THE DYNAMICS OF SOFTWARE DEVELOPMENT PROJECT MANAGEMENT: AN INTEGRATIVE SYSTEM DYNAMICS PERSPECTIVE

by

TAREK K. ABDEL-HAMID

B.Sc., CAIRO UNIVERSITY, CAIRO
(1972)

MBA, STATE UNIVERISITY OF NEW YORK, ALBANY
(1978)

Submitted to the Department of Management
in Partial Fulfillment of the
Requirements for the Degree of

DOCTOR OF PHILOSOPHY

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

January 1984

Signature of Author: _____
Department of Management, 6 January 1984

Certified by: _____
Stuart E. Madnick, Thesis Supervisor

Accepted by: _____
Chairman, Department Committee on
Graduate Studies

THE DYNAMICS OF SOFTWARE DEVELOPMENT PROJECT MANAGEMENT: AN INTEGRATIVE SYSTEM DYNAMICS PERSPECTIVE

by

Tarek K. Abdel-Hamid

Submitted to the Department of Management
in Partial Fulfillment of the
Requirements for the Degree of
Doctor of Philosophy

Software is big business. It has been estimated that expenditures for software development and maintenance were 40 billion dollars in 1980, or approximately 2 % of GNP. Even more impressive are the projections that software will be the dominant portion of an information processing industry that is expected to grow to 8.5 % of GNP by 1985 and to 13 % of GNP by 1990.

The growth in the software industry has not, however, been painless. The record indicates that the development of software systems has been plagued by cost overruns, late deliveries, and users' dissatisfaction. A set of difficulties that some refer to as the "software crisis." The problems persist inspite of the significant software engineering advances that have been made over the last decade in tackling many of the technical hurdles of software production. In recent years, the managerial aspect of software development has gained recognition as being at the cores of both the problem and the solution. Along with this recognition there are, however, serious and legitimate reservations and concerns. Chief among them is the belief that, as of yet, we still lack the fundamental understanding of the software development process, and that without such an understanding the likelihood of any significant gains on the managerial front is questionable.

The objective of this research effort is to enhance our understanding of, and gain insight into, the general process by which software development is managed. To achieve this objective we accomplished the following three tasks:

First, we developed an integrative system dynamics model of software development project management. The model was developed on the basis of an extensive review of the literature supplemented by 27 focused field interviews of software project managers in 5 organizations. The model complements and builds upon current research efforts, which

tend to focus on the micro components (e.g., scheduling, programming, productivity, ... etc.), by integrating our knowledge of these micro components into an integrated continuous view of the software development process.

Second, a case-study in a sixth organization was conducted to test the model. The model was highly accurate in replicating the actual development history of the software project selected (by the organization) for the case-study. Project variables tracked included: the workforce level, the schedule, the cost, error generation and detection, and productivity.

Third, the model was used as an experimentation vehicle to study/predict the dynamic implications of an array of managerial policies and procedures. Four areas were studied: (1) scheduling; (2) control; (3) Quality Assurance; and (4) staffing. 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 quality assurance effort); and (3) provided new insights into software project phenomena (e.g., Brooks' Law).

Thesis Supervisor : Dr. Stuart E. Madnick
Associate Professor of Management Science
Sloan School of Management
Massachusetts Institute of Technology

ACKNOWLEDGEMENTS

Many a doctoral candidate, sitting with pencil stub in hand, must have thought, as I did, of the first few phrases of Don Quixote: "Idle reader, you may believe me without any oath that I would want this (work), the child of my brain, to be the most beautiful, the happiest, the most brilliant imaginable. But I could not contravene that law of nature according to which like begets like." If such can be said of Cervantes' brainchild, what can one possibly say about one's own?

Only that one has done one's best. And yet, if the truth be told, that "best" may prove to belong as much to certain others as to one's self. Among the many people who have made invaluable contributions to this thesis, I am particulary indebted to Professor Stuart E. Madnick for his support, encouragement, and constructive suggestions from the time this study was conceived through the writing of the final draft. His interest and willingness to help in every way possible have kept me on course. I have also been fortunate in having the counsel and assistance of Professors Ugo Gagliardi, John Morecroft, and Edward Roberts; their efforts in my behalf are sincerely appreciated.

Although I cannot hope to mention them all by name in this short space, I would like to acknowledge the debt owed to each and every individual in the organizations with whom I worked in collecting the data for this thesis. The open and supportive way in which I was accepted in all of these organizations was most gratifying, and, indeed, instrumental to my research work.

I would also like to gratefully acknowledge my main sources of financial support. These include several Sloan School tuition fellowships, an IBM Information Systems Fellowship, and research funding from NASA (Grant No. NAGW-448).

Nadia, my wife, is the one person of whom it can be said that without her this thesis would not have been written. Her companionship, support, and intellectual stimulation have been at the very core of it since its inception.

TO NADIA

MY WIFE AND BEST FRIEND

TABLE OF CONTENTS

I. INTRODUCTION: BACKGROUND, OBJECTIVE, AND APPROACH 8

I.1 BACKGROUND 8
I.2 Research Objective and Approach 18
I.2.1 Why an Integrative Model 20
I.2.2 Why a System Dynamics Model 24
I.3 Research Accomplishments 31
I.4 Thesis Outline 35

II. REVIEW OF RELEVANT LITERATURE 38

II.1 System Dynamics Modeling of Project Management 38
II.2 Software Engineering Project Management Literature Review 48
II.2.1 Review Models and Frameworks 48
II.2.2 Planning 72
II.2.3 Management of the Human Resource 78
II.2.4 Control 87

III. MODEL DEVELOPMENT 94

III.1 Introduction 94
III.2 Sources of Information 96
III.3 Model Boundary 109
III.4 Model Structure 113
III.4.1 Model Overview 114
III.4.2 System Dynamics Schematic Conventions 117
III.4.3 Human Resource Management 121
III.4.4 Software Production 133
III.4.5 Controlling 231
III.4.6 Planning 256
III.5 Summary 268

IV. A CASE-STUDY: THE NASA DE-A SOFTWARE PROJECT 270

IV.1 The DE-A Project 271
IV.2 Model Parameterization 272
IV.3 Actual and Simulated Project Behavior 289
IV.4 Conclusion 296

V. MODEL BEHAVIOR: AN ANALYSIS OF THE DYNAMICS OF SOFTWARE DEVELOPMENT 301

V.1 Introduction 301
V.2 The "EXAMPLE" Software Project 303
V.3 Software Cost and Schedule Estimation 338
V.3.1 On the Accuracy of Software Estimation 342
V.3.2 On the Portability of the Quantitative Software Estimation Models 358
V.3.3 On the Analogy Method of Software Estimation 387
V.4 The "90% Syndrome" 400
V.5 The Economics of Quality Assurance 410
V.6 Staffing: Brooks' Law Revisited 428
V.7 Summary 438

VI. CONCLUSIONS AND SUGGESTIONS FOR FUTURE RESEARCH 441

VI.1 Summary of Results 441
VI.2 Suggestions for Future Research 453

BIBLIOGRAPHY 459

APPENDIX: MODEL DOCUMENTATION 497

I. INTRODUCTION:
BACKGROUND, OBJECTIVE, AND APPROACH

I.1. Background:

> In the brief history of the electronic digital computer, the 1950s and 1960s were decades of hardware. The 1970s were a period of transition and a time of recognition of software. The decade of software is now upon us (Pressman, 1982).

One convincing impact of software is directly on the pocketbook. It has been estimated that, here in the U.S., expenditures for software development and maintenance were 40 billion dollars in 1980, or about 2 percent of the Gross National Product (Boehm, 1981). Even more impressive, Boehm projects that "computer software will be the dominant portion of an (overall computer and information processing) industry expected to grow to 8.5% of the Gross National Product by 1985 and to 13% of the GNP by 1990."

This growth in demand for software has not, however,

been painless. Indeed, as the industry was making the transition in the 1970s, " ... we (grew) to recognize circumstances that are collectively called the 'software crisis,' ... (a term that) alludes to a set of problems that are encountered in the development of software" (Pressman, 1982).

The record shows that the software industry has been marked by cost overruns, late deliveries, poor reliability, and users' dissatisfaction. [For example, see (Block, B58), (Boehm, 1981), (Frank, 1983), (Glaseman, 1882), (Jensen & Tonies, 1979), (Mills, 1976), (McKeen, 1983), (Thayer & Lehman, 1980), and (Thayer el al, 1981).]

