THE DYNAMICS OF SOFTWARE PROJECT SCHEDULING

TAREK K. ABDEL-HAMID and STUART E. MADNICK *Massachusetts Institute of Technology*

Tarek K. Abdel-Hamid's present research interests include the management of software development projects, application of system dynamics to software engineering. Stuart E. Madnick's research includes system design methodologies, management information systems and database computers. He is an associate editor of TODS, a trustee of the VLDB Foundation, a former member of the board of governors of the IEEE Computer Society, and a founding member and past chairman of the IEEE Technical Committee on Database Engineering.

ABSTRACT: Software project scheduling is one of the major problem areas faced by software project managers today. While several quantitative software project resource and schedule estimation methods have been developed, such techniques raise some important, but as yet unresolved, dynamic issues. A systems dynamics (SD) approach is used to analyze several key dynamic software project scheduling issues.

1. THE DYNAMICS OF SOFTWARE PROJECT SCHEDULING: AN INTRODUCTION

The software industry is young, growing, and marked by rapid change in technology and application. It is not surprising, then, that the ability to estimate project resources, including the time resource, is still relatively undeveloped. In a recent study investigating the major problem areas faced by software project managers today, "... the ability to estimate accurately the resources required to accomplish a software development" and "... the ability to estimate accurately the delivery time on a software development" were two of the thirteen "definite" problem areas identified [8].

> The problem of resource estimating of computer program system development is fundamentally qualitative rather than quantitative. We don't understand what has to be estimated well enough to make accurate estimates. Quantitative analyses of resources will augment our qualitative understanding of program system development, but such analyses will never substitute for this understanding [1].

This is why when methods of estimating are ranked, the list is headed by what Aaron called the experience method—estimates that are largely based on human judgements gained through previous experiences with software projects [1].

In the last few years there has been a surge in activity to develop "quantitative" resource estimation methods, for example, TRW's COCOMO model [3] and Putnam's SLIM [5]. A closer look, however, reveals that "at heart" such methods are still of the experience type, since they all get calibrated using historical data on completed software projects.

For example:

> The initial COCOMO model ... was calibrated using 12 completed projects. The resulting model was then evaluated with respect to a fairly uniform sample of 36 projects, primarily aerospace applications, producing fairly good agreement between estimates and actuals ...
>
> The calibration and evaluation of COCOMO has not relied heavily on advanced statistical techniques. After trying to apply advanced statistical techniques to software cost estimation, and after observing similar efforts by others, I have become convinced that the software field is currently too primitive, and software cost driver interactions too complex, for standard statistical techniques to make much headway; and that more initial progress

Authors' Present Address: Tarek K. Abdel-Hamid and Stuart E. Madnick, Center for Information Systems Research, MIT, Sloan School of Management, 50 Memorial Drive, Cambridge, MA 02139.

Madnick @ MIT-Multics

0001-0782/83/0500-0340$00.75.

could be made by trying to formulate empirically the nature of the interactions between cost drivers, using functional forms which reflected the best available perspectives and data on software life-cycle phenomenology.

Because the currently available quantitative techniques are usually tailored to a limited set of project/organizational types and are still imperfect, the developers of such techniques emphasize the necessity to continuously collect project data via the planning and control activities, compare estimates to actuals, and use the results to improve the estimating tools.

Furthermore, research findings over the past few years have clearly shown that the decisions that people make in organizations and the actions they choose to take are ***significantly*** influenced by the pressures, perceptions, and incentives produced by the organization's planning and control system(s) [9]. In particular, knowledge of ***project schedules*** was found to affect the real progress rate that is achieved, as well as the progress and problems that are reported upward in the organization.

What this implies is the existence of a feedback loop (see Figure 1): an estimation technique produces project schedules, which affect the decisions and actions of the technical performers and their managers, which in turn

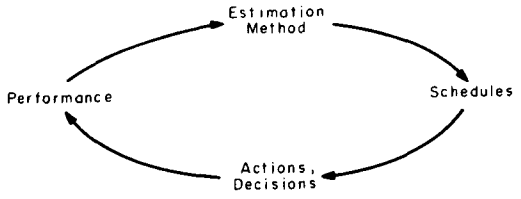

FIGURE 1. Scheduling Feedback Loop.

affect work performance, which eventually is fed into the organization's projects' database to influence future estimations.

