LESSONS LEARNED FROM MODELING THE DYNAMICS OF SOFTWARE DEVELOPMENT

Tarek K. Abdel-Hamid
Stuart E. Madnick

August 1988

CISR WP No. 184
Sloan WP No. 2069-88

©1988 T.K. Abdel-Hamid, S.E. Madnick

This paper is a companion piece to CISR WP No. 163, "Modeling the Dynamics of Software Project Management". Whereas WP 163 discusses the model itself in some detail, this paper focuses on the managerial lessons to be learned from the simulation model.

Center for Information Systems Research
Sloan School of Management
Massachusetts Institute of Technology
LESSONS LEARNED FROM MODELING THE DYNAMICS OF SOFTWARE DEVELOPMENT

Abstract

The development of software systems has been marked by cost overruns, late deliveries, poor reliability, and users' dissatisfaction. The problems persist in spite of significant advances in the software engineering field to tackle the technological hurdles of software production. In recent years, the managerial aspects of software development have been gaining increasing recognition as being at the cores of both the problem and the solution.

The objective of this paper is twofold. First, we present a paradigm for the study of software project management that is grounded in the feedback systems principles of system dynamics. Feedback processes are universal in social systems in general, and we show how, when applied to software project management, they do provide a powerful lens to view and understand software project behavior.

Second, we summarize the stream of research findings obtained to date and the lessons learned regarding the dynamic implications of an array of managerial policies and procedures pertaining to software development.

1. INTRODUCTION

1.1 The Criticality of Software Management Decisions

The impressive improvements that are continuously being made in the cost-effectiveness of computer hardware are causing an enormous expansion in the number of applications for which computing is becoming a feasible and economical solution. This, in turn, is placing greater and greater demands for the development and operation of computer software systems. A conservative estimate indicates a hundred fold increase in the demand for software in the last two decades (Musa, 1985).

The growth of the software industry has not, however, been painless. The record shows that the development of software has been marked by cost overruns, late deliveries, poor reliability, and users' dissatisfaction [(Buckley and Poston, 1984), (Ramamoorthy et al., 1984), and (Newport, 1986)].

In an effort to bring discipline to the development of software systems, attempts have been made since the early 1970s to apply the rigors of science and engineering to the software production process. This leads to significant advances in the technology of software production (e.g., structured programming, structured design, formal verification, language design for more reliable coding, diagnostic compilers, and so forth).
The managerial aspects of software development, on the other hand, have attracted much less attention from the research community (Zmud, 1980). Cooper (1978) provides an insightful explanation for the reasons why:

Perhaps this is so because computer scientists believe that management per se is not their business, and the management professionals assume that it is the computer scientists' responsibility.

This "deficiency" in the field's research repertoire may account for the persistence of the difficulties in producing software systems. A chief concern expressed is that, as of yet, we still lack a fundamental understanding of the software development process. Without such an understanding the possibility or likelihood of any significant gains on the managerial front is questionable [(Basili, 1982) and (McKeen, 1983)].

This paper reports on a stream of research designed to address the above concerns. Specifically, our goal is to develop a comprehensive model of the dynamics of software development that enhances our understanding of, provides insight into, and makes predictions about the process by which software development is managed. The following examples illustrate some of the critical management decisions that have been addressed in this research effort:

1. A project is behind schedule. Possible management actions include: revise completion date, hold to planned completion date but hire more staff, hold to planned completion date but work current staff overtime, etc. What are the implications of these alternatives?
2. How much of the development effort should be expended on quality assurance and how does that affect completion time and total cost?
3. What is the impact of different effort distributions among project phases (e.g., should the division of effort between development and testing be 80:20 or 60:40 percent)?
4. What are the reasons for and implications of the differences between potential productivity, actual productivity, and perceived productivity?
5. Why does the "90% completion syndrome" chronically recur?

In the remaining parts of this paper we discuss the integrative dynamic model of software project management that has been developed. We will provide an overview of both the model's structure and its behavior followed by a discussion of the insights gained. We begin our presentation, however, by first presenting arguments for the necessity of an integrative and dynamic modeling approach in the study of software project management.
1.2 The High Complexity of the Software Project Management Process

A simple view of the dynamics of project management is illustrated by the single-loop model shown in Figure 1 (Roberts, 1981). The model portrays how project work is accomplished through the utilization of project resources (manpower, facilities, equipment). As work on the project is accomplished, it is reported through some project control system. Such reports cumulate and are processed to create the project's forecast completion time by adding to the current date the indicated time remaining on the job. Assessing the job's remaining time involves figuring out the magnitude of the effort (e.g., in man-days) believed by management to be remaining to complete the project, the level of manpower working on the project, and the perceived productivity of the project team. The feedback loop is completed (closed) as 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. This new level of resources results in a new work rate and the loop is repeated again.

