookkeeping can be perilous. The human mind, studies have shown, is simply not adapted to reliably trace through time the implications of such a complex set of interacting factors (Richardson and Pugh, 1981).

Engineers turn to laboratory experiments to understand the behavior of complex engineering systems. Why, then, do we not use the same approach of making models of social systems and conducting laboratory experiments on those models? Controlled laboratory experiments on managerial systems are indeed possible with computers that can simulate social systems:

> Model experimentation is now possible to fill the gap where our judgement and knowledge are weakest --- by showing the way in which the known separate system parts can interact to produce unexpected and troublesome overall system results... The manager, like the engineer, can now have a laboratory in which he can learn quickly and at low cost the answers that would seldom be obtainable from trials on real organizations... Circumstances can be studied that

might seem risky to try with an actual company can be investigated (Forrester, 1961).

Like the microscope and the telescope did in a previous age, the computer is opening up a new window on reality (Pagels, 1988). In the next section a system dynamics based model of the software development process is described, and later used as a laboratory vehicle for analyzing the DE-A software project experience.

A SYSTEM DYNAMICS MODEL OF SOFTWARE PROJECT MANAGEMENT

A comprehensive system dynamics simulation model of software development has been developed as part of a wide-ranging study of the software development process (Abdel-Hamid, 1984). 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 2 depicts a highly aggregated view of the model's four subsystems, namely: (1) Human Resource Management; (2) Software Production; (3) Control; and (4) Planning. The figure also illustrates some of the interrelations between the four subsystems.

The Human Resource Management Subsystem captures the hiring, training, assimilation, and transfer of the project's human resource. Such actions are not carried out in vacuum, but, as Figure 2 suggests, they are affected by the other subsystems. For example, the project's hiring rate is a function of the workforce level needed to complete the project on a certain planned completion date. Similarly, what workforce is available has direct bearing on the allocation of manpower among the different software production activities in the Software Production Subsystem.

The four primary activities in the Software Production Subsystem are: development, quality assurance, rework, and testing. The development activity comprises both the design and coding of the software. As the software is developed, it is also reviewed, e.g., using structured walkthroughs, to detect any errors. Errors detected