But what does the existence of such a feedback loop mean? Is it good or bad? To most of us, the answers to such questions will ***not*** be intuitively obvious, that is, we cannot answer them with confidence merely on the basis of our private mental models. The human mind is not adapted to correctly anticipate the dynamic consequences of interactions between the parts of a complex societal system [4], such as that of software project management. The most familiar example in the literature is perhaps "Brooks' Law.' When a project falls behind schedule, managers often attempt to speed up the project by adding more people. Brooks suggests that managers fail to anticipate the dynamic consequences of such an action, for example, the increase in the communication overhead and the need to divert productive time of the current personnel to the training of the new people. The net effect is that the project may actually fall further behind schedule.

A Systems Dynamics Model is important because "Unlike a mental model, a system dynamics computer model can reliably trace through time the implications of any messy maze of assumptions and interactions" [6]. It can do so without stumbling over phraseology, emotional bias, or gaps in intuition. Even for those gifted few who can correctly answer the above questions on the basis of mere intuition, a formal system dynamics model can provide a powerful communication tool that can minimize misunderstanding and miscommunication.

In the remainder of this paper we will elaborate the above argument further. We will attempt to demonstrate how the modeling, simulation, and analysis techniques of system dynamics (SD) can be powerful tools in studying the complex area of software project management in general, and the scheduling issues discussed above in particular.

2. COMPUTER MODELING OF SOCIAL SYSTEMS: THE SYSTEM DYNAMICS PERSPECTIVE

> People would never attempt to send a spaceship to the moon without first testing the equipment by constructing prototype models and by computer simulation of the anticipated space trajectories. No company would put a new kind of household appliance or electronic computer into production without first making laboratory tests. Such models and laboratory tests do not guarantee against failure, but they do identify many weaknesses which can then be corrected before they cause full-scale disasters.
>
> Our social systems are far more complex and harder to understand than our technological systems. Why, then, do we not use the same approach of making models of social systems and conducting laboratory experiments on those models before we try new laws and government programs in real life? The answer is often stated that our knowledge of social systems is insufficient for constructing useful models. But what justification can there be for the apparent assumption that we do not know enough to construct models but believe we do know enough to directly design new social systems by passing laws and starting new social programs? I am suggesting that we now do know enough to make useful models of social systems. Conversely, we do not know enough to design the most effective social systems directly without first going through a model-building experimental phase. But I am confident, and substantial supporting evidence is beginning to accumulate, that the proper use of models of social systems can lead to far better systems, laws, and programs [4].

Indeed, it is now possible to use the modeling, simulation, and system analysis techniques of system dynamics in the laboratory to construct models of managerial systems. Such models are obviously simplifications of the actual social systems but can be far more comprehensive than the mental models that are otherwise used.

System Dynamics (SD) is the application of feedback control systems principles and techniques to managerial and organizational problems. The SD philosophy rests on a belief that the behavior (or time history) of an organizational entity is principally caused by its structure. The structure includes, not only the physical aspects, but more importantly, the policies and traditions, both tangible and intangible, that dominate decision making in the organizational entity. Such a structural framework contains sources of amplification, time lags, and information feedback similar to those found in complex engineering systems.

The system dynamics approach begins with an effort to understand the system of forces that has created a problem and continues to sustain it. Relevant data are gathered usually from a variety of sources, for example, literature, informed people, etc. As soon as a rudimentary measure of understanding has been achieved, a formal model is developed. This model is initially in the format of a set of logical diagrams showing cause-and-effect relationships. As soon as is feasible, the visual model is translated into a mathematical version. The model is exposed to criticism, revised, expressed again,

and so on, in an iterative process that continues as long 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 participate in the process.

Such an approach forces those involved in system design to make explicit and thoroughly test the assumptions that underlie their design decisions: the nature of problems, their causes, the consequences of alternative actions, and how various human, managerial, economic, and operational factors interrelate. Weil reports that his experience has shown that this is a very valuable process; people are really quite surprised when it turns up things no one had thought of before, incorrect assumptions, and differences of opinion about cause and effect [9].

Roberts has stated that people's

> . . . intuition about the probable consequences of proposed policies frequently proves to be less reliable than the model's meticulous mathematical approach . . . This is not surprising as it may first appear. Management systems contain as many as 100 or more variables that are known to be related to one another in various nonlinear fashions. The behavior of such a system is complex far beyond the capacity of intuition. Computer simulation is one of the most effective means available for supplementing and correcting human intuition [7].