A report to Congress by the Comptroller General, General Accounting Office (GAO), FG MSD-80-4, November 9, 1979, cites 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 reflects the views of 163 software contracting firms and 113 federal government project officers, as well as experience with specific contracts for software development. The summarized indictment is severe:

1. Dollar overruns are fairly common in more than 50 percent of cases

2. Calender overruns occur in more than 60 percent of cases

3. Of the nine contracts examined (eight of which were admittedly in trouble), of $6.8 million expended, the results were:

   a. Software delivered but never used: $3.2 million

   b. Software paid for, but never delivered: $1.9 million

   c. Software extensively reworked before used: $1.3 million

   d. Software used after changes: $198,000

   e. Software used as delivered: $119,000.

As the report concludes, "The government got for its money less than 2 percent of the total value of the contracts."

Big as the direct costs of the "software crisis" are, the indirect costs can be even bigger, because software, in many cases, is on the critical path in overall system development (e.g., weapon systems such as the B-1 bomber). That is, any slippages in the software schedule translate directly into slippages in the overall delivery schedule of the system. For example:

> Let's see what this meant in a recent software development for a large defense system. It was planned to have an operational lifetime of seven years and a total cost of about $1.4 billion --- or about $200 million a year worth of capability. However, a six-month delay caused a six-month delay in making the system available to the user, who thus lost about $100 million worth of needed capability --- about 50 times the direct cost of $2 million for the additional software effort. Moreover, in order to keep the software from causing further delays, several important functions were not provided in the initial delivery to the user (Boehm, 1973).

The "software crisis" is, by no means, confined to software projects developed by or for the federal government. There is every indication that it is similarly prevalent within private sector organizations [(Brooks, 1978), (Mclure, 1955), (McFarlan, 1974), and (Zmud, 1980)]. For example, in his most recent book, DeMarco (1982) writes about:

> ... some disquieting facts to be considered:
>
> * 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.
>
> (And that) 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.

In an effort to bring discipline to the development of

software systems, attempts have been made since the early 1970s to apply the more rigorous discipline of engineering to software production. This new discipline is called "Software Engineering." And it encompases both the technical aspects of software development (e.g., design, testing, validation, ... etc.) as well as the managerial ones (Thayer, 1979), (Boehm, 1980)).

However, even though both technology and management were equally recognized very early on as parts of both the problem and the solution [(Kolence, 1968), (Perlis, 1969), and (Mills, 1974)], there was a huge disparity in the attention they received from the research Community.

On the technology side, a number of methodologies have evolved, over the last decade, that address many of the technical problems experienced in software development. A large number of articles addressing such topics as better coding style "Structured programming", structured design, testing, formal verification, language design for more reliable coding, diagnostic compilers, and so forth, have appeared in the literature (e.g., in the IEEE Transactions on Software Engineering, Proceedings of the International Conferences on Software Engineering, Proceedings of the ACM Conferences on the Principles of Programming Languages, ...). (See, for example, (Dijkstra, 1971), (Fagen, 1976), (Jensen and Tonies, 1979), (Mills, 1971), (Parnas, 1972), and

(Stevens et al, 1974).)

> ... software engineers have progressed to the point where many major issues relevant to the technology of software production have been identified and considerable progress in addressing these issues has been made. Practical working tools to support improved software production are commonly available, and their design and generation have become a recognized topic for university instruction (Thayer et al, 1981).

A comparable evolution in Management methodologies, however, has not occured [(Cooper, 1978), (DoD, 1982), (Gehring and Pooch, 1980), (Jensen and Tonies, 1979), (Hausen and Mullerburg, 1982a), (McClure, 1981), (McFarlan, 1974), (McKeen, 1981), (Reifer, 1979), (Thayer, 1979), (Weinberg, 1982), (Zmud, 1980), and (Beck and Perkins, 1983)].

In a special isssue of the IEEE Transactions on Software Engineering devoted to project management, Dr. Richard E. Merwin (1978), the Guest Editor, pointed out that an overall software engineering management discipline is missing. He stated:

> Programming discipline such as top-down design, use of standardized high level programming languages, and program library support systems all contribute to production of reliable software on time, within budget ... What is still missing is the overall management fabric which allows the senior project manager to understand and lead major data processing development efforts.

And, within the same issue, Cooper, (1978) commented

that:

> Although the need is apparent, there appears to be precious little innovative activity in the area of software management. 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.

Three years later, Thayer et al, (1981) writing in the same Journal, stated that:

> Software engineering project management (SEPM) has not enjoyed the same progress (as the technology of software development). While it might be argued that SEPM has been defined, it is far from a recognized discipline. Software developers who have demonstrated competence as developers and programmers have been elevated to project managers without training or guidelines to help them. The major issues and problems of SEPM have not been agreed on by the computing community as a whole, and consequently, priorities for addressing them have not been widely established. Furthermore, research in this area has been scant.

This position is further substantiated by a survey, reported in the same paper, of a number of leading universities, which revealed that only a handful of the prominent universities surveyed offered courses exclusively on SEPM.

But what have been the consequences of this "deficiency" in our "research repertoire?"

First, our difficulties in producing software that is on

time, within budget, and that meets user requirements, are obviously very much still "alive." (Refer to the many references cited in the early part of this discussion.)

Second, and because this continues to be the case inspite of substantial progress in the technological (vis-a-vis the managerial) aspects of software production, there is a decided shift in "faith." Consider:

> There are more opportunities for improving software productivity and quality in the area of management than anywhere else. (Boehm, 1976)
>
> Many of our technical and managerial leaders believe that the more effective management of a software development project (i.e., project management) would eliminate or reduce the severity of these software failures (Thayer, 1979).
>
> The basic problem is management itself (Gehring and Pooch, 1977).
>
> A major barrier to the successful design and implementation of information systems has been the management of the software developement activity itself (Moore, 1979).
>
> Poor management can increase software costs more rapidly than any other factor (Weinberg, 1982).
>
> A comprehensive study for the U.S. Air Force found that the problems of software productivity on medium- to large-scale projects are mostly problems of management: thorough organization, good contingency planning, thoughtful establishment of measurable project milestones, continuous monitoring as to whether the milestones are properly passed, and prompt investigation and corrective action in case the milestones are not met. However, beyond these familiar concessions to classic management theory, the study group offered no novel approaches to finding out why they do not work for software development. (Pooch and Gehring, 1980)
>
> We ran into problems because we didn't know how to manage what we had, not because we lacked the techniques

themselves (Thomsett, 1980).

Along with the growing "faith" in software engineering project management, there are, however, serious and legitimate reservations and concerns. Chief among them is the belief that, as of yet, we still lack the fundamental understanding of the software development process [(Comper, 1979), (DOD, 1982), (Fireworker, 1980), (Gehring, 1976), (Merwin, 1978), (McKeen, 1983), (McKeen, 1981), (Oliver, 1982), and (Wesserman, 1980)], and that without such an understanding the possibility or likelihood of any significant gains on the management front is questionable [(Basili, 1982), (Basili and Zelkowitz, 1978), (Brooks, 1978), (Basili, 1981), (Canning, 1978), (McKeen, 1981), and (Mitchell, 1980)].

This is no trivial impediment ((McKeen, 1981), (Oliver, 1982)). But, if it is any solace, it is not one that is unique to our young field:

> Any worthwhile human endeavor emerges first as an art ...
>
> Over the centuries, management as an art has progressed by the acquisition and recording of human experience. But as long as there is no orderly underlying scientific base, the experiences remain as special cases. The lessons are poorly transferrable either in time or in space ... (And) in time (the art) ceases to grow because of the disorganized state of its knowledge ...
>
> The development of the underlying science (is then) motivated by the need to understand better the foundation on which the art rested ...

When the need and necessary foundation coincide, a science develops to explain, organize, and distill experience into a more compact and usable form ... Such a base of applied science would permit experience to be translated into a common frame of reference from which they could be transfered from the past to the present or from one location to another, (and) to be effectively applied in new situations ... (Forrester, 1961).

To summarize:

* The record shows that the software industry continues to be plagued by cost overruns, late deliveries, poor reliability, and users' dissatisfaction. A set of difficulties that some refer to as the "Software Crisis."

* In an effort to bring discipline to the development of software, attempts have been made since the early 1970's to apply the more rigorous discipline of engineering to software production and management. The new discipline is called "Software Engineering."