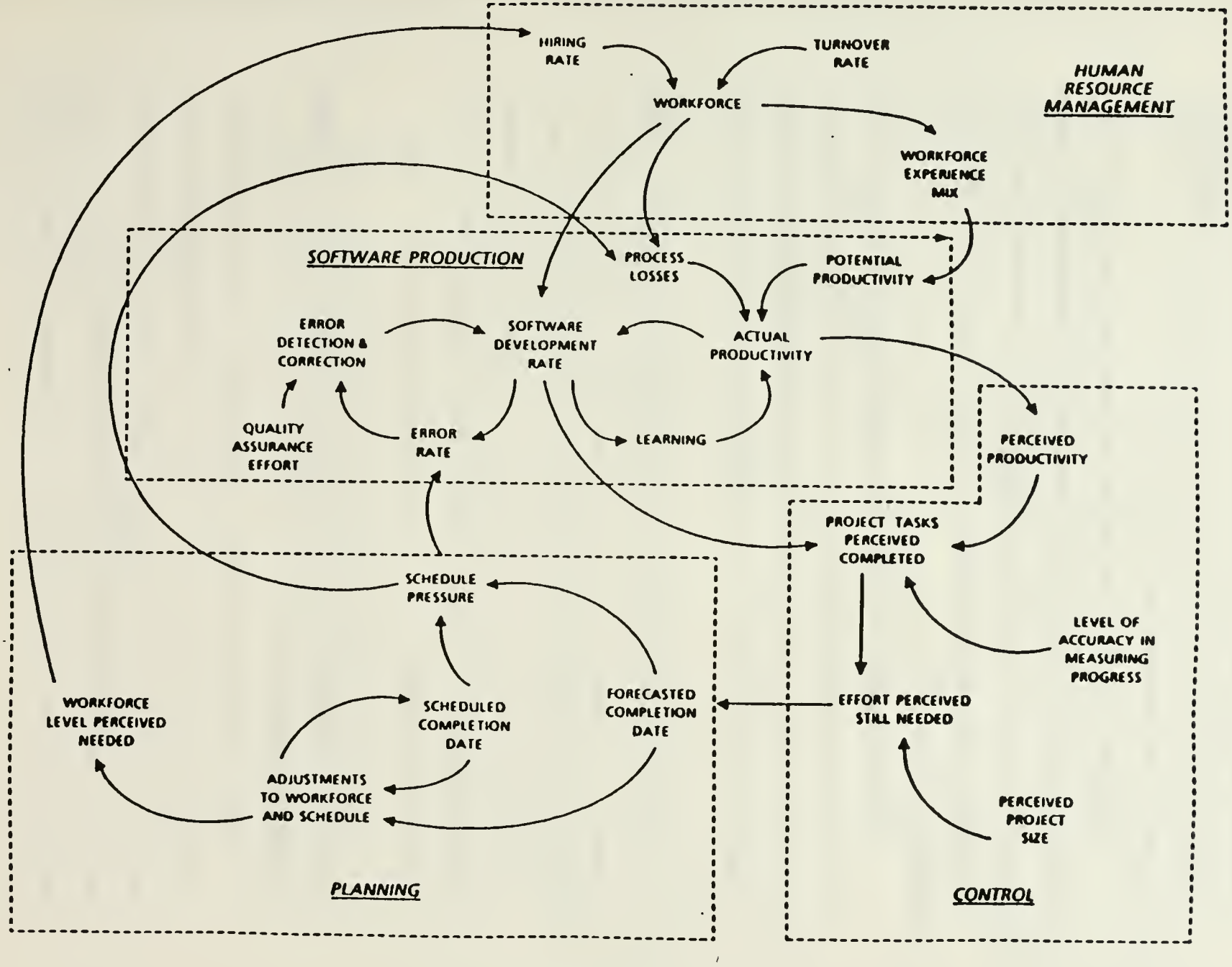


FIGURE 2
OVERVIEW OF MODEL STRUCTURE

through such quality assurance activities are then reworked. Not all errors get detected and reworked at this phase, however. Some "escape" detection until the end of development e.g., until the system testing phase.