System dynamics is suitable for addressing certain kinds of complex problems. In addition to complexity, such problems have at least two features in common. First, they are dynamic, that is, they involve quantities that change over time and that can, therefore, be expressed in terms of graphs of variables over time. Oscillating levels of employment in an industry, a decline in a city's tax base and quality of life, and the dramatically rising pattern of health care costs are all dynamic problems [6]. Cost overruns, slippages in scheduled completion dates, and the variability in productivity are some of the dynamic problems that can arise in a software project.

A second feature of the problems to which the system dynamics perspective applies involves the notion of feedback. "Most succinctly, feedback is the transmission and return of information. The emphasis, inherent in the word feedback itself is on the return" [6]. A feedback system exists whenever an action taker will later be influenced by the consequences of his or her actions. The consequences may be quick and directly apparent in results produced, for example, when the hiring of ten more programmers increases the programmers' workforce to a certain desired level, which in turn feeds back to affect the hiring rate, that is, stopping further hiring. Or the consequences may be delayed though directly apparent in results produced, for example, when a software development manager's decision to use a particular package to estimate his/her human resource requirements affects the project's completion time and cost, which in turn influences the manager's later estimation procedure. Finally, the consequences may be both delayed and quite indirect in perceived results, for example, when a decision to increase the software development budget leads to the hiring of higher quality managers and analysts/programmers, who may then develop improved products, which may enhance the company's competitive position, in turn increasing sales and/or profits, which may then influence the decision on the software development budget. In all these cases a "closing of the loop" occurs. A delay, whether short or long, intervenes between initial action and feedback results. Closed loops and time delays in consequences are characteristic of all feedback processes.

It is apparent from the above that the management of software development is a prime candidate for application of the system dynamics method. It clearly is complex, it is dynamic, and it does exhibit feedback behavior.

In [2] we discuss, in some detail, a system dynamics model we developed of software project management. The model was developed to be a tool in investigating the many managerial problems that seem to be plaguing software development activities in most organizations, for example, cost and schedule overruns. In particular, the model serves as a "skeleton" on top of which "custom" models can be built to fit different organizational/project settings.

In Sec. 4 we use the model to analyze the dynamics of software project scheduling, an exercise we hope will produce some insights into the issues raised in Sec. 1. To set the stage for this discussion, we will characterize, next in Sec. 3, the software project (which we will simply call EXAMPLE) to be used in our analysis.

3. CHARACTERIZING THE "EXAMPLE" SOFTWARE PROJECT

One of the attractive features of a computer model is its high versatility. By simply changing a few model parameters, for example, one can easily simulate a wide range of software project types. The software project EXAMPLE involves the development of a 400,000 DSI system. (DSI stands for "Delivered Source Instructions." See [3] for a complete definition.) All of the model's graphical output, will be in terms of the unit, ***task***, where a task is

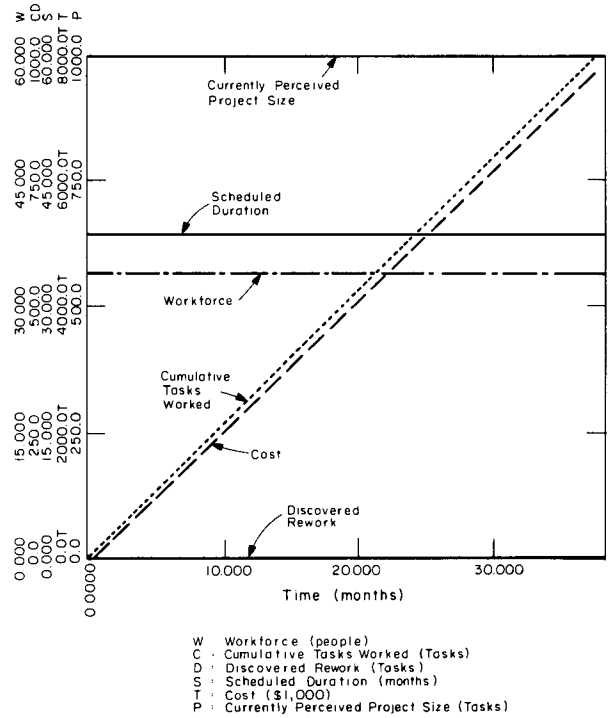

FIGURE 2. The "Flawless" Project.

equivalent to 400 DSI. (Our project is, thus, of size 1000 tasks.) The development period modeled begins at the beginning of the product design phase and ends at the end of the integration and test phase. We deliberately excluded the requirements definition/specification phase so we could incorporate the use of TRW's **CO**nstructive **CO**st **MO**del (**COCOMO**) [3]. That is, we assume that management will use COCOMO to estimate the effort in man-months (MM), the total development time in months (TDEV), and the staff size (SS).