What is attractive about the above model is that it is reasonable, simple, and manageable. It is the mental model that many project managers "rely" on (Roberts, 1980). But is it an adequate model of the dynamics of software project management?

The software project management system is a far more complex conglomerate of interdependant variables that are interrelated in various nonlinear fashions. By excluding vital aspects of the real software project environment, the above model could seriously misguide the unsuspecting software manager. To see how, let us consider just a few of the many typical decisions pondered in a software project environment.

Adding more people to a late project: The mental picture of Figure 1 suggests a direct relationship between adding people resources and increasing the rate of work on the project, i.e., the higher the level of project resources the higher the work rate. This ignores one vital aspect of software project dynamics, namely, that adding more people often leads to higher communication and training overheads on the project, which can in turn dilute the project team's productivity. Lower productivity translates into lower progress rates, which can, therefore, delay the late project even further. This, in turn, can trigger an additional round of workforce additions and another pass around this "vicious cycle." These dynamic forces create the phenomenon often referred to as "Brooks' Law", i.e., that adding more people to a late software project makes it later (Brooks, 1978).
Figure (1)
A MODEL OF SOFTWARE PROJECT MANAGEMENT
In Figure 2a we, therefore, amend Figure 1 by incorporating the vital link between the workforce level and productivity.

**Adjusting the Schedule of a Late Project:** Another part of the real system that is ignored by Figure 1 concerns the impact of schedule pressures on the software developers' actions and decisions. For example, when faced with schedule pressures that arise as a project falls behind schedule, software developers typically respond by putting in longer hours and by concentrating more on the essential tasks of the job (Ibrahim, 1978). In one experiment, Boehm (1981) found that the number of man-hours devoted to project work increased by as much as 100%. This additional link between schedule pressure and productivity is captured in Figure 2b.

The impact of schedule pressures on software development, however, is not limited to the above relatively direct role. Schedule pressures can also play less visible roles. For example, as Figure 2c suggests, schedule pressures can increase the error rate of the project team and thus the amount of rework on the project ([Radice, 1982] and [Mills, 1983]).

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 (DeMarco, 1982).

The rework necessary to correct such software errors obviously diverts the project team's effort from making progress on new project tasks, and thus can have a significant negative impact on the project's progress rate.

**Finally, How Really Late is a Late Software Project?** Because software remains largely intangible during most of the development process, it is often difficult for project managers to assess real progress on the project (Baber, 1982). To the extent that the perceived progress rate differs from the real progress rate, an error in perceived cumulative progress will gradually accumulate (Figure 2d). Furthermore, bias, often in the form of overoptimism, and delay in gathering and processing control information additionally distorts the reported progress. This undoubtedly poses yet another complication that is too real for the software project manager to exclude from a model of the process.

2. **AN INTEGRATIVE SYSTEM DYNAMICS PERSPECTIVE OF SOFTWARE DEVELOPMENT**

The above discussion illustrates that there are a large number of variables, both tangible and intangible, that impact the software development process. Furthermore, these variables are not independent, but are related to one another in complex fashions. Perhaps most importantly, understanding the behavior of such systems is complex far beyond the capacity of human intuition (Roberts, 1981).
Figure (2)
AMENDMENTS TO THE PROJECT MANAGEMENT MODEL
A major deficiency in much of the research to date on software project management has been the inability to integrate our knowledge of the micro components of the software development process such as scheduling, productivity, and staffing to derive implications about the behavior of the total socio-technical system. In the research effort described in this paper we build upon and extend what has been learnt about the micro components, to construct a holistic model of the software development process. It integrates the multiple functions of software development, including both the management-type functions (e.g., planning, controlling, staffing) as well as the software production-type activities (e.g., designing, coding, reviewing, testing).

A second unique feature of our modeling approach is the use of the feedback principles of system dynamics 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. Examples of such feedback systems in the software project environment have already been demonstrated in the above discussion and are evident in Figures 1 and 2.