* While significant inroads have been made in tackling the technical hurdles of software development, the managerial aspects of software production attracted much less attention.

* There is a growing "faith" that the next significant "battle" will be won on the "managerial front."

* A necessary first step, however, is gaining a fundamental understanding of the general nature of the software development process.

I.2. Research Objective and Approach:

The objective of this research effort is to develop and test an integrative system dynamics model of software development project management which would enhance our understanding of, provide insight into, and make predictions about, the general process by which software development is managed.

The first, and primary, purpose of the model is to enhance our understanding of the software development process. In general;

> What is gained in understanding through the use of a scientific model to portray a portion of the real world is achieved by comprehending the law or laws built into the model. The locus of understanding in a scientific model is to be found in its laws of interaction (i.e., the modes of interaction among the the variables of a model) (Dubin, 1971).

There are hundreds of variables that affect software development. Furthermore, these variables are not independent; many of them are related to one another (Myres, 1976). 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).

Even though we do not de-emphasize the "difficulty and complexity of the software development process," we feel that the powerful formalization and simulation tools of the System Dynamics methodology, have allowed us (as we shall explain in more detail later in this section) to adequately manage it.

The second purpose of our model, is to make predictions about the general process by which software systems are developed. As such, the model would serve as a framework for experimentation, e.g., to test out the implications of new managerial policies and procedures. Providing such a capability, is "especially useful for analyzing consequences of changes in the (modeled) system where controlled manipulation of the system itself is impossible, or at least impractical or undesirable due to time, cost, inaccessibility, political or moral considerations, or other reasons" (Schultz and Sulliven, 1972).

In the remaining part of this section we will elaborate further on the above ideas. We will do that as we argue for the two characteristic features of our model and which together distinguish it from most others in the software engineering area. The two characteristic features being:

(1) It is integrative, and (2) it is a system dynamics model.

I.2.1. Why an Integrative Model:

Our model is integrative in the sense that it integrates the multiple functions of the software development process, including the management-type functions, e.g., planning, controlling, and staffing, as well as the production-type functions that constitute the software development life cycle, e.g., designing, coding, reviewing, and testing.

A major defect in much of the research to date has been its inability to integrate our knowledge of the micro components, such as project management, programming, testing, ... etc., for deriving implications about the behavior of the organization in which the micro components are embedded ((Boehm, 1976), (Thayer, 1979)). Paraphrasing Jensen and Tonies (1979):

> There is much attention on individual phases and functions of the software development sequence, but little on the whole life cycle as an integral, continuous process --- a process that can and should be optimized.

Clearly, this "micro-oriented" type of work is a useful beginning in helping us obtain a better understanding of the software development activity. However, before we can say that we have a complete understanding of any such activity,

" ... it is necessary to show that our knowledge of the individual components can be put togehter in a total system, i.e., an organization can be synthesized, which allows for the interactions of all the relevant variables and of all the structural components" (Cohen, 1965).

The basic argument for this, is that interactions and interdependencies are common in all social systems, e.g., management-type systems (Kotter, 1978), (Schein, 1980), (Weick, 1979). Paraphrasing Cleland and King (1972):

> The management system is a conglomerate of interrelated and interdependent functions. No one management subsystem can perform effectively without the others. Action taken by one subsystem can be traced throughout the entire management system and throughout the complex environment in which the management system exists.

And, that as a result:

> The behavior of an individual subsystem in isolation may be very different from its behavior when it interacts with other subsystems (Cohen, 1965).

It is no wonder, then, that integrative-type models are viewed as useful and powerful aids in understanding management-type social systems generally, and in trying to improve their functioning (Schein, 1980). And the management of software development is, certainly, no exception:

> ... the solution to the (software management) problem involves more than just finding better tools and local optimization methods; it calls for an integrated

approach ... (Jensen and Tonies, 1979).

In addition to the benefit of helping us achieve overall understanding, an integrative perspective can be useful in two more "tactical" ways: problem diagnosis and solution evaluation.

A "corollary" of the above statements by Cleland and King (1972), is that the interactions and interdependencies which tend to characterize our management systems generally, will similarly characterize the problems that beset such systems (Cleland and King, 1975). Which does indeed seem to be the case in software development (Glassman, 1982), where " ... no one thing seems to cause the difficulty ... But the accumulation of simultaneous and interacting factors ... " (Brooks, 1978).

An integrative perspective would, therefore, be useful since, at worst, it would not "inhibit" our search for the multiple, and potentially "diffused," set of factors that are interacting to cause our software problem(s), while, at best, actually "prompting" and "facilitating" such a search. Such prompting should be useful since experience suggests that more often than not people opt for a "parochial mode" of problem solving (Ackoff, 1978), (Cleland and King, 1975). By doing so, the problem solve, in effect, brings to the problematic situation under study a set of ready-made

criteria of relevance. Quite a "risky" strategy when we admittedly lack a fundamental understanding of the problem area.

To see the second potential benefit of our integrative perspective, we need a second "corollary," namely: the chain of effects in going from a particular managerial intervention (e.g., to solve a perceived problem) to immediate consequences, and then to second- and third-order consequences and newly created problems is another pervasive characteristic of management-type social systems ((Cleland and King, 1975), (Weick, 1979)).

By providing us with a comprehensive world view, the model would help us to more fully assess such second- and third-order consequencies of, for example, a set of management policies and procedures we need to test. And it would do that, again, by, at worst, not "inhibiting" our search for such multiple, and potentially diffused, set of consequences, while, at best, actually "prompting" and "facilitating" such a search. Such prompting should be useful, since often,

> ... consequences are not given much attention, and apparently logical solutions may prove faulty as their consequences ramify. Furthermore, since the consequences of a decision often occur much later than the decision itself, it is difficult for the members to trace backward from the disruptive consequences to determine precisely what caused them. The members cannot make such an analysis, simply because there are

too many competing explanations. Thus, the only thing members can do when a new problem arises is to engage in more localized problem-solving (Weick,1979).

Notice that Weick's statements highlight two "new" complicating factors, namely, that the consequences are dynamic and that they are complex. And that's quite timely, since these are issues we address next.

I.2.2. Why a System Dynamics Model:

"System Dynamics is the application of feedback control systems principles and techniques to managerial, organizational, and socioeconomic problems" (Roberts, 1981).

The System Dynamics philosophy is based on several premises ((Forrester, 1961), and (Roberts, 1981)):

1. 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 procedures, both tangible and intangible, that dominate decision-making in the organizational entity.

2. Managerial decision-making takes place in a framework that belongs to the general class known as information-feedback systems.

3. Our intuitive judgement is unreliable about how these systems will change with time, even when we have good knowledge of the individual parts of the system.

4. 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 over-all system results.

Based on these philosophical beliefs, two principal foundations for operationalizing the system Dynamics technique were established. These are:

1. The use of information-feedback systems to model and understand system structure (Premises 1 and 2).

2. The use of computer simulation to understand system behavior (Premises 3 and 4).

In the remaining part of this section we would like to discuss these two important concepts in more detail, e.g., find out what they mean and why they are useful?

(a) The use of information feedback systems:

"Feedback," is the process in which an action taken by a

person or thing will eventually affect that person or thing. A feedback loop is a closed sequence of causes and effects, a closed path of action and information. Feedback loops divide naturally into two categories which are labelled deviation-amplifying feedback (DAF) or positive loops, and deviation-counteracting feedback (DCF) or negative loops. An interconnected set of feedback loops is a feedback system (Richardson and Pugh, 1981).

> The first year of exploration (in System Dynamics) pointed toward the concepts of feedback systems as being much more general, more significant, and more applicable to social systems than had been commonly realized ... Feedback processes emerged as universal in social systems and seemed to hold the key to structuring and clarifying relationships that had remained baffling and contradictory (Forrester, 1968).

The significance and applicability of the feedback systems concept to managerial systems has, since then, been further substantiated by a large number of studies in the System Dynamics field. (See for example Roberts, 1981). But what, perhaps, is more interesting is to see "endorsements" of the concept from outside the System Dynamics community. For example:

> 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, ... (Weick, 1979).