In this section we characterize our EXAMPLE software project. We will do it in a stepwise fashion. We will start with an "ideal" project situation and then, step-by-step, add increments of "reality."

Step (1): The "Flawless" Project

In the ideal case, management's estimate of the project's size will be exactly on target, that is, 400,000 DSI or 1000 tasks. Using the (Basic form of the) COCOMO model, an estimate of the effort in man-months (MM) can then be made.

$$\begin{aligned} MM &= 2.4\ (DSI/1000)^{1.05} \\ &= 2.4\ (400{,}000/1000)^{1.05} \\ &= 1295 \text{ man-months} \end{aligned}$$

From this an estimate of the project's total development time (TDEV) can be calculated.

$$\begin{aligned} TDEV &= 2.5\ (MM)^{0.38} \\ &= 2.5\ (1295)^{0.38} \\ &= 38 \text{ months} \end{aligned}$$

Finally, the average staff size (SS) is determined.

$$\begin{aligned} SS &= MM/TDEV \\ &= 34 \text{ people} \end{aligned}$$

The dynamic behavior of the "flawless" project situation is shown in Figure 2. The project's scheduled completion duration is set to 38 months and a workforce of 34 people is assembled (e.g., during the requirements/specification phase). As can be seen, the project proceeds "smoothly" and is completed on schedule at a total cost of $7,810,600. (It is assumed that a rather uniform effort is required throughout the project. This simplifying assumption will be relaxed in a later version of the model in which the software lifecycle will be explicitly divided into three phases: design, coding, and testing.)

Unfortunately the project behavior of Figure 2 is rarely (if ever) realized in real organizations. It will, therefore, only serve us as a reference point for further analysis.

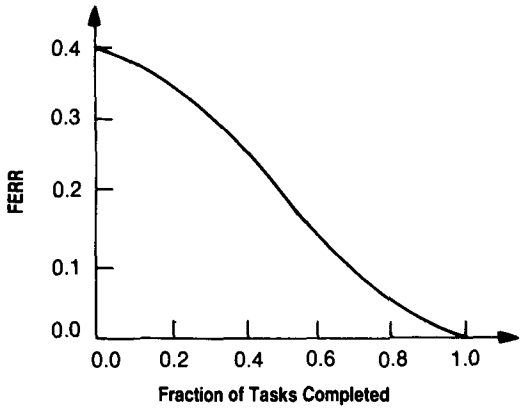

FIGURE 3. Assumed Shape of FERR.

Step (2): Introducing Rework

Not all work done in the course of a large software project is errorless. Some fraction of the work will be less than satisfactory (e.g., inconsistent design, defective code, etc.) and must be redone. The unsatisfactory work may not be discovered right away, however. For some time it passes unrecognized, until the need for reworking the tasks shows up. The discovery of unsatisfactory work that needs reworking can, of course, cause major disruptions in a software project (e.g., people are diverted from new tasks to redoing old ones) and is, therefore, a significant element of the software development environment.

In the model, the generation of rework is regulated by FERR, which is the fraction of work that is erroneous. For example, a value of FERR equal to 0.1 means that 10 percent of the tasks completed in a particular unit of time will be defective, and will thus require reworking. FERR is not modeled as a constant, however. It is a variable that changes during the life of the project. Figure 3 depicts the assumed relationship between FERR and the fraction of tasks completed.

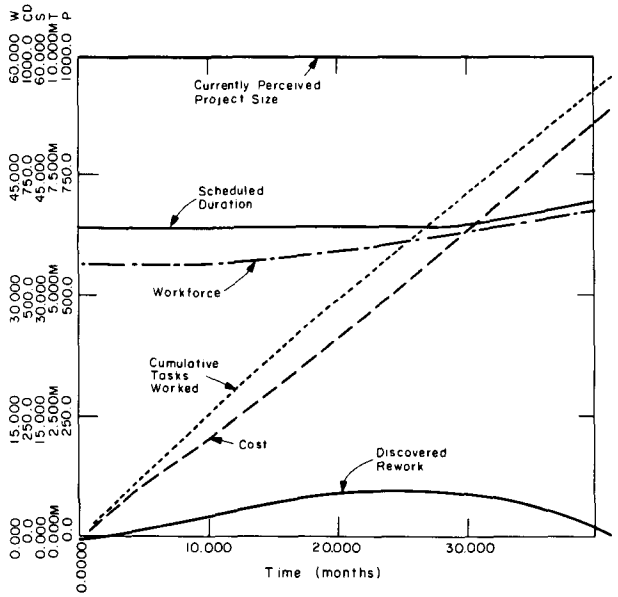

FIGURE 4. Introducing Rework.