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, ...

One of the pioneering works in the field is Roberts' (1964) published doctoral dissertation, which involved the development of a comprehensive system dynamics model of R&D project management. The model traces the full lifecycle of a single R&D project and incorporates the interactions between the R&D product, the firm, and the customer. Roberts' work spurred a large number of system dynamics studies of project management phenomena. For example, Nay (1965) and Kelly (1970) extended Roberts' work in their research on multi-project environments. Richardson (1982) took a different tack, focusing on the development group. His model reproduces the dynamics of a development group over an eight year period as a continuous stream of products are developed and placed into production.

While the bulk of the system dynamics modeling work in the project management area has been devoted to the R&D environment, the applicability of the methodology to the domain of software
production has been alluded to in the literature [e.g., (Graham, 1982), (Lehman, 1978), and (Putnam, 1980)]. Perhaps this should come as no suprise, since "the stages of research and development are similar in many respects to the stages of software analysis and design" (Gehring and Pooch, 1977). In the remainder of this section we describe how the system dynamics modeling technique was extended to the software project domain.

2.1 Model Development and Structure

The model was developed on the basis of a field study of software project managers in five organizations. The process involved three information gathering steps:

First, we conducted a series of interviews with software development project managers in three organizations. The purpose of this set of interviews was to provide us with a first hand account of how software projects are currently managed in software developing organizations.

The information collected in this phase, complimented with our own software development experience, were the basis for formulating a "skeleton" system dynamics model of software project management.

The second step was to conduct an extensive review of the literature. The "skeleton" model served as a useful "road-map" in carrying out this literature review. When this exercise was completed, many knowledge gaps were filled, giving rise to a second much more detailed version of the model.

In the third, and final step:

The model is exposed to criticism, revised, exposed again and so on in an iterative process that continues as it proves to be useful. Just as the model is improved as a result of successive exposures to critics a successively better understanding of the problem is achieved by the people who participated in the process (Roberts, 1981).

The setting for this was a second series of intensive interviews with software project managers at three organizations (only one of which was included in the first group).

Figure 3 depicts a highly aggregated view of the model's four subsystems, namely: (1) the Human Resource Management Subsystem; (2) the Software Production Subsystem; (3) the Controlling Subsystem; and (4) the Planning Subsystem. The figure also illustrates some of the interrelationships among the four subsystems. Similarities to Figure 2d can be recognized. Since the actual model is very detailed, containing over a hundred causal links, only a high level description of the model can be presented in the limited space of this paper. For a full discussion of the model's structure and its mathematical formulation the reader is referred to [(Abdel-Hamid, 1984) and (Abdel-Hamid and Madnick, 1988)].
2.2 Human Resource Management Subsystem

The Human Resource Management Subsystem captures the hiring, training, assimilation, and transfer of the human resource. The project's total workforce is segregated into different types of employees, e.g., "Newly Hired Workforce" and "Experienced Workforce." Segregating the workforce into such categories is necessary for two reasons. First, newly added team members are normally less productive (on the average) than the "old timers" (Cougar and Zawacki, 1980). Secondly, it allows us to capture the training processes involved in assimilating the new members into the project team.

On deciding upon the total workforce level needed, project managers consider a number of factors. One important factor, of course, is the project's completion date. As part of the planning subsystem (to be discussed later), management determines the workforce level that it believes is necessary to complete the project on time. In addition, though, consideration is also given to the stability of the workforce. Thus, before adding new project members, management contemplates the duration for which the new members will be needed. In general, the relative weights given to workforce stability versus on-time completion is dynamic, i.e., will change with the stage of project completion. For example, toward the end of the project there could be considerable reluctance to bring in new people. This reluctance arises from the realization that there just wouldn't be enough time to acquaint the new people with the mechanics of the project, integrate them into the project team, and train them in the necessary technical areas.

2.3 The Software Production Subsystem

This Software Production Subsystem models the software development process. The operation and maintenance phases of the software lifecycle are, thus, not included. The development lifecycle phases incorporated include the designing, coding, and testing phases. Notice that the initial requirements definition phase is also excluded. There are two reasons for this. The primary reason relates to the desire to focus this study 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. The requirements definition phase was, thus, excluded since in many environments the definition of user requirements is not totally within the control of the software development group (McGowan and McHenry, 1980). Second, "Analysis to determine requirements is distinguished as an activity apart from software development." (McGowan and McHenry, 1980).
As software is developed, it is also reviewed to detect any errors, e.g., using quality assurance activities such as structured walkthroughs. Errors detected through such activities are reworked.