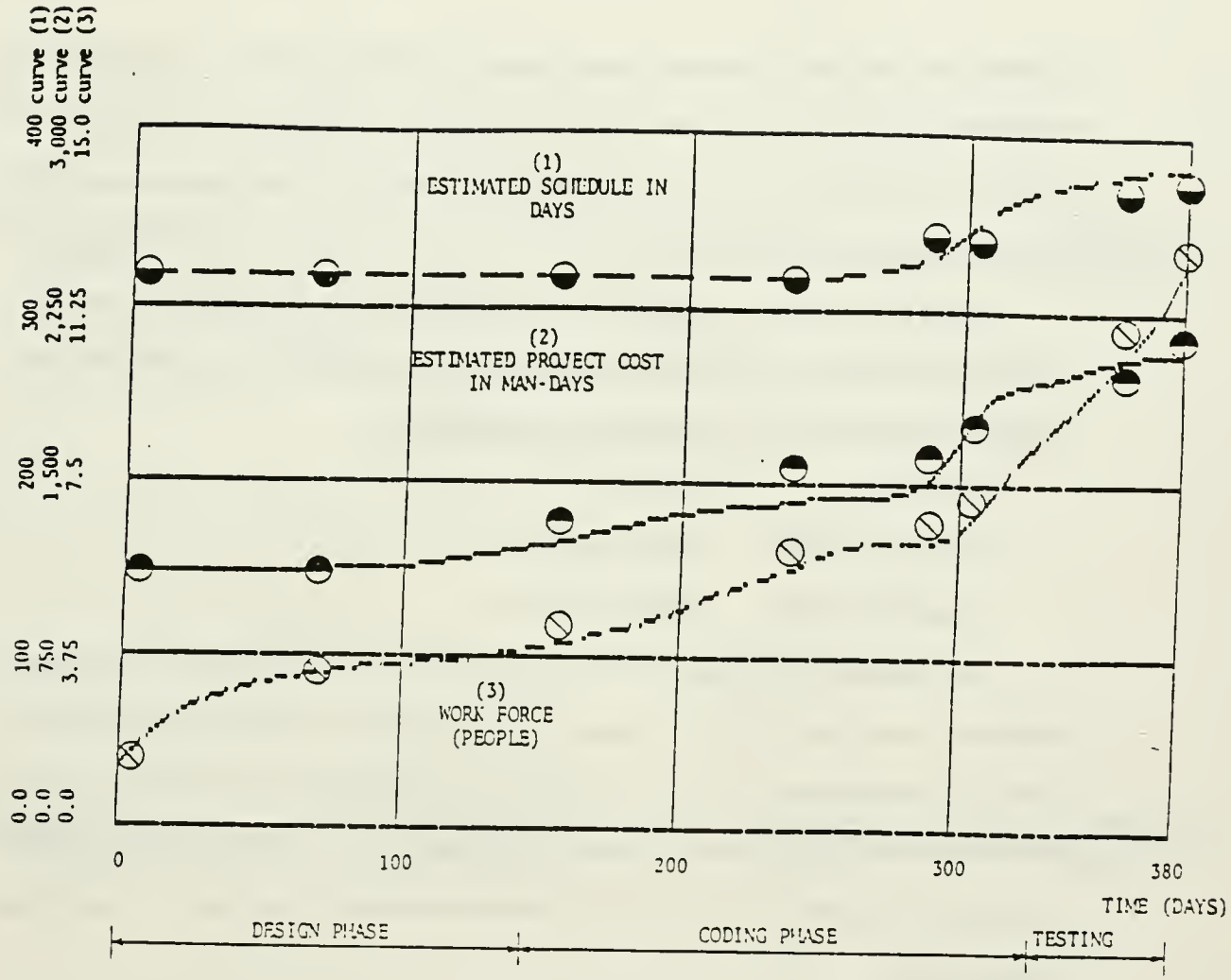
As progress is made on the software production activities, it is reported. A comparison of where the project is versus where it should be according to plan is a control-type activity captured within the Control Subsystem. Once an assessment of the project's status is made using available information, it becomes an important input to the planning function.

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

A full discussion of the model's structure, its mathematical formulation, and its validation is available in other reports [(Abdel-Hamid, 1984), (Abdel-Hamid, 1989a), and (Abdel-Hamid and Madnick, 1990)]. The DE-A project case-study, which was conducted at NASA after the model was completely developed, constituted an important element in validating model behavior. In Figure 3, actual DE-A project results are compared with the model's simulation output.

PROJECT UNDERESTIMATION REVISITED: AN OVER-USED BUT UNDER-EXPLOITED LESSON

While undersizing is obviously a serious problem in software development, identifying it as the culprit may not be very helpful to the practicing software manager, and indeed it may even be harmful. First, it is a problem that the software manager might not be able to avoid. Second, there is a strong temptation to embrace undersizing as a convenient scapegoat for all the project's difficulties. Such rationalization can only "numb"



DE-A's actual ESTIMATED SCHEDULE IN DAYS

DE-A's actual ESTIMATED PROJECT COST IN MAN-DAYS

DE-A's actual WORK FORCE (Full-time equivalent people)

FIGURE 3
ACTUAL VERSUS SIMULATED PROJECT VALUES

organizational curiosity, depriving management of the opportunity to appreciate the multitude of project factors that can and do contribute to cost/budget overruns. In the next section, we will see how other (not so obvious) factors did in fact contribute to DE-A's cost and schedule overrun problems.