The rationale for making FERR a variable in this fashion is the realization that tasks near the beginning of a large software project (e.g., design) are much different from those near the end (e.g., documentation). Near the beginning, the project is being defined, different approaches are being explored, ideas are on the drawing board, etc. Near the end, the tasks are more like finishing touches: assembling documentation, typing reports, and so on. The likelihood of performing work that must be redone is, thus, modeled to be greater at the beginning of the project than at the end.

The behavior of the model is shown in Figure 4. At the bottom of the figure, the level of rework perceived (i.e., discovered) throughout the life of the project is shown. The generation of rework means, of course, that more effort will be required to complete the project satisfactor-

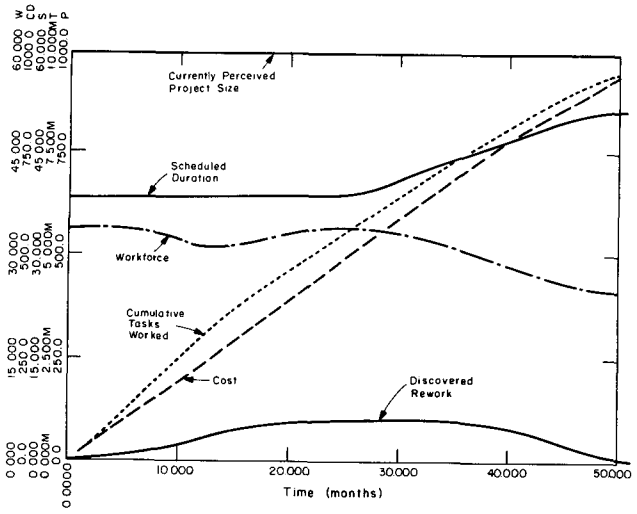

FIGURE 5. Introducing Personnel Turnover.

ily. After month 13 enough rework is discovered to cause management to start hiring more people in order to meet the initial scheduled completion date. This policy continues until month 29, when management chooses instead to extend the project's target completion date and slow down the hiring of new people. The project eventually completes after 41.5 months, at a total cost of $9,024,700.

Step (3): Introducing Personnel Turnover

In this run we divert even further from the flawless project situation. ***In addition*** to the generation of rework, we introduce the effect of personnel turnover. In the flawless project run it was assumed that people do not quit (after all, who would quit an ideal project?). For this run, we assume that the average employment time is 24 months. As shown in Figure 5, the project is adversely affected, finishing even later at month 51 and at a higher cost of $9,582,400.

Step (4): Introducing Estimating Error

In the flawless project we assumed that management's estimate of the project's size was exactly on target, that is, 1000 tasks. Obviously this is too optimistic an assumption.

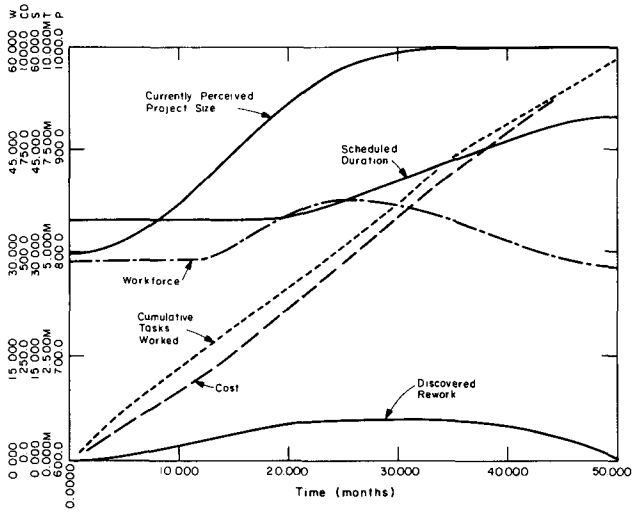

FIGURE 6. Introducing Estimating Error.