The formulation of software productivity is based on the work of the psychologist Ivan Steiner (1972). Steiner’s model can simply be stated as follows:

Actual Productivity = Potential Productivity - Losses Due to Faulty Process

Potential productivity is defined as "the maximum level of productivity that can occur when an individual or group ... makes the best possible use of its resources". It is a function of two sets of factors, the nature of the task (e.g., product complexity, database size) and the group’s resources (e.g., personnel capabilities, experience level, software tools). Losses due to faulty process refer to the losses in productivity incurred as a result of the communication and coordination overheads and/or low motivation.

2.4 The Control Subsystem

Decisions made in any organizational setting are based on what information is actually available to the decision maker(s). Often, this available information is inaccurate. Apparent conditions may be far removed from the actual or true state, depending on the information flows that are being used and the amount of time lag and distortion in these information flows. Thus, system dynamicists go through great lengths to differentiate between actual and perceived model variables (Forrester, 1961).

True productivity of a software project team is a good example of a variable that is often difficult to assess. To know what the true value of productivity is at a particular point in time requires accurate knowledge regarding the rates of accomplishment and resources expended over that period of time. However, because software is basically an intangible product during most of the development process, “It is difficult to measure performance in programming . . . It is difficult to evaluate the status of intermediate work such as undetected programs or design specification and their potential value to the complete project” (Mills, 1983).

How, then, is progress measured in a software project’s Control System? Our own field study findings corroborate those reported in the literature, namely, that progress, especially in the earlier phases of software development, is typically measured by the actual expenditure of budgeted resources rather than by some count of accomplishments (DeMarco, 1982). Baber (1982) explains:

It is essentially impossible for the programmers to estimate the fraction of the program completed. What is 45% of a program? Worse yet, what is 45% of three programs? How is he to guess whether a program is 40% or 50% complete? The easiest way for the programmer to
estimate such a figure is to divide the amount of time actually spent on the task to date by the
time budgeted for that task. Only when the program is almost finished or when the allocated
time budget is almost used up will he be able to recognize that the calculated figure is wrong.

When progress in software development is measured solely by the expenditure of budgeted
resources, status reporting ends up being nothing more than an echo of the original plan.

As the project advances towards its final stages, work accomplishments become relatively more
visible and project members become increasingly more able to perceive how productive the
workforce has actually been. As a result, perceived productivity gradually ceases to be a function of
projected productivity and is determined instead on the basis of actual tasks developed.

2.5 The Planning Subsystem

In the Planning Subsystem, initial project estimates (e.g., for completion time, staffing, man-
days) are made at the beginning of the project using a variety of techniques (Kemerer, 1987). These
estimates are then revised, as necessary, throughout the project's life. For example, to handle a
project that is perceived to be behind schedule, plans can be revised to add more people, extend the
schedule, or do a little of both. Such planning decisions are driven by variables that can change
dynamically throughout the project lifecycle. For example, while it is common for management to
respond to a delay in the early stages of the project by increasing staff level, there is often great
reluctance to do that later in the lifecycle. This reluctance arises from the realization that there just
wouldn't be enough time to acquaint the new people with the mechanics of the project, integrate
them into the project team, and train them in the necessary technical areas.

3. Model Validation

3.1 Validation Tests Performed

The process of judging the validity of a system dynamics model includes a number of objective
tests (Richardson and Pugh, 1981), all of which were performed to validate this model:

- **Face validity.** To test the fit between the rate/level/feedback structure of the model and the
  essential characteristics of the real system. This 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 system under study. Reference modes
  include the "90% syndrome," (Abdel-Hamid, 1988c) and Brook's Law (Abdel-Hamid, 1988a).
- **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., hiring delay equals to zero) is suspect, because one may not be certain when aspects of extreme conditions may occur in ordinary runs.

- **Case study.** After the model was completely developed, a Case Study was conducted at NASA’s Goddard Space Flight Center (GSFC) to validate the model. (NASA was not one of the five organizations studied during model development.) The Case Study involved a simulation of one of GSFC’s software projects, namely, the DE-A project.