Embracement of the feedback concept can even by "spotted" in the software engineering literature. For example:

> Discussion and research into the framework of software development and support, by dividing such efforts into phases of work, has overemphasized the discrete nature of that work. Indeed such project life cycles can be viewed, at least after the fact, as having been composed of such segments. However, the dynamics essence, the behavior over time, of the process is distorted. The emphasis is upon discrete sets of activities separated in time and lacking any base of underlying common elements to bind them. From this it is clear, that the fundamental systems nature of the process is ignored. The ever-present and controlling feedback between action, results, information, and new action is overlooked by such an approach (Mercer, 1982).

Feedback processes in software development were also discussed by Belady and Lehman (in Wegner, 1980), (lehman, 1978), (Putnam, 1980), and (Zelkowitz et al., 1979).

A point which is important in particular to the application of deviation-amplifying feedback (DAF) to management, concerns the distinction between (1) the initial event (from outside a loop) which starts the deviation amplifying process in motion, and (2) the dynamics of the feedback process which perpetuates it. While the initial event is important in determining the direction of the

subsequent deviation amplification, the feedback process is more important to an understanding of the system (Ashton, 1976). The initial event sets in motion a cumulative process which can have final effects quite out of proportion to the magnitude of the original push. The push might even be withdrawn after a time, and still a permanent change will remain or even the process of change will continue without a new balance in sight. A further problem is that, after some period of time has elasped, it may be difficult, if not impossible, to discover the initial event. An interesting example of this has been provided by Wender (1968):

> ... a fat and pimply adolescent may withdraw in embarrassement and fail to acquire social skills; in adulthood, acne and obesity may have disappeared but low self-esteem, withdrawal, and social ineptitude may remain. Social withdrawal and low self esteem are apt to stay fixed because the DAF chain now operates: social ineptitude leads to rejection, which leads to lowered self-esteem, greater withdrawal, less social experience, and greater ineptitude. What has initiated the problem is no longer sustaining it. A knowledge of the problem's origin would not be expected to alter the currently operative loop unless such insight served to motivate behavioral change ...
> Finding the initial event (acne and obesity) may have less usefulness than understanding the current sustaining feedback mechanism. Furthermore, in some instances the initial event may have left no traces of its existance and may be undiscovered.

It is no wonder, then, that "most 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).

(b) The use of computer simulation:

So far, we have argued for an integrative model of software development, which in addition captures its information feedback systems. To stop here is not enough. We need a tool for handling the high complexity of such a model. There are two sources of high complexity; and computer simulation can be an effective tool to handle both:

> First,
>
> Managerial systems contain as many as 100 or more variables that are known to be relevant and believed 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 (Roberts, 1981).
>
> And second,
>
> The behavior of systems of interconnected feedback loops often confounds common intuition and analysis, even though the dynamic implications of isolated loops may be reasonably obvious. The feedback structures of real problems are often so complex that the behavior they generate over time can usually be traced only by simulation (Richardson and Pugh, 1981).

Simulation's particular advantage is its greater fidelity in modeling processes, making possible both more complex models and models of more complex systems. It also allows for vicarious experimentation.

Using the simulation model as an experimentation vehicle, should be particularly welcomed by the software engineering community. Several authors have "complained" about the lack of tested "ideas" in the software engineering field (Thayer, 1979), (Weinwurm, 1970). For example Weiss (1979) commented:

> ... in software engineering it is remarkably easy to propose hypotheses and remarkably difficult to test them. Accordingly, it is useful to seek methods for testing software engineering hypotheses.

Unfortunately, controlled experiments in the area of software development tend to be costly and time consuming (Myers, 1978). Furthermore, those who try it often find that " ... the isolation of the effect and the evaluation of impact of any given practice within a large, complex and dynamic project environment can be exceedingly difficult" (Glass, 1982).

In addition to permitting less-costly and less-time consuming experimentation, simulation models make "perfectly" controlled experiments possible. Which, as the following quotation shows, addresses the difficulty expressed by Glass above:

> The effects of different assumptions and environmental factors can be tested. In the model system, unlike real systems, the effect of changing one factor can be observed while all other factors are held unchanged.

> Such experimentation will yield new insights into the characteristics of the system that the model represents. By using a model of a complex system, more can be learned about internal interactions than would ever be possible through manipulation of the real system. Internally, the model provides complete control of the system organizational structure, its policies, and its sensitivities to various events. Externally, a wider range of circumstances can be generated than are apt to be observable in real life (Forrester, 1961).

Finally, the very process of constructing the simulation can be useful in several ways (Schultz and Sullivan, 1972):

1. Confrontation --- vague generalizations crumble when put to the test of modeling.
2. Explication --- assumptions must be made explicit, logical, and precise in order to build a simulation model.
3. Expansion --- the tendency to a holistic approach in simulation forces a broadening of one's horizon, a looking into other relevant fields for ideas.
4. Communication --- problem-oriented simulation lead to jumping of disciplinary boundaries, less parochialism. And,
5. Involvement --- it can be fun, the construction process motivates the modeler to attempt to fill in the knowledge gaps.

I.3. Research Accomplishments:

As mentioned in Section I.2., the objective of this

research effort is to enhance our understanding of, and gain insight into, the general process by which software development is managed. To achieve this objective we accomplished the following three tasks:

1. Developed an integrative system dynamics model of software development project management.
2. Conducted a case-study to test the model.
3. Used the model as an experimental vehicle to study/predict the dynamic implications of an array of managerial policies and procedures.

In the remaining part of this section, we will elaborate further on the above three research accomplishments.

Model Development:

The development of the integrative system dynamics model of software development project management constitutes the following set of accomplishments:

1. The mathematical formulation of a system dynamics model forces explication, i.e., structural relationships between variables must be explicitly and precisely defined. As such, the model sets the foundation for the development of a theory of software project management. Paraphrasing Dubin (1971):

> A theory is the attempt of a man to model some aspects of the empirical world ... A theory tries to make sense out of the observable world by ordering the relationships among 'things' that constitute the theorist's focus of attention in the world 'out there' ... The process of putting things or units together in lawful relation to each other establishes the fundamental building blocks out of which a theory is constructed.

2. The model complements and builds upon current research efforts, which tend to focus on the micro components (e.g., project management, programming, testing, productivity, ... etc.), by integrating our knowledge of these micro components into an integrated continuous view of the software development process, allowing us to identify and capture a richer set of interactions and interdepencies between the variables of software project management.

3. The model identifies feedback mechanisms, and uses them to structure and clarify relationships in software project management. While the significance and applicability of the feedback systems concept to the study of managerial systems has been substantiated in a large number of studies outside software engineering, it still remains largely foreign to the software engineering project management community. We, therefore, view our work as having an "educational" value to the software engineering community.

4. The high degree of explication required in the model helped us ferret out "knowledge gaps" in the literature. And a set of 27 interviews with software development managers in 5 organizations helped us fill these knowledge gaps. The model, therefore, incorporates new findings about the management of software project management (e.g., on manpower acquisition policies under different scheduling considerations).

Case Study:

The model was developed on the basis of both an extensive review of the literature and information gathered through a set of 27 interviews in 5 organizations involved in the production of software. After the model was developed, we then conducted a case-study in a sixth organization to test the model. The model was highly accurate in replicating the actual development history of the software project selected (by the organization) for the case study. Project variables tested included: the workforce level, the schedule, and the cost.

Experimentation

If "understanding" is the intellectual outcome of a theoretical model, then "prediction" is its practical outcome (Dubin, 1971). The model was used as an experimental vehicle to study/predict the dynamic

implications of an array of managerial policies and procedures. Four areas were studied: (1) scheduling; (2) Quality Assurance; (3) control; and (4) staffing. 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 quality assurance effort); and (3) provided new insights into software project phenomena (e.g., "90 % syndrome").

I.4. Thesis Outline:

Each chapter of this thesis may be considered in terms of its relationship to the model, which is the focus of the study.

Chapters (I) and (II) serve as a background and an introduction. In Chapter (I), we discussed the problems and challenges of software development project management. We also argued for the integrated System Dynamics modeling approach, as a vehicle to address those problems and challenges.

In Chapter (II), we conduct a survey of the literature. The presentation is conveniently broken into two sections. First, we survey the System Dynamics

literature that addresses the general area of project management. This is a particularly appropriate starting point, since it is this research track that provided the first stimulant to our work. The second part of the chapter, is a survey of the software engineering literature to see what has been proposed /done to understand and solve the problems of software project management.