For this run, we assume that management initially estimates the system's size to be 320,000 DSI or 800 tasks, that is, 20 percent lower than the real size of 400,000 DSI or 1000 tasks. The resulting project behavior is shown in Figure 6. The project completes in 51 months but at the slightly higher cost of $9,757,200.

Notice that the "currently perceived project size" starts at 800 tasks, but as the project progresses and the level of uncertainty decreases, it is adjusted upwards until at about month 35 "currently perceived project size" is in fact equal to 1000 tasks—the true project size. As management learns about the upward adjustments in the project's size, it adjusts its workforce upwards to the level it perceives is sufficient to complete the project within the scheduled 35 months. However, unexpected problems arise, for example, a system integration test fails miserably, which will necessitate the reworking of tasks believed to be successfully completed. When such disruptions occur towards the end of the project, management is reluctant to hire new employees, and there is almost no other alternative but to extend the scheduled completion date, as shown in Figure 6.

In the next section we will use EXAMPLE to analyze the dynamics of software project scheduling. In all our subsequent simulation runs we will maintain our project's present level of "realism," that is, include rework, personnel turnover, and underestimation.

4. THE DYNAMICS OF SOFTWARE PROJECT SCHEDULING: A SIMULATION EXPERIMENT

In Sec. 1 we suggested that project schedules create pressures, perceptions, and incentives that affect the decisions and actions of the technical performers and their managers, that this in turn affects work performance, which is all eventually fed into the organization's projects' database to influence future scheduling activities. We also suggested that the dynamic consequences of such a feedback loop are ***not*** intuitively obvious. The modeling, simulation, and analysis techniques of system dynamics were then proposed as a powerful tool to reliably deduce and analyze the dynamic behavior of complex feedback systems, such as that of managing software projects. In this section we use our system dynamics model of software project management to conduct a laboratory experiment to investigate the software project scheduling issues raised in Sec. 1.

The experiment involves a hypothetical situation in which a company undertakes a sequence of ten ***identical*** software projects, all identical to project EXAMPLE of Sec. 3. On the first such project, EXAMPLE1, the company (lacking the experience) underestimates the size of the project by 20 percent, that is, estimates the project's size to be only 800 tasks. Using the (TRW calibrated) COCOMO model, the project's total effort and duration are then estimated to be 1025 man–months and 35 months, respectively. But as we have already seen in Sec. 3 (Figure 6), the project actually consumes 1626 man–months and is completed in 51 months.

After completing project EXAMPLE1, the following is learned:

- Project EXAMPLE1 is really 1000 (and not 800) tasks.
- It consumes 1626 man–months.
- It takes 51 months to complete.

The COCOMO model's "guardian" in the company then notes that had an accurate estimate of project size (i.e., 1000 tasks) been made, COCOMO's estimates for effort and project duration would have been 1295 man-months and 38 months, respectively. Knowing that COCOMO is an imperfect estimation tool that needs to be continuously improved, adjustments are made such that for any future projects identical to EXAMPLE1 estimates of 1626 man-months and 51 months would be produced.

Some time later when project EXAMPLE2 (which is identical to EXAMPLE1) comes along, management will be in a position to better estimate its true size. In fact, we assume that EXAMPLE2's size will be estimated perfectly, that is, to be 1000 tasks. Furthermore, the now "improved" COCOMO model will produce estimates of 1626 man-months and 51 months for EXAMPLE2's total effort and duration, respectively. Based on these figures, management determines that a staff size of 32 will be required.

Conducting project EXAMPLE2 under such circumstances produces the behavior of Figure 7. The project ***still*** finishes late, 56 months after it started, and is 5 months behind the "improved" schedule.

When we repeated the above sequence of actions and reactions eight more times for projects EXAMPLE3 through EXAMPLE10, we were surprised to observe that the schedule was overrun in each case. As a result management started each project (e.g., EXAMPLEi) with a slightly longer scheduled duration than the previous one (i.e., EXAMPLEi-1). However, EXAMPLEi would still overrun its schedule, which caused management to use