### 3.2 DE-A Case Study

The objective of the DE-A project was to design, implement, and test a software system for processing telemetry data and providing attitude determination and control for the DE-A satellite. The project's size was 24,000 delivered source instruction (24 KDSI), the development and target operations machines were the IBM 360-95 and -75, and the programming language was FORTRAN. Initially, the project was estimated to require 1,100 man-days and to be completed in 320 working days. The actual results were 2,200 man-days and 380 days, respectively.

The model’s DE-A simulation run is depicted in Figure 4. As shown, the model’s results (represented by lines) conformed quite accurately to the project’s actual behavior (represented by the circular points in the figure). Notice how 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 instead made to the project’s workforce 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 simply 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 workforce pattern, on the other hand, is quite atypical. In the literature, workforce buildup tends to follow a concave curve that rises, peaks, and then drops back to lower levels as the project proceeds towards 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 three months before launch. As this date was approached, pressures developed that overrode normal workforce stability considerations. That is, project management became increasingly willing to “pay any price”
Figure (4)

Model Simulation of the DE-A Project

- DE-A's actual "Estimated Schedule in Days"
- DE-A's actual "Estimated Project Cost in Man-Days"
- DE-A's actual "Workforce" (full-time-equivalent people)
necessary to avoid overshooting the three months before-launch date. This translated, as the figure indicates, into a management that was increasingly willing to add more people. [In (Abdel-Hamid, 1988a) we investigate whether that staffing policy did or did not contribute to the project's late completion.]

On the other hand, various typical behavior patterns can be seen, such as the "90% completion syndrome" [(DeMarco, 1982) and (Abdel-Hamid, 1988c)]:

... estimates of the fraction of the work completed (increase) as originally planned until a level of about 80-90% is reached. The programmer's individual estimates then increase only very slowly until the task is actually completed. (Baber, 1982)

Its manifestation in the DE-A project is depicted in Figure 5. By measuring progress in the earlier phases of the project by the rate of expenditure of resources, status reporting ended up being nothing more than an "illusion" that the project was right on target. However, as the project approached its final stages (e.g., when 80-90% of the resources are consumed), discrepancies between the percent of tasks accomplished and the percent of resources expended became increasingly more apparent. At the same time, project members became increasingly able to perceive how productive the workforce has actually been. This resulted in a better appreciation of the amount of effort actually remaining. As this appreciation developed, it started to, in effect, discount the project's progress rate. Thus, although the project members proceeded towards the final stages of the project at a high work rate because of schedule pressures, their net progress rate slowed down considerably. This continued until the project completed.

4. EXPERIMENTS UNDERTAKEN AND IMPLICATIONS OF RESULTS

"In software engineering it is remarkably easy to propose hypotheses and remarkably difficult to test them" (Weiss, 1979). Many in the field have, thus, argue for the desirability of having a laboratory tool for testing ideas and hypotheses in software engineering (Thayer, 1979).

The computer simulation tools of system dynamics provide us with such an experimentation vehicle. 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. Internally, the model provides complete control of the system's organizational structure, its policies, and its sensitivities to various events.

Currently, the model is being used to study and predict the dynamic implications of managerial policies and procedures on the software development process in a variety of areas. This has produced three kinds of results: (1) Uncovered dysfunctional consequences of some currently
Figure (5)
REPORTED % OF WORK COMPLETED
adopted policies (e.g., in the scheduling area); (2) Provided support for managerial decision-making (e.g., on the allocation of the quality assurance effort); and (3) provided insight into software project phenomena (e.g., "90% syndrome" and "Brooks' Law").

4.1 Uncovering dysfunctional consequences of some currently adopted policies.

We investigated the project scheduling practices in a major U.S. minicomputer manufacturer in (Abdel-Hamid and Madnick, 1986). In the particular organization, software project managers use Boehm's (1981) COCOMO model to determine initial project estimates, which are then adjusted upwards using a judgemental "safety factor" to come up with the project estimates actually used. In this organization, project managers were rewarded on how closely their project estimates matched actual project results. The purpose of the experiment was to investigate the implications of this safety factor policy.