While the industry's predisposition to exaggerate the sins of undersizing may not be a shocking revelation, that organizations consistently under-exploit the undersizing lesson should be. Recall that DE-A's final results were as follows:

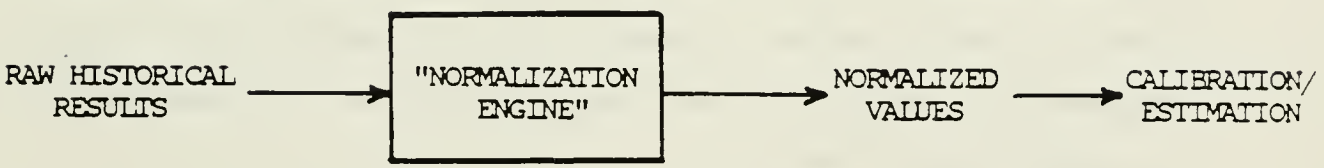
- project size: 24,400 DSI
- development cost: 2,200 man-days
- completion time: 380 working days

The standard procedure in the industry (as well as at NASA) is to directly incorporate project results such as the above into a database of historical project statistics to support the development, calibration, and fine-tuning of software estimation tools as depicted in Figure 4a. The underlying assumption here is that such project results constitute the most preferred and reliable benchmark for future estimation purposes ... after all they are actual values.

However, from our earlier discussion of the DE-A project history we must suspect that its final cost of 2,200 man-days may not be a desirable benchmark for future estimates. The reason being that such a value reflects the inefficiencies incurred in the staffing of the project, and which in turn were a result of its initial undersizing. Thus, if a new project comes along that happens to be exactly similar to DE-A, and if we assume that its size will be properly estimated at the start (as we assume when estimating for any new project), then a more effective staffing plan would be devised that avoids DE-A's last-minute staff explosion. As a result the new project should require less than 2,200 man-days to accomplish.

Notice, we used the word should and not would. For, if DE-A's (inflated) 2,200 man-days value were in fact adopted as the benchmark for estimating the new 24,400 DSI

(a) Current Practice



(b) Proposed Normalization Strategy

FIGURE 4

project, savings due to better staffing may indeed not be realized. The reason: the self-fulfilling prophecy of Parkinson's law. Work on a software project can expand in many different forms to fill the available time. For example, work expansion could take the form of goldplating (e.g., adding features to the software product which make the job bigger and more expensive, but which provide little utility to the user or maintainer when put into practice), or it could be in the form of an increase in people's slack time activities (Boehm, 1981).

What is needed is a strategy that allows us to fully capitalize on DE-A's learning experience by "wringing" out those man-day excesses caused due to undersizing, to derive an ex-post set of normalized cost and schedule estimation benchmarks (see Figure 4b). The system dynamics model developed for this study provides a viable tool for such a task. In addition to permitting less costly and less time-consuming experimentation, it makes "perfectly" controlled experimentation possible where "... all conditions but one can be be held constant and a particular time-history repeated to see the effect of the one condition that was changed" (Forrester, 1961).

Specifically, the strategy involves the re-simulation of the DE-A project with no undersizing. In order to determine the extent of the man-day excesses not one but several simulation runs were conducted. In all runs the initial schedule estimate was held constant at 380 days, while the man-day estimate was gradually decreased to lower values. The results of such an experiment are shown in Figure 5. The X-axis depicts the different initial man-day estimates, while the Y-axis depicts the project's final (simulated) cost in man-days.

The results indicate that using DE-A's raw value of 2,200 man-days is indeed wasteful. As the initial man-day estimate for the project is gradually lowered, savings are achieved as wasteful project practices such as goldplating, unproductive slack time activities, are gradually shrunk. This continues until the 1,900 man-day level is reached. Lowering the project's initial man-day estimate below this point, however, becomes