Chapter (III) is on model development. In it we discuss in detail the development, structure, and equation formulation of the model. The model has four sectors. At the heart of the model is the software production sector, where software production activities such as coding and testing are modeled. The project management activities comprise the remaining three sectors: planning, human resource management, and control.

In Chapter (IV) we discuss the results of a case study conducted to test the model's ability to replicate the development history of a completed software development project. Project variables tracked included: the workforce level, the schedule, and the cost.

In Chapter (V), the model is used as an

experimentation vehicle to study/predict the dynamic implications of an array of managerial policies and procedures. Four areas are studied: (1) scheduling; (2) control; (3) Quality Assurance; and (4) staffing.

Finally, Chapter (VI) concludes the thesis with a summary of findings and suggestions for further research.

II. REVIEW OF RELEVANT LITERATURE

In this chapter two bodies of literature relevant to our research are reviewed. The first is the System Dynamics literature that addresses the general area of project management. This is a particularly approriate starting point, since it is this research track that provided the first stimulant to our work. In the second part of the chapter, we review the software engineering literature in the area of software development project management. Thus, while in the first section we look at research works that share with us our basic research approach, in the second section we turn our attention to those that share with us our research objective (i.e., the understanding of the software development process).

II.1. System Dynamics Modeling of Project Management:

Professor Edwards B. Roberts, of MIT's Sloan School Of Management, has been the pioneer of this research effort, as

well as continuing to be its major driving force. His doctoral dissertation on "The Dynamics of Research and Development," in 1962 (which was also published as a book) was the first scholarly effort to apply the then young System Dynamics methodology to the project management area (within an R&D environment). It still continues to be the most comprehensive treatment of the subject. Since then, and primarily in his capacity as a thesis advisor, he continues to play an active "guiding" role in the field's advancement. And which, as a result, continued to focus on the study of R&D type projects. Roberts' thesis work together with that of his MIT students, constitute the bulk of this body of research.

It might be interesting to make a brief digression here and explain how and why this body of research, lying at the overlap between the System Dynamics and the Management of R&D literatures, first attracted our attention and interest. It was (surprisingly) while we were surveying the latter and not the former. At the time, feeling frustrated by the lack of innovative activity in the area of software management, we decided to look into other more established fields for new ideas. The management of R&D was the obvious first choice. And for good reason. It is the area we found to be most often likened, in the software engineering literature, to software production. For example, paraphrasing Gehring and Pooch (1977):

> The stages of research and development are similar in many respects to the stages of software analysis and design. First, the determination of what the system is to do (specification of outputs and inputs) is very ill-defined, making the estimation of the time and cost of its development uncertain (like the research stage). Second, the specification of how inputs (file specification, programmning) is easier to estimate (like the development state). These similarities suggest that a good many managerial practices and procedures from the latter may be applied to the former.

The similarity in project cost estimation, between the two fields, was also suggested by Wolverton, in his highly referenced 1974 paper, when he wrote: "The general principles involved in pricing large R&D efforts of any kind ... apply to large software development as well."

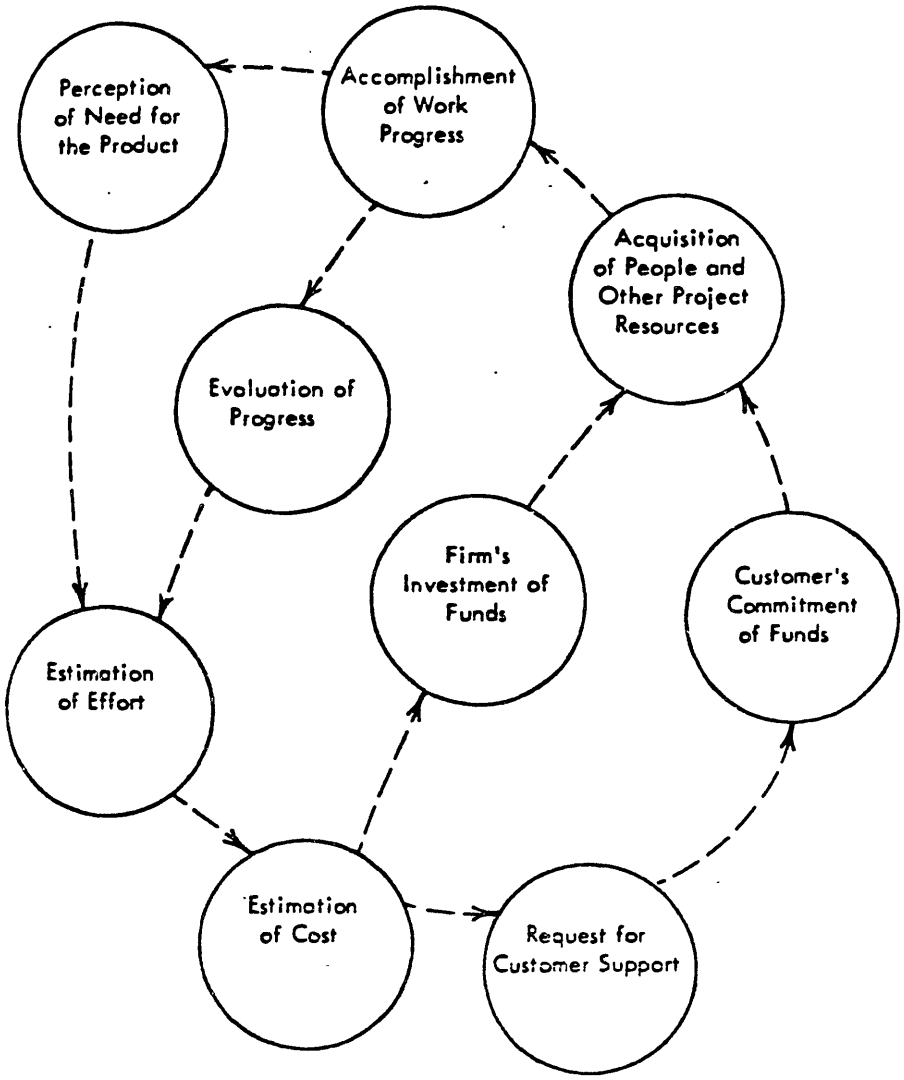
Also, it is interesting to note, that Putnam's celebrated SLIM model for software cost estimation (Putnam, 1980) is based on the R&D work of Peter Norden. Norden had showed that R&D projects have a well defined manpower pattern of the Rayleigh form (Norden, 1963) . When Putnam "adapted" Norden's findings (on R&D projects) to the software environment, he found that, here too, manpower application follows the same Rayleigh pattern.

So, with great enthusiasm and anticipation we embarked on a survey of the R&D literature. And read Roberts' doctoral thesis. End of digression.

While perhaps interesting as a historical perspective on our research effort, the above digression serves an additional purpose. For, it suggests that our stated argument for the relevance of the System Dynamics modeling work of R&D project management to our own, namely, their sharing of the same research methodology and approach, is really a conservative one. The two areas have, in fact, much more in common. And with this in mind, we now resume our review of the literature.

As stated above, Roberts' System Dynamics model of R&D project management, continues to be the most comprehensive work published in the area. The model traces the full life cycle of a single R&D project. And it incorporates the interactions between the R&D product, the firm, the customer, and the processes relating to the nature of the work itself. Figure II.1. (from Robert's thesis) is an overview of the model's sectors, and the interrelationships among them.

Rather than delve into a detailed discussion of Roberts' R&D model, we will limit out discussion of his work to those aspects of the model which we found particularly relevant to the study of software project management. Specifically, we will present some of his models' conceptual building-blocks (i.e., his assumptions/ findings about R&D projects). And to underscore the correspondance to the software production environment, we will append the presentation with "excerpts"



Over-all organization of system equations.

Figure II.1

from the software engineering literature.

On project Planning

Roberts:

No unerring formula can be used to estimate the total number of man-years required to carry out a given (R&D) project. This kind of general statement reflects the inherent nature of research and development: The exact character of a specific task is indefinite, (and) the specific technical requirements are uncertain ...

The Software Engineering Literature:

* ... quantitative software engineering has not progressed to the point that we can even begin to provide (software sizing) formulas. And it is not clear that we will ever get very close to such an ideal (Boehm, 1981).

* We lack the means " ... to provide clear, concise, and unambiguous statements of user requirements ... The problem here again has to do with the "absence" of a clear understanding on the part of both software users and developers as to what can be accomplished with software" (DeRose and Nyman, 1979).