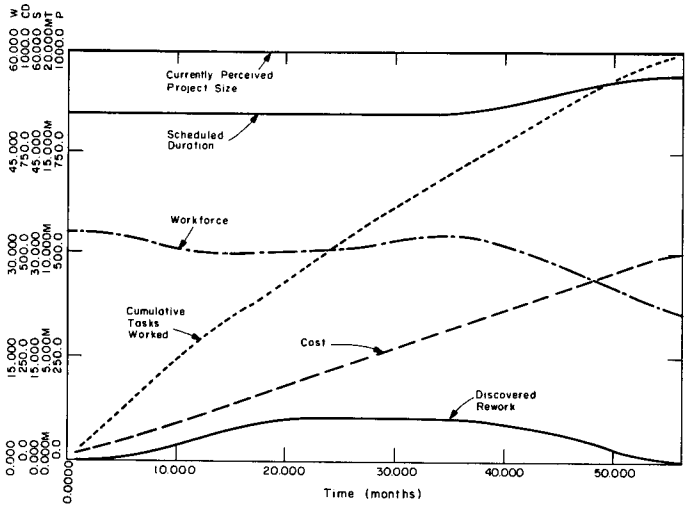

FIGURE 7. EXAMPLE 2 [Notice change in the COST scale].

an even longer scheduled duration for the next project. The results for the ten simulation runs are shown in Figure 8.

It seems our feedback loop turned out to be quite a villain, producing devastating effects, especially over the long run. Notice that EXAMPLE1 was completed in 51 months, while nine projects later, EXAMPLE10 completed in 81.25 months! A very significant deterioration indeed.

The surprising phenomenon we observed is one that has been frequently encountered in system dynamics studies of real organizations. It has been termed "The

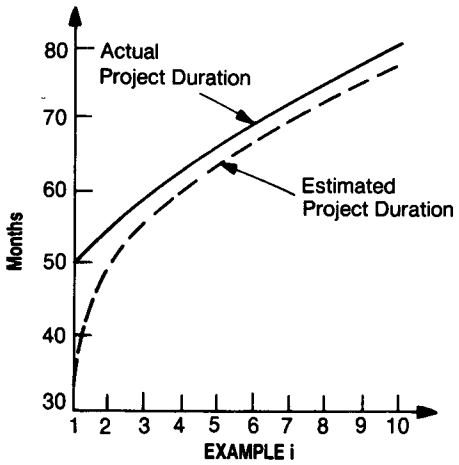

FIGURE 8. Results of the Ten Simulation Runs.

Policy Resistance of Social Systems," "Shifting the Burden to the Intervenor," and "Addiction," among other things. A simple example of such a phenomenon is that of caffeine addiction, whereby an addict has to consume a certain amount of caffeine per day to maintain a certain level of alertness. As time goes on the burden of maintaining alertness will keep shifting from the normal physiological body processes to the externally supplied caffeine dose. The result, of course, is that higher and higher doses will be required to maintain the *same* level of alertness.

Forrester put it this way:

> Social systems are inherently insensitive to most policy changes that people select in an effort to alter the behavior of the system. In fact, a social system tends to draw our attention to the very points at which an attempt to intervene will fail. Our experience, which has been developed from contact with simple systems, leads us to look close to the symptoms of trouble for a cause. When we look, we discover that the social system presents us with an apparent cause that is plausible according to what we have learned from simple systems. But this apparent cause is usually a coincident occurrence that, like the trouble symptom itself, is being produced by the feedback-loop dynamics of a larger system [4].

In our experiment we saw how looking at the symptoms of trouble (i.e., schedule overruns) for a cause and opting for the most apparent one, namely, that schedules were too tight, might lead to relaxing the schedules. However, this turns out to be quite ineffective in curing the system's problematic behavior.

An attractive feature of system dynamics models is the ability to conduct simulation experiments in which we can isolate the effects of factors we suspect are causing the problematic behavior. By exploring the behavior generated by individual feedback loops and by various combinations of loops in a model, the modeler can precisely pinpoint the structure responsible for the behavior.

By conducting such experimentation with our model, we were able to identify the real cause of the persisting schedule overrun problem in our modeled software development project. It turned out to be a consequence of the interaction between management's hiring/firing policy and the "elusive" nature of software errors.

When deciding upon the number of employees, management (in the model) takes into account the perceived time remaining for the project. Toward the end of the

project, for example, management would be very reluctant to bring in new people, even though the perceived time and effort remaining imply more people are needed. It would just take too much time to acquaint new people with the mechanics of the project, integrate them into the project team, and train them in the necessary technical areas.