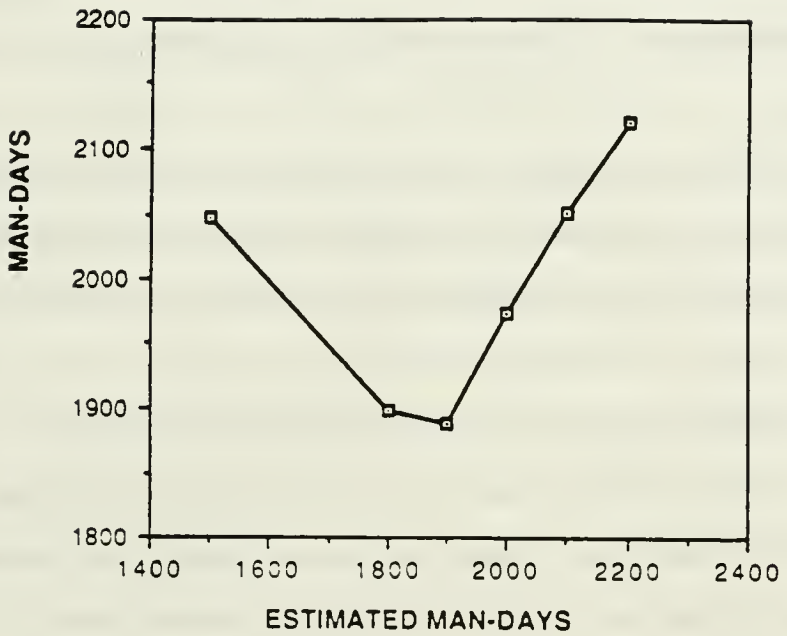


FIGURE 5
IMPCAT OF INITIAL ESTIMATES ON PROJECT COST

counter-productive, as the project not only sheds all its excess, but becomes in effect an underestimated project. Initial underestimation is costly (whether it is is due to initial undersizing or not), as it leads to an initial understaffing, followed by a costly staff buildup later in the lifecycle.

The above results clearly indicate that the widely-held notion that raw historical project results constitute the most preferred benchmark for future estimation is not only flawed, but can be costly as well. In the particular case of NASA's DE-A project, a 1,900 man-day value is clearly a more preferred benchmark for inclusion in the normalized database of historical project results over DE-A's raw 2,200 value, as it would save NASA 234 man-days --- a 10.6% saving in cost. And in view of the pervasiveness of the undersizing problem in the software industry, we can only suspect that the potential for such savings is not only realizable, but indeed abounds in the software industry.

A SENSE OF FALSE SECURITY?

In the previous sections our focus has been on problems, their causes, and the lessons we derive from them. Our focus here, on the other hand, is on "success" and the lessons we can gain "in spite of it". Such a quest is rarely, if ever, undertaken. "March and Simon (1958) designated dissatisfaction as the major trigger of organizational problem solving. Organizations begin to search when problems are discovered or when gaps between performance and expectations become large enough" (Hedberg, 1981). When, on the other hand, goals are accomplished the system often deludes us into a sense of false security which may not always be justifiable or wise. For it may breed complacency and possibly even reinforce dysfunctional behavior.