* The production of software is not a deterministic activity. Product specifications are liable to be shifted (Trichritzis, 1977).

Roberts:

Two factors significantly influence the initial estimate of the job size: (1) the firm's previous experience; and (2) the general over-all tendency to underestimate the job size.

The Software Engineering Literature:

* ... when methods of estimating are ranked, the list

is headed by the Experience Method ... This approach takes advantage of experience on a similar job ... The major problem in the method is that it does not work on systems larger than the base used for comparison. System complexity grows as the square of the number of system elements; therefore, experience with a small system cannot account for all the things that will have to be done in a large system. Neither will the Experience Method apply to systems of totally different content" (Aron, 1976).

* The software undersizing problem is our most critical road block to accurate software cost estimation ... there are no magic formulas that we can use to overcome the software undersizing problem. In the absence of any such formula, it is important to understand the major sources of the software undersizing problem ... A major (reason) is a strong tendency to underestimate the size of support software (e.g, compilers, tools, utilities), which for large operational systems is generally three to five times as large as the operational software (Boehm, 1981).

On the Management of the Human Resource:

Roberts:

* Whatever the know-how developed in solving the R&D project problems, some time is required for it to be adequately, absorbed. Then, as the experiences accumulate, the firms' engineers supplement their nonproject skills with these new, more specific insights and approaches to the task.

The Software Engineering Literature:

* Programmers become more effective during larger programming operations because of "learning." The programmer gains familiarity with program logic, coding notation, testing restrictions, and other requirements as he progresses through each major activity in the programming methods (Shell, 1972).

Roberts:

* Above a certain level, the assignment of additional

personnel to a large project may not only reduce total time proportionality, but in fact may increase total time to accomplishment.

The Software Engineering Literature:

* Increasing the size of a software team increases the amount of software produced per unit time, up to a point. Then the problems of communication among the programmers begin to dominate the project and reduce the amount of software being produced (Boebert, 1979)

And finally, on the Control of Progress:

Roberts:
(Control) problems ... result from lack of tangible, precise measurement in R&D ..."

The Software Engineering Literature:
* Abstraction, or intangibility, is a management challenge for such rudiments as recognizing process, exhibiting results, and communicating between packets of work. And compounding this is lack of hardware-like measures ... " (Sampson).

* It is difficult to measure performance in programming ... (And) it is difficult to evaluate the status of intermediate work such as undebugged programs or design specifications and their potential value to the completed project (Mills, 1983).

Roberts:
* One particular difficulty is that, during the very early phases of a project, milestones have a tendency to be less precisely definable, and hence less accuratily measurable, than during later phases of the project ... The shortcomings of the concept, "percent complete," were sufficiently great to negate its value. While projects tended to make rapid progress towards completion when work first began, it took an inordinately long time to get from 90 percent to 100 percent.

The Software Engineering Literature:

* In the early stages of a project, it is difficult to distinguish between 5% completion and 10% completion, yet the resultant projection can vary 100% based on which number is chosen (Donelson, 1976).

* One frequent difficulty stems from an over-reliance on individual percent-complete estimates as indicators of project progress (Boehm, 1981).

* (This) method of estimating progress typically leads to estimates of the fraction of work completed which increase as originally planned until a level of about 80-90% is reached. The programmers' individual estimates then increase only very slowly until the task is actually completed (Baber, 1982).

It is clear from the above presentation that some of the problems that Roberts' model was built to address do resemble some of those we are struggling with today in the software engineering area. It is no wonder then, that we felt (and did find) that the approach he effectively used i.e., Systems Dynamics Modeling, to be an effective tool for addressing the problems of software development project management.

As we mentioned in the beginning of this discussion, Roberts' thesis was to become the foundation for further System Dynamics studies of the R&D project management area. One obvious extension was to study multi-project environments. In such an environment project competition for company resources becomes a significant dimension. Two such multi-project models are those of Nay (1965) (a four-project model) and Kelly (1970) (a two-project model). In both models the focus remained, as was in Roberts, on project life cycle behavior. Edelman's (1975) work, however, is a

departure from that. While building on Nay's model, he chose to focus, instead, on the allocation and utilization of manpower resources and the effects of the management system design on effectiveness.

Richardson (1982) took still a different tack. Rather than focusing on a project, he focussed, instead, on the development group. His model, therefore, does not trace the life cycle(s) of one or more projects; rather, it reproduces the dynamics of a development group over an eight year period as a continuous stream of products are developed and placed into production. The model focuses on the number of products under development, the use of resources required, and the aggregate average product development time.

Finally, several more recent models are emphasizing the role of rework in project management. Rework can be caused by errors committed in the earlier phases of a project (e.g., design errors of a VLSI circuitry) that escape detection until later in the projects' life cycle. Of course, the longer an error goes undetected, the more extensive the necessary rework and the greater the cost. Changing design specifications after development begins, also generates the need for rework. Cooper (1980), describes a large system dynamics study of cost overruns in a shipbuilding contract. The study showed that the rework required by frequent design changes imposed by the Navy were the major contributing

factor to a $500 million dollars overrun. Undiscovered rework is also the focus of the simple R&D project models in Roberts (1981b) and (Richardson and Pugh, 1981).

II.2. Software Engineering Project Management Literature Review:

As we stated in chapter (I), the focus of this research is on software development project management, and our objective is to improve our understanding of it. In this section we review the software engineering literature on project management, to assess the current "state-of-understanding," and the means/tools used to achieve it.

We will begin by reviewing overview-type models and frameworks. This will then be followed by separate discussions on software project planning, human resource management, and control i.e., the three project management subsystems that together constitute the project management activities in our model (as will be explained in chapter III).

II.2.1. Overview Models and Frameworks:

Richard Thayer's 1979 Ph.D dissertation at the University of California at Santa Barbara on "Modeling a

Software Engineering Project Management System," is a fitting starting point for this discussion. For one, it probably was indeed ... "the first attempt to completely model a software engineering project management system" (Thayer, 1979). But, perhaps more important, if we judge from the number of publications it generated (one in IEEE Transaction on Software Engineering (Thayer et al, 1981), two in Computer (Thayer et al, 1980) and (Thayer et al, 1982), plus several conference papers), the thesis' results did have a significant impact on the software engineering community.

Thayer's research goal was twofold: (1) to develop and verify "a generalized descriptive management model of a software engineering project management system," and (2) to "identify and verify the major issues of software engineering project management."

To develop his model, he first identified the various functions, actions, procedures, and tools used, or proposed for use, in managing a software engineering project. This was done on the basis of a literature survey as well as his own personal experience. He then superimposed these functions, actions, procedures, and tools on the "classic management model," i.e., that breaks the management activity into the five functions of planning, organizing, staffing, directing, and controlling.

The "skeleton" of his model is shown in Figure (II.2). Each of the shown eight model sections, i.e., "Project Identification," "Requirements and Constraints," "Planning," ... etc., was then expanded further. For example, his "detailed planning Section" is shown in Figure (II.3.a), together with the set of assumptions he used to formulate it (in Figure (II.3.b).

As we mentioned above, in addition to developing the model, Thayer had a second objective, namely, to "identify and verify the major issues of software engineering project management." And, it is interesting to note, that even though Thayer considered the development of the model to be the most important contribution of his work, it was his findings here that has, in fact, generated all his above mentioned publications.

To identify the major issues of software engineering project management, his first step was to review the literature for software engineering problems. Then, by using the software engineering delivery and success model shown in Figure (II.4) he hypothesized which of these problems can most affect the success of software delivery. These, he believed were the major issues.