To test the efficacy of various safety factor policies, we ran a number of simulations on a prototypical software project that we will call project Example. Project Example's actual size is 64,000 DSI. At its initiation, however, it was incorrectly estimated to be 42.88 KDSI in size (that is, 33\% percent smaller than 64 KDSI). This incorrectly perceived project size of 42.88 KDSI was the input used in COCOMO's estimation equations. We experimented with safety factor values ranging from 0 (the base run) to 100 percent. For a 50-percent safety factor, for example, the actual estimate used on the project would be (1 + 50/100) * COCOMO's estimates.

In Figure 6, the "percent relative error" in estimating man-days, defined as 100 x |Actual - Estimatetl/Actual, is plotted against different values of the safety factor. Notice that the safety factor policy seems to be working -- the larger the safety factor, the smaller the estimation error.

The rationale for using a safety factor is based on the following assumptions:

1. Past experience indicates a strong bias among software developers to underestimate the scope of software projects (Boehm 1981).

2. One might think biases are the easiest of estimating problems to correct since they involve errors moving always in the same direction. But as DeMarco (1982) suggests, biases are almost by definition invisible; the same psychological mechanism that creates the bias (for example, the optimism of software developers) works to conceal it.

3. To rectify such bias, project managers often use a safety factor. Pietrasanta (1968) observes that when project managers add contingency factors (ranging, say, from 25 to 100 percent), they are
Figure (6)

The percentage of relative error in estimating actual man-days
saying in essence: I don't know all that is going to happen, so I'll estimate what I don't know as a percentage of what I do know.

In other words, the assumption is that safety factors are simply mechanisms to bring initial man-day estimates closer to true project size in man-days (see Figure 7a). Such an assumption cannot be contested solely on the basis of Figure 6 which provides only part of the story. Figure 7b presents a more complete picture; here, we used the model to calculate the actual man-days consumed by the project Example when different safety factors were applied to its initial estimate. The Figure 7a assumption is obviously invalidated. As we use higher safety factors, leading to increasingly generous initial man-days allocations, the actual amount of man-days consumed does not remain at some inherently defined value. In the base run, for example, project Example would be initiated with a man-day estimate of 2,359 man-days and would consume 3,795 man-days. When a 50-percent safety factor is used, leading to a 3,538-man-day initial estimate, Example ends up consuming not 3,795 man-days but 5,080 man-days.

The above results clearly indicate that by imposing different estimates on a software project we would, in a real sense, be creating different projects. This can be explained by realizing that schedules have a direct influence on decision-making behavior throughout a software project's life. In TRW's COCOMO model (Boehm, 1981), for example, the project's average staff size would be determined by dividing the man-day estimate by the development time estimate (TDEV). Thus, a tight time schedule means a larger work force. Also, scheduling can dramatically change manpower loading throughout the life of a project. For example, the work force level in some environments shoots upwards towards the end of a late project when there are strict constraints on the extent to which the project's schedule is allowed to slip. Through its effects on the work force level, a project's schedule also affects productivity (as illustrated in Figure 3). For example, a higher work force level means more communication and training overhead, affecting productivity negatively.

Productivity is also influenced by how tight or slack a project schedule is. If a project falls behind under a tight schedule, software developers often decide to work harder in an attempt to compensate for the perceived shortage and bring the project back on schedule. Conversely, man-day "excesses" could arise if project management initially overestimates a project; as a result, the project would be perceived ahead of schedule. When such a situation occurs, "Parkinson's law indicates that people will use the extra time for... personal activities, catching up on the mail, etc." (Boehm, 1981). Of course, this means that they become less productive.
(a) A COMPARISON OF ASSUMED MAN-DAYS WITH ESTIMATED MAN-DAYS

(b) A COMPARISON OF ACTUAL MAN-DAYS WITH ESTIMATED MAN-DAYS

Figure (7)
DIFFERENCES BETWEEN ASSUMED MAN-DAYS AND ACTUAL MAN-DAYS
One important managerial lesson learned from the above experiment is this: More accurate estimates are not necessarily "better" estimates. An estimation method should not be judged only on how accurate it is; it should also be judged on how costly the projects it creates are! For example, in one situation studied, we found that the estimation error which would have been 38% had been reduced to 9% by the safety factor policy. But, that policy resulted in a 43% cost increase in the project. For the first time management had a realization of the "cost" of their more accurate schedule estimation policy.