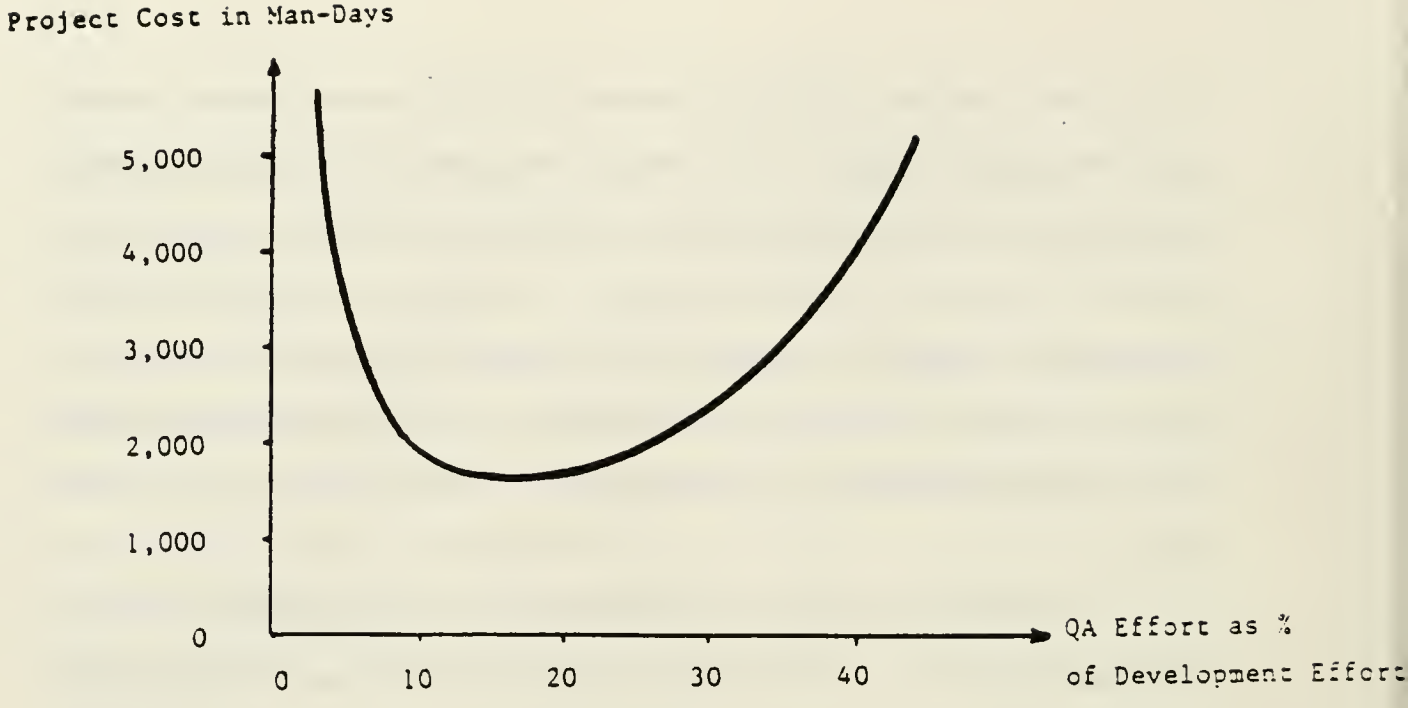
As was mentioned above, DE-A's end product was reported to be of high quality. This was attained through an aggressive quality assurance (QA) policy in which as much as 30% of the development effort was allocated to QA activities. Was such a high level of QA effort cost-effective?

Such a question is difficult to answer. In principle, we could conduct a real life experiment in which the DE-A project is repeated many times under varied QA expenditure levels. Such an experimental approach, however, is too costly and time consuming to be practical. Furthermore, even when affordable, the isolation of the effect (cost) and the evaluation of the impact of any given practice (QA) within a large, complex, and dynamic social system such as a software project environment can be exceedingly difficult (Glass, 1982). Simulation modeling, on the other hand, does provide a viable alternative for such a task.

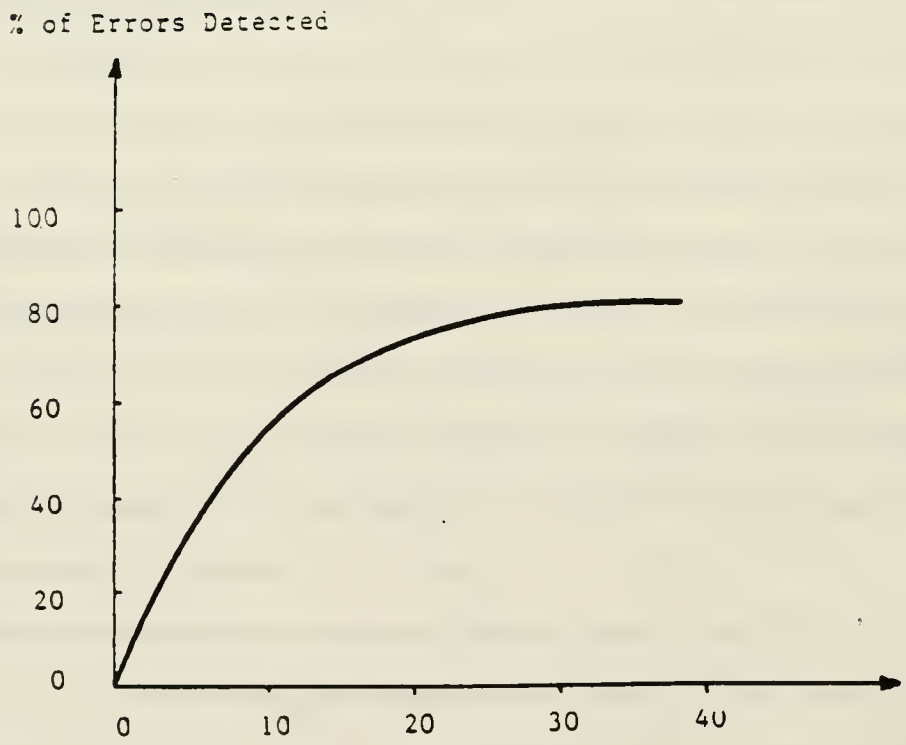
In (Abdel-Hamid, 1988b) our model was used to investigate the impact of different QA expenditure levels on the DE-A project. Figure 6a plots the impact of QA expenditures (defined as a percentage of total man-days) on the project's total cost. At low values of QA, the increase in total project cost results from the high cost of the testing phase. On the other hand, at high values of QA expenditures, the excessive QA effort is itself the culprit. The reason: as QA expenditures increase beyond the 15-20% level, "diminishing returns" are experienced as shown in Figure 6b. Such behavior is not atypical. "In any sizable program, it is impossible to remove all errors (during development) ... some errors manifest themselves, and can be exhibited only after system integration" (Shooman, 1983).