The issues were then reworded as problems as seen by the project manager, and classified on the basis of the "classic"

Overview Model of a Software Engineering
Project Management System

GENERAL MANAGEMENT AND PRODUCTION MODEL [5]	SOFTWARE ENGINEERING PROJECT MANAGEMENT AND PRODUCTION MODEL
Project Identification	Program Identification Hardware Identification Customer Identification Contract Identification Cost & Schedule Identification Software Identification Complexity Identification Data Base Identification
Requirements & Constraints	Requirement Specifications Document Requirements Customer Constraints
Planning	Planning and Scheduling Quality Assurance Program
Organizing	Preorganization Function Project Management Organization Software Engineering Proj Team
Staffing	Project Manager Staffing Software Development Staff Staff Support Training
Directing/Monitoring	Responsibility and Authority Management Techniques Assignment of Work
Controlling	Project Control Reporting Formal Reviews Configuration Management Informal Reviews and Walk-throughs
Deliveries & Successes	Schedule Cost Meets Requirements Meets Reliability Standards Meets Maintainability Standards Meets Useability Standards

Figure II.2

Planning Model

GENERAL MANAGEMENT FUNCTIONS [6],[7]	PROJECT MANAGEMENT ACTIVITIES
Analyze Requirement	Analyze inputs and output requirements, functions of the system, and deliverables.
	Determine hardware and system software restrictions.
	Determine user identification and type contract.
	Determine size, complexity, and user or company constraint.
Set Objectives	Determine and establish success criteria.
	Determine attributes of delivered software: reliable, maintainable, useable, etc.
Forecast	Determine cost and schedule to deliver software.
Set Procedures	Select planning and project control tools and techniques.
	Develop quality assurance plan.
	Select design, programming, and testing tool, technique, and methods.
Develop Strategies	Same
Develop Policies	Same
Program	Determine priority and milestones for events.
Budget and Allocate Resources	Budget, locate and secure resources: funds, programmer/analyst, computer time, etc.

Figure II.3 a

A separate organization from the development organization would perform the planning and scheduling (this is also an element of the organizing model)

Planning would be accomplished through the use of formal planning guides, methods, and tools

The plan, no matter how well accomplished by the planning group, would be modified by either the senior manager or the customer

Planning documentation would be prepared

The planning function would be a formal function with time allocated for planning

Modular planning design and delivery techniques would be used on the software development project

The planning function would include a software quality assurance program

Each project would use some of the tools, techniques and procedures known as "modern programming techniques"

Software development tools, techniques and aids would be used on the software development project

Software test tools, techniques, and methods would be used in the software development project.

Figure II.3b

Software Development Delivery and Success Model

- o Deliveries:
  - Software
  - Documentation
- o Success Attributes:
  - On time
  - Within resources
  - Meets requirements
  - Useable
  - Reliable
  - Maintainable

Figure II.4

management model of planning, organizing, staffing, directing, and controlling. He found that "By far, the two dominant (problematic) activities are planning and controlling, which together (accounted) for 80 % of the issues, with planning alone involving ten isues." The 20 issues he identified are shown in Figure (II.5).

To verify his hypothesized issues he did two things. First he conducted "an opinion survey with a selected sub-set of the computer community." This included: "technical leaders in computer science," "software engineering authors," "project managers," "R&D personnel," and "software engineering educators." (Two hundred and ninety four replies were received.) The surveyees were asked to comment on whether or not they felt each of the hypothesized problems was a critical problem, an important problem, not important, not a problem at all, or lastly, disagree with the hypothesis completely and by the way it was stated. The surveyees were, in addition, asked to state how they would (or did) solve the problem.

The 13 starred (*) issues in Figure (II.5) were the ones verified on the basis of this survey, (Verification meant that at least 70% of the respondents felt that the issue was either "critical" or "important".)

(Note: Most of his surveyees either came from large

Twenty hypothesized problems in SEPM

Planning

1. *Requirements:* Requirement specifications are frequently incomplete, ambiguous, inconsistent, and/or unmeasurable.

2. *Success:* Success criteria for a software development are frequently inappropriate, which result in "poor-quality" delivered software; i.e., not maintainable, unreliable, difficult to use, relatively undocumented, etc.

3. *Project:* Planning for software engineering projects is generally poor.

4. *Cost:* The ability to estimate accurately the resources required to accomplish a software development is poor.

5. *Schedule:* The ability to estimate accurately the delivery time on a software development is poor.

6. *Design:* Decision rules for use in selecting the correct software design techniques, equipment, and aids to be used in designing software in a software engineering project are not available.

7. *Test:* Decision rules for use in selecting the correct procedures, strategies, and tools to be used in testing software developed in a software engineering project are not available.

8. *Maintainability:* Procedures, techniques, and strategies for designing maintainable software are not available.

9. *Warranty:* Methods to guarantee or warranty that the delivered software will "work" for the user are not available.

10. *Control:* Procedures, methods, and techniques for designing a project control system that will enable project managers to successfully control their project are not readily available.

Organizing

11. *Type:* Decision rules for selecting the proper organizational structure; e.g., project, matrix, function, are not available.

12. *Accountability:* The accountability structure in many software engineering projects is poor, leaving some question as to who is responsible for various project functions.

Staffing

13. *Project manager:* Procedures and techniques for the selection of project managers are poor.

Directing

14. *Techniques:* Decision rules for use in selecting the correct management techniques for software engineering project management are not available.

Controlling

15. *Visibility:* Procedures, techniques, strategies, and aids that will provide visibility of progress (not just resources used) to the project manager are not available.

16. *Reliability:* Measurements or indexes of reliability that can be used as an element of software design are not available and there is no way to predict software failure; i.e., there is no practical way to show the delivered software meets a given reliability criteria.

17. *Maintainability:* Measurements or indexes of maintainability that can be used as an element of software design are not available; i.e., there is no practical way to show that a given program is more maintainable than another.

18. *Goodness:* Measurements or indexes of "goodness" of code that can be used as an element of software design are not available; i.e., there is no practical way to show that one program is better than another.

19. *Programmers:* Standards and techniques for measuring the quality of performance and the quantity of production expected from programmers and data processing analysts are not available.

20. *Tracing:* Techniques and aids that provide an acceptable means of tracing a software development from requirements to completed code are not generally available.

Figure II.5

companies or obtained their knowledge from data processing in large companies. Therefore, it can be assumed that the viewpoint as to whether or not a given problem was critical, important, or not important at all, was the viewpoint of the large DP shop.)

The second verification step was through a second separate survey of 60 software development projects in the aerospace industry. And he checked for whether "the condition described in the major issue existed, and (that) the existence of the condition was a problem to the project manager ... If the data substantiates (this) the hypothesized issue is labelled a problem."

Nine of the 20 major issues (marked with + in Figure (II.5)) were verified as problems, two were inconclusive, and nine were not problems. As a result, six major issues concerning planning and one concerning controlling were judged conclusively as problems by both surveys.

Thayer noted with interest, though, that "there is some disagreement between the general data processing community and the project managers and developers." Which prompted him to comment: "The fact that these two groups do not, in general, agree on the major issues is in itself a fundamental problem of project management."

In addition:

> Similar to the problem in identifying the major issues, the computing community is divided on the solutions to the major problems. there are no well defined software management techniques to guarantee a successful software delivery.

Finally, we conclude our discussion of Thayer's work with some of his own concluding remarks:

> Future research should continue to "refine" this model ... This model, as a first attempt, has many ommissions and frequent generalizations. Similar research projects, using a different approach, could fine-tune this model and find more elements with a full range of values for each element.
>
> This research identified a number of major issues of software engineering project management and proposed a number of solutions. What is needed is a good definitized experimentation method that can be used as a test bed for validating new project management tools, techniques, and procedures, ... etc.
>
> There is still a long way to go, this is only the beginning.

In another doctoral thesis, Riehl (1977) developed a "planning and control framework to assist in the management of computer-based information systems development in large organizations." The general scope of the research encompassed two basic avenues of endeavor: (1) an extensive literature survey to compile "those concepts and practices that are advanced by authorities in the field of computer-based information systems and electronic data processing management," and (2) a determination of those policies and procedures actually employed in practice by

companies "judged to be effective managers of computer-based information systems."

His model, termed the "Composite-Working Model," consisted of some 25 "principles " and 50 "issues." Principles are those "specific concepts, policies, and procedures upon which general agreement was found to exist in the literature and in the observed practices of the (5) companies investigated." Issues, on the other hand,

"identify those proposed practicies about which disagreement or uncertainty exists within the literature or which are the subject of clear divergences between the concepts advanced in the literature and the majority practices of the firms in the research." The principles and issues were classified into 4 categories: strategic planning, project planning, project control, and organizational behavior considerations.

For purposes of reference, a summary of the major categories of the Composite-Working Model is presented in Figure (II.6). As an illustration, consider the "Consensus Principle V (PP): Project Plan," within the "project planning " category. It was included because "the importance of a project plan is widely recognized in the source literature ... (and) the research findings supported the principle." Furthermore, "A single issue was generated

concerning the degree of detail that should be included in the project plan. Brandon, for example, proposes a very comprehensive scheme based on an automated system. Other writers generally provide considerably fewer details on the subject." A similar disagreement was observed between the companies studied.

In his conclusion, Reihl asserts that he has met his research goal, namely, to develop "a planning and control framework to assist in the management of computer-based information systems development in large organization, by identifying those practices and procedures which are both advocated in the literature as well as used by (selected) large business organizations with a reputation for effective computer-based information systems management."

Instead of focusing, as the above two pieces of research did, on the set of issues that are common among software development projects generally, McFarlan's (1974) research focus was on the differences between projects. "One conclusion from my research stands out," he wrote, and that was:

> A monolithic approach to systems and programming project managment is unlikely to produce the most satisfactory results. There are critical differences in project composition ... which influence the mix of tools that should be brought on its management.

SUMMARY OF THE COMPOSITE-WORKING MODEL

Strategic Planning

Consensus Principle I(SP): Master Systems Planning

Issue A: Structure for Planning
Issue B: Type of Planning

Consensus Principle II(SP): Management Involvement

Issue A: Top Management Involvement
Issue B: User-Management Involvement
Issue C: Chief Executive Officer Involvement

Consensus Principle III(SP): Master Systems Plan

Issue A: Planning Details

Consensus Principle IV(SP): Planning Coordination

Issue A: Planning Integration

Consensus Principle V(SP): Provision for Change

Issue A: Means for Achieving Change

Project Planning

Consensus Principle I(PP): System Development Life Cycle

Issue A: Description of the System Development Life Cycle

Consensus Principle II(PP): Feasibility Study and Project Proposal

Issue A: Analysis of Alternative Designs
Issue B: Feasibility Study

Consensus Principle III(PP): Economic Analysis

Issue A: Treatment of Reliability
Issue B: Present Value Discounting
Issue C: Estimating Intangible Benefits
Issue D: Approval Criteria

Figure II.6

(Project Planning--Continued)

Consensus Principle IV(PP): Project Management

- Issue A: Assignment of Project Manager
- Issue B: Project-Status Audit
- Issue C: Project Thresholds
- Issue D: Project Establishment

Consensus Principle V(PP): Project Plan

- Issue A: Project Plan Detail

Consensus Principle VI(PP): Project Control Reporting

- Issue A: Reported Information
- Issue B: Management Review

Consensus Principle VII(PP): Estimation Process

- Issue A: Estimating Methods
- Issue B: Reliability of Estimates

Consensus Principle VIII(PP): Change Control

- Issue A: Review of Changes
- Issue B: Limiting Impact of Changes

Consensus Principle IX(PP): System Development Standards

- Issue A: Form of Standards

Consensus Principle X(PP): Cost Allocation

- Issue A: Method of Cost Allocation
- Issue B: Influence on User Behavior

Project Control

Consensus Principle I(PC): User-Management Control

- Issue A: Level of Management Control
- Issue B: Key Check-Points
- Issue C: Form of Check-Point Reviews

Consensus Principle II(PC): Information Requirements Definition

- Issue A: Methods of Requirements Identification
- Issue B: Requirements Validation

Figure II.6

(CONT.)

(Project Control—Continued)

Consensus Principle III(PC): Functional Specifications

Issue A: User Participation
Issue B: Conversion Plan

Consensus Principle IV(PC): Performance Criteria

Issue A: Performance Criteria Specifications

Consensus Principle V(PC): Detailed Design Specifications

Issue A: User Participation

Consensus Principle VI(PC): System Implementation

Issue A: User Participation

Consensus Principle VII(PC): System Testing

Issue A: User-Management Involvement
Issue B: User Representative Participation

Consensus Principle VIII(PC): Conversion and Cut-Over

Issue A: Conversion Organization
Issue B: Management Control

Consensus Principle IX(PC): Post-Implementation Audit

Issue A: Conduct of Audit
Issue B: Documentation Audit

Organizational Behavior Considerations

Consensus Principle I(BC): User Acceptance

Issue A: Intergroup Communications
Issue B: Personnel Management
Issue C: User-Management Involvement
Issue D: User Participation and Control of Change
Issue E: Awareness of User Attitudes

Figure II.6

(CONT.)

He identified three "important" dimensions for characterizing software development projects. These are: (1) The degree of predetermined structure inherent in the project (he defined a highly structured project to be "one where the processing routines and outputs of the system are so determined by the project's environment in advance that there are little or no design options open to the system architect or user"); (2) The degree of company-relative computer technology implicit in the project (a high "company-relative technology" project is defined as "one which involves complex hardware-software features which have not been dealt with previously in the organization"); And (3) Project size in terms of man-years of effort or manpower dollars of expenditures ("In this context a $50,000 project will be considered small while a $1 million project will be considered large").

Figure (II.7) shows how, using these dimensions, a project may be classified as falling into one of eight different categories.

As stated above, McFarlan felt that a project's classification should "influence the mix of tools that should be brought on its management." To show how, he first provided a scheme to divide project management tools into four main groups. The four groups are: (1) Formal integration procedures with users of the project's output,

Classification of Systems and Programming Project Types

		Degree of Structuredness	
		High	Low
Degree of Company-Relative Technology	Low	I. LARGE PROJECT	V. LARGE PROJECT
		II. SMALL PROJECT	VI. SMALL PROJECT
	High	III. LARGE PROJECT	VII. LARGE PROJECT
		IV. SMALL PROJECT	VIII. SMALL PROJECT

Figure II.7

who are located outside the EDP department (e.g., a formal User-EDP project advisory committee); (2) Formal integration procedures within the EDP design team and between the various units of the EDP department (e.g., formal flow charts and other documentation to highlight interfaces between key systems components); (3) Formal planning tools (e.g., PERT or CPM); and (4) Formal control tools (e.g., regular use of formal post-audit procedures).

The final step was to put the two pieces together into what he called a "contingency theory" of EDP systems and programming project-management. The outcome is exhibited in Figure (II.8).

At still a higher level of specifity are the research efforts to delineate phase differences <u>within</u> the life of a <u>single</u> project. According to McKeen (1981):

> The dominant organizing framework for application system development is the life cycle concept. This methodology apportions the total developmental effort into identifiable stages --- each stage representing a distinct activity characterized by a starting point, an ending point, and deliverables in concert with an express purpose.

The life cycle model was formally acknowledged as an important element in systems development by its inclusion in the information system curricular proposed by the ACM Curriculum Committee on Computer Education for Management

Project Types	Project Description	External* Integ.	Internal Integ.**	Formal Planning	Formal Control
I	High Structure, Low Tech., Large	Low	Medium	High	High
II	High Structure, Low Tech., Small	Low	Low	Medium	High
III	High Structure, High Tech., Large	Low	High	Medium	Medium
IV	High Structure, High Tech., Small	Low	High	Low	Low
V	Low Structure, Low Tech., Large	High	Medium	High	High
VI	Low Structure, Low Tech., Small	High	Low	Medium	High
VII	Low Structure, High Tech., Large	High	High	Low+	Low+
VIII	Low Structure, High Tech., Small	High	Medium	Low	Low

* No attempt is made here to suggest how external integration may shift over time as the user becomes more sophisticated through experience. My research suggests this may be important.

This table highlights the importance of external integration in getting user commitment to a project structure. It does not explicitly address his important role in enabling the EDP technicians to adequately understand the process to be automated. This appears to be important even in highly structured situations. Thus even these projects which are ranked low in the above table in external integration, may involve considerable user liaison of the fact finding sort.

** This does not identify the sharp split in the mix of the tools in internal integration identified in the text. Later work may split this into two categories.

Figure II.8

(Ashenhurst, 1972). In recent years, many books and papers on the life cycle concept have been published (e.g., (Boehm, 1981) (Gaffney, 1980) (Metzger, 1981) (Thomsett, 1980) (Yourdon, 1982)).

According to Davis (1974), the foundation for the life cycle concept is that application systems need to undergo a similar process when they are conceived, developed and implemented. Further, neglecting any portion of the life cycle activities may have serious consequences for the end result. The contribution of the life cycle concept to systems development is described by Davis as follows:

> Information system development involves considerable creativity, the use of the life cycle is the means for obtaining more disciplined creativity by giving structure to a creative process. The life cycle is important in planning, management, and control of information system application development.

The steps or phases in the software development life cycle are described differently by different authors, but the differences are primarily in amount of detail and number of categorizations. A common breakdown is given by Glass (1979):

Requirements/Specifications

Design

Implementaion