4.2 Provide support for management decision making.

The quality assurance (QA) function has, in recent years, gained the recognition of being a critical factor in the successful development of software systems. However, because the utilization of QA tools and techniques can add significantly to the cost of developing software, the cost-effectiveness of QA has been a pressing concern to the software quality manager. As of yet, though, this concern has not been adequately addressed in the literature.

We have investigated the tradeoffs between the economic benefits and costs of QA in [(Abdel-Hamid, 1988b) and (Abdel-Hamid and Madnick, 1987b)]. To do this, we utilized the model as a laboratory vehicle to conduct controlled experiments on QA policy. Effects considered in this experiment included error generation rate factors, such as schedule pressures and phase of project, and error detection factors, such as productivity, error types, error density. The results showed that the percent of the effort allocated to QA dramatically affects the total project cost. In one case studied (Figure 8a), the resulting total project cost ranged from 1,648 man-days to 5,650 man-days with the optimal QA allocation being 15% of the development effort.

Two important conclusions can be drawn from Figure 8a. The first, more generalizable conclusion, is that QA policy does have a significant impact on total project cost. 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 expenditures, the excessive QA expenditures are themselves the culprit. The relationship between QA effort and the percentage of errors detected is shown in Figure 8b. 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 [e.g., (Boehm, 1981) and (Shooman, 1983)].

The second important conclusion we can draw, concerns the 15% value for the optimal QA expenditure level. What, in our opinion, is really significant about this result is not its particular
(a) IMPACT OF DIFFERENT QA EXPENDITURE LEVELS ON PROJECT COST

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

Figure (8)
IMPACT OF DIFFERENT QA EXPENDITURE LEVELS
value, since this cannot be generalized beyond the DE-A software project used for this study, but rather the process of deriving it, namely, this paper’s integrative system dynamics simulation approach. Beyond controlled experimentation (which would be too costly and time consuming to be practical), as far as we know, this model provides the first capability to quantitatively analyze the costs/benefits of QA policy for software production. This, it is encouraging to note, is generalizable in the sense that one can customize the model for different software development environments to derive environment-specific optimality conditions.

4.3 Providing new insights into software project phenomena.

One oft-cited software project phenomenon is “Brooks’ Law,” which states that adding manpower to a late software project makes it later (Brooks, 1978). Since its publication, Brooks’ Law has been widely referenced in the literature even though it has not been formally tested. Furthermore, it has often been applied indiscriminately e.g., for applications-type projects as well as systems programming-type projects, both large and small (Pressman, 1982), even though Brooks was quite explicit in specifying the domain of applicability of his insights, to what he called “jumbo” systems programming projects.

We have studied “Brooks’ Law” in the context of medium-size applications-type projects in (Abdel-Hamid, 1988a). Recall the workforce staffing pattern experienced on NASA’s medium-sized DE-A project of Figure 4. It indicates that management is (implicitly if not explicitly) oblivious to the lesson of Brooks’ Law. 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 (or overreacted) by adding new people to the project to meet the strict launch deadline . . . as evidenced by the rising workforce curve in the final stages of the project.

The lesson of Brooks’ Law would, of course, suggest that by adding new people to the late DE-A project, management actually delayed it further. To test this hypothesis, we re-simulated the DE-A project under different staffing policies. The experimental results showed that while adding more people to a late project of this type does cause it to become more costly, it does not always cause it to complete later. The increase in the cost of the project is caused by the increased training and communication overheads, which in effect decrease the average productivity of the workforce and thus increase the project’s cost in man-days. For the project’s schedule to also suffer, the drop in productivity must be severe enough and late enough in the project’s lifecycle to render an additional
person's net cumulative contribution to the project to be, in effect, a negative contribution. Our experimental results indicate that this happens only where management's willingness to add new staff members persists until the very final stages of the testing phase. In the particular case of the DE-A project where hiring did continue until the very end (as seen in Figure 4), our analysis indicates that the project period could have been cut by two calendar weeks by curtailing hiring during the testing phase.

CONCLUSION

The objective of this research effort is to enhance our understanding of the software development process and how it is managed. There are two principal features that characterize our research paradigm, which we would like to reiterate in these concluding remarks. First, we emphasize the integrative perspective. We have attempted to demonstrate how the software management system is a conglomerate of interrelated and interdependent functions. Action taken by one subsystem (e.g., human resource management) can be traced throughout the entire management system (e.g., software production, planning, and control). The behavior of an individual subsystem in isolation may be very different from its behavior when it interacts with other subsystems (e.g., the lesson of Brooks' Law). Furthermore, differences in the environment and management policies between companies could explain why a software engineering technique that is effective in one organization may be ineffective in another organization (Abdel-Hamid and Madnick 1987a).