It is quite clear from Figure 6a that the QA expenditure level has a significant influence on the total cost of the project. Specifically, DE-A's cost ranged from a low of 1,648 man-days to a high of 5,650 man-days over the range of QA policies tested. It is also obvious that DE-A's actual 30% QA allocation level is sub-optimal, yielding a total project cost of 2,200 man-days that is 35% higher than the optimal of 1,648 man-days. Notice that the latter could have been achieved with only a 15% QA allocation. (It is important to note here that under the different QA policies tested, the quality of the software end product was controlled to remain the same through increased testing phase activities.)

The significance of the above result is not in deriving the particular optimal QA allocation level, since this cannot be generalized beyond the specific DE-A software project,



(a) IMPACT OF DIFFERENT QA EXPENDITURE LEVELS ON PROJECT COST



(b) IMPACT OF DIFFERENT QA EXPENDITURE LEVELS ON % OF ERRORS DETECTED

FIGURE 6

but rather it is in the process of deriving it using the system dynamics simulation modeling approach. Beyond controlled experimentation (which would be too costly and time-consuming to be practical), as far as the authors know, this model provides the first capability to quantitatively analyze the economics of QA policy. In this case it allowed us to discern a problem of over-spending which would have otherwise been effectively disguised under the consuming sense of self-congratulation in having achieved the project's quality goals. When such inefficiencies are not detected and quickly corrected they tend, overtime, to be institutionalized into organizational "fat" that is then very difficult and often painful to shed.

CONCLUSIONS

In this article we proposed a system dynamics based tool for increasing the effectiveness of postmortem software project diagnosis. When applied to a NASA project, the exercise produced three types of insights. First, we showed how intuition alone may not be sufficient to handle the complex and dynamic interactions characterizing the software project environment, and may indeed mislead us in deriving the wrong lesson (Brooks' law). Second, we showed how the experimentation capability of the model may be utilized to derive additional learning "dividends" from what continues to be a seriously under-exploited software project lesson (undersizing). And third, the model uncovered a disguised lesson, a case where too much of a good thing was bad (QA).

In general, without an effective postmortem diagnostic exercise to identify problems and their causes, project deficiencies will not receive special scrutiny and may be repeated on future projects. Errors made in one system will appear in the next one. On the other hand, the payoff from an effective postmortem is a smarter organization that truly learns from its failures.

BIBLIOGRAPHY

1. Abdel-Hamid, T.K. "The Dynamics of Software Development Project Management: An Integrative System Dynamics Perspective." Unpublished Ph.D. dissertation, Sloan School of Management, MIT, January, 1984.

2. Abdel-Hamid, T.K. "Understanding the '90% Syndrome' in Software Project Management: A Simulation-Based Case Study." Journal of Systems & Software, September, 1988a.

3. Abdel-Hamid, T.K. "The Economics of Software Quality Assurance: A Simulation-Based Case Study." MIS Quarterly, September, 1988b.