While such a policy may sound perfectly reasonable, and may in fact be successfully employed in many areas of managerial endeavour, it does pose certain risks (our model reveals) when applied to the software development area. As was mentioned in Sec. 3, not all work done in the course of a large software project is errorless. Some fraction of the work will be less than satisfactory and must be redone. The unsatisfactory work is not discovered right away, however. For some time it passes unrecognized, until the need for reworking the tasks involved shows up. When the unsatisfactory work does get discovered, it usually causes major disruptions. The problems, however, are particularly devastating when this happens towards the end of the project, for example, at system integration testing, when management (under the above hiring/firing policy) is very reluctant to hire new employees. When this happens, management will have almost no other alternative but to extend the project's scheduled completion date.

There is still, though, an interesting and as yet unanswered question: Why was the system so "unresponsive" to the "schedule relaxing policy?" The answer is because of the ***compensating property*** of complex societal systems. Changes in most (but certainly not all) parameters in ***one*** part of such systems may weaken or strengthen some feedback loop, but the multiloop nature of complex feedback systems will, in most cases, "naturally" strengthen or weaken other loops to compensate [6]. In our particular software project situation, extending the project's schedule weakens the strength of the schedule pressure in the system, to which the hiring loop (for one) simply compensates by causing the project to start with a smaller workforce. For example, in EXAMPLE2, on the basis of an estimated project duration of 51 months, management starts the project with 32 people. When supplied with the more generous and presumably more accurate estimate of 56 months for EXAMPLE3, management simply (and rationally) factors that in the decision making process, and then determines that a workforce of 29 people (1626 MM/56 months) is needed.

5. CONCLUSIONS

It is clear from the above discussion that viewing the problem of software project scheduling as simply a problem of generating improved schedules is too limited a view, and one which can in fact lead to a serious long term deterioration in an organization's effectiveness in managing its software projects. Our own research efforts, as well as those of others, have convinced us that the project management of software development is a very complex undertaking in which a complex network of interrelationships and interactions exists. It is, therefore, essential that software project managers adopt an ***integrative*** perspective or model of software project dynamics in order to effectively answer the difficult questions they need to raise when assessing their organizations' health, selecting improvement interventions, and implementing their choices. Such an integrative model would be useful to alert managers to all the important elements or aspects of a problematic situation, and to help them assess the second- and third-order consequences of some set of actions.

However, such an integrative model will undoubtedly contain a large number of components with a complex network of interrelationships. What would still be needed is an effective means to determine both accurately and efficiently the dynamic behavior implied by such component interactions. We feel (and hopefully have demonstrated) that the computer-based simulation techniques of system dynamics can be a powerful tool to do just that.

REFERENCES

1. Aaron, J. D. Estimating resources for large programming systems. *Software Engineering: Concepts and Techniques*. Edited by J. M. Buxton, P. Naur, and B. Randell. Litton Educational Publishing, Inc., 1976.
2. Abdel-Hamid, T. K. and Madnick, S. E. A model of software project management dynamics. *The Sixth Int'l Computer Software and Applications Conference (COMPSAC)*, November 8–12, 1982.
3. Boehm, B. W. *Software Engineering Economics*, Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1981.
4. Forrester, J. W. Counterintuitive behavior of social systems. *Technology Review*, January, 1971.
5. Putnam, L. H. Software cost estimation and life-cycle control: Getting the software numbers. IEEE Computer Society, IEEE Catalog No. EHO 165-1, 1980.
6. Richardson, G. P. and Pugh III, A. L. *Introduction to System Dynamics Modeling with Dynamo*. The MIT Press, Cambridge, Massachusetts, 1981.
7. Roberts, E. B., Ed. *Managerial Applications of System Dynamics*. The MIT Press, Cambridge, Massachusetts, 1981.
8. Thayer, R. H., Pyster, A. B., and Wood, R. C. Major issues in software engineering project management. *IEEE Trans. on Software Engineering*, July, 1981, pp. 333–342.
9. Weil, H. B. Industrial dynamics and management information systems. *Managerial Applications of System Dynamics*. Edited by E. B. Roberts. The MIT Press, Cambridge, Massachusetts, 1981.

CR Categories and Subject Descriptors: D.2.9 [**Software Engineering**]: Management—*cost elimination*; K.6.1 [**Management of Computing and Information Systems**]: Project and People Management—*management techniques*

General Terms: Management

Additional Key Words and Phrases: software project scheduling, computer simulation, system dynamics

Received 6/82; revised 9/82; accepted 10/82