Second, our research approach is grounded in the feedback systems principles of system dynamics. Feedback processes are universal in social systems in general. We have attempted to show how, when applied to software project management, they do provide a powerful lens to view and understand software project behavior. This is particularly important because:

"most (software) managers get into trouble because they forget to think in circles. I mean this literally. Managerial problems persist because managers continue to believe that there are such things as unilateral causation, independent and dependent variables, origins, and terminations" (Weick, 1979).

What is gained in understanding though the use of such a model to portray a portion of the real world is achieved by comprehending the law or laws built into the model. There are hundreds of variables that affect software development. Furthermore, these variables are not independent; many of them are related to one another. So far the many studies on the subject emphasize the difficulty and complexity of the process, but have done little to reveal a well-defined methodology
or to delineate precise relationships among project variables (Oliver, 1982). We feel that the research paradigm presented in this paper provides a viable vehicle.

If understanding is the intellectual outcome of a theoretical model, then prediction is its practical outcome. This model is also being used as an experimentation vehicle to study and predict the dynamic behavior of the software development process and of the implications of managerial policies and procedures pertaining to the development of software. The exercise produced three kinds of results: (1) uncovered dysfunctional consequences of some currently adopted policies (e.g., in the scheduling area); (2) provided guidelines for managerial policy (e.g., on the allocation of the quality assurance effort); and (3) provided new insights into software project phenomena (e.g., Brooks' Law).

Finally, a note on possible future research directions. Two such research areas are particularly promising. The first involves extending this modeling approach to investigate, not the dynamics of a single software project, but rather to study the software development organization, as a continuous stream of software products are developed, placed into operation, and maintained. A number of research questions are ripe for investigation, including: (1) the efficacy of different organizational structures (e.g., project, functional, and matrix); (2) personnel turnover, its costs (e.g., recruiting and training overheads), its benefits (e.g., access to new ideas and methodologies), and its causes (e.g., schedule pressures, maintenance load); and (3) the organizational/environmental determinants of productivity (e.g., standards, software tools, use of librarians, documentation requirements). Again, one needs to investigate both short-term as well as long-term implications. For example, an investment in developing powerful software development tools (e.g., compilers, automated testing tools) might hamper productivity in the short-run, but may lead to better software in the long-run.

Second, we are investigating the utility of incorporating AI-based modules into the model (Abdel-Hamid and Sivasankaran, 1988). According to (Nielsen, 1986), an expert-system form of representation for decision making can be particularly advantageous in situations where the actual system being modeled is not fully automated but, instead, relies upon human judgment or direct human intervention for a portion of its operation or behavior. "A knowledge-based programming approach may be used to capture the human element of the decision making, so that such knowledge may be appropriately reflected in the model" (Nielsen, 1986).

The management of software projects is such an application area. Capturing the decision making process (e.g., in the staffing area) in a rule-based knowledge base, rather than using the
traditional representation in procedural code has a number of benefits. For example, it allows for
the incorporation of an explanation capability to the model.

The experiments that have been performed already, described in the previous section, illustrate
the insights that can be gained from applying this paradigm to the myriad of concerns facing
software development managers. Further work in these directions will help to resolve many more of
these concerns.

Acknowledgements

We appreciate the contribution of each of the individuals in the organizations providing
perspectives and data to this research effort. In addition, we thank Robert Zmud and Chris Kemerer,
whose suggestions have improved this article's organization and readability. Work reported herein
was supported, in part, by NASA research grant NAGW-448.

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

2. Abdel-Hamid, T.K. "The Dynamics of Software Project Staffing: A System Dynamics Based

3. Abdel-Hamid, T.K. "The Economics of Software Quality Assurance: A Simulation-Based Case

4. Abdel-Hamid, T.K. "Understanding the '90% Syndrome' in Software Project Management: A

5. Abdel-Hamid, T.K. and Madnick, S.E. "Impact of Schedule Estimation on Software Project


   Dynamics Based Simulation Approach," The Annals of the Society of Logistics Engineers, Vol. 1,
   No. 2, (October, 1987b), 8-32.