4. Abdel-Hamid, T.K. "The Dynamics of Software Project Staffing: A System Dynamics Based Simulation Approach." IEEE Transactions on Software Engineering, February, 1989a.

5. Abdel-Hamid, T.K. "Investigating the Cost-Schedule Tradeoff in Software Development." IEEE Software, forthcoming, 1989b.

6. Abdel-Hamid, T.K. and Madnick, S.E. "Software Productivity: Potential Actual, and Perceived." System Dynamics Review, Vol. 5, No. 2, Summer 1989.

7. Abdel-Hamid, T.K. and Madnick, S.E. The Dynamics of Software Development, Englewood Cliffs, New Jersey: Prentice-Hall, Inc., forthcoming, 1990.

8. Baber, R.L. Software Reflected, New York: North Holland Publishing Company, 1982.

9. Baker, C. and Silverberg, D., Defense News, Vol. 4, No. 51, December 18, 1989.

10. Bales, C.F. "The Myths and Realities of Competitive Advantage." Datamation, October 1, 1988.

11. Boddie, J. "The Project Postmortem." Computerworld, December 7, 1987.

12. Boehm, B.W. Software Engineering Economics, Englewood Cliffs, New Jersey:Prentice-Hall, Inc., 1981.

13. Boehm, B.W., "Improving Software Productivity." Computer, September, 1987.

14. Brooks, F.P. The Mythical Man Month, Reading, Mass: Addison-Wesley Publishing Co., 1975.

15. Carroll, P.B., "Creating New Software Was Agonizing Task for Mitch Kapor Firm." The Wall Street Journal, Vol. CCXV, No. 93, May 11, 1990.

16. Cleland, D.I. and King, W.R. Systems Analysis and Project Management, New York: McGraw-Hill, 1975.

17. DeMarco, T. Controlling Software Projects, New York: Yourdon Press, Inc., 1982.

18. Forrester, J.W. Industrial Dynamics, Cambridge, Mass: The MIT Press, 1961.

19. Forrester, J.W. "Counterintuitive Behavior of Social Systems." Technology Review, Vol. 73, No. 3, January, 1971.

20. Glass, R.L. The Universal Elixir and other Computing Projects which Failed. Seattle. Washington: R.L. Glass, 1977.

21. Hedberg, B. "How Organizations learn and Unlearn." In Handbook of Organizational Design. Volume I, edited by P. C. Nystrom and W.H. Starbuck. Oxford, England: Oxford University Press, 1981.

22. McGowan, C.L. and McHenry, R.C. "Software Management." In Research Directions in Software Technology. by P. Wegner (ed.). Cambridge, Massachusetts: The MIT Press, 1980.

23. Mills, H.D. Software Productivity. Canada: Little, Brown & Co., 1983.

24. NASA. "Software Development history for Dynamic Explorer (DE) Attitude Group Support System." NASA/GSFC CODE 580, June, 1983.

25. Neustadt, R.E. and May, E.R. Thinking in Time. New York: The Free Press, 1986.

26. Newport, John P. Jr. "A Growing Gap in Software." Fortune, April 28, 1986, 132-142.

27. Pagels, H. The Computer and the Rise of the Sciences of Complexity. New York: Simon and Schuster, 1988.

28. Porter, M.E. and Miller, V. "How Information gives you Competitive Advantage." Harvard Business Review, July/August, 1985.

29. Pressman, R.S. Software Engineering: A Practitioner's Approach. New York: McGraw-Hill, Inc., 1982.

30. Ramamoorthy, C.V. et al. "Software Engineering: Problems and Perspectives." Computer, Oct., 1984.

31. Richardson, G.P. and Pugh, G.L. III. Introduction to System Dynamics Modeling with Dynamo. Cambridge, Mass: The MIT Press, 1981.

32. Shooman, M.L. Software Engineering - Design, Reliability and Management.. New York: McGraw-Hill, Inc., 1983.

33. Zmud, R.W., "Management of Large Software Development Efforts." MIS Quarterly, Vol. 4, No. 2, June, 1980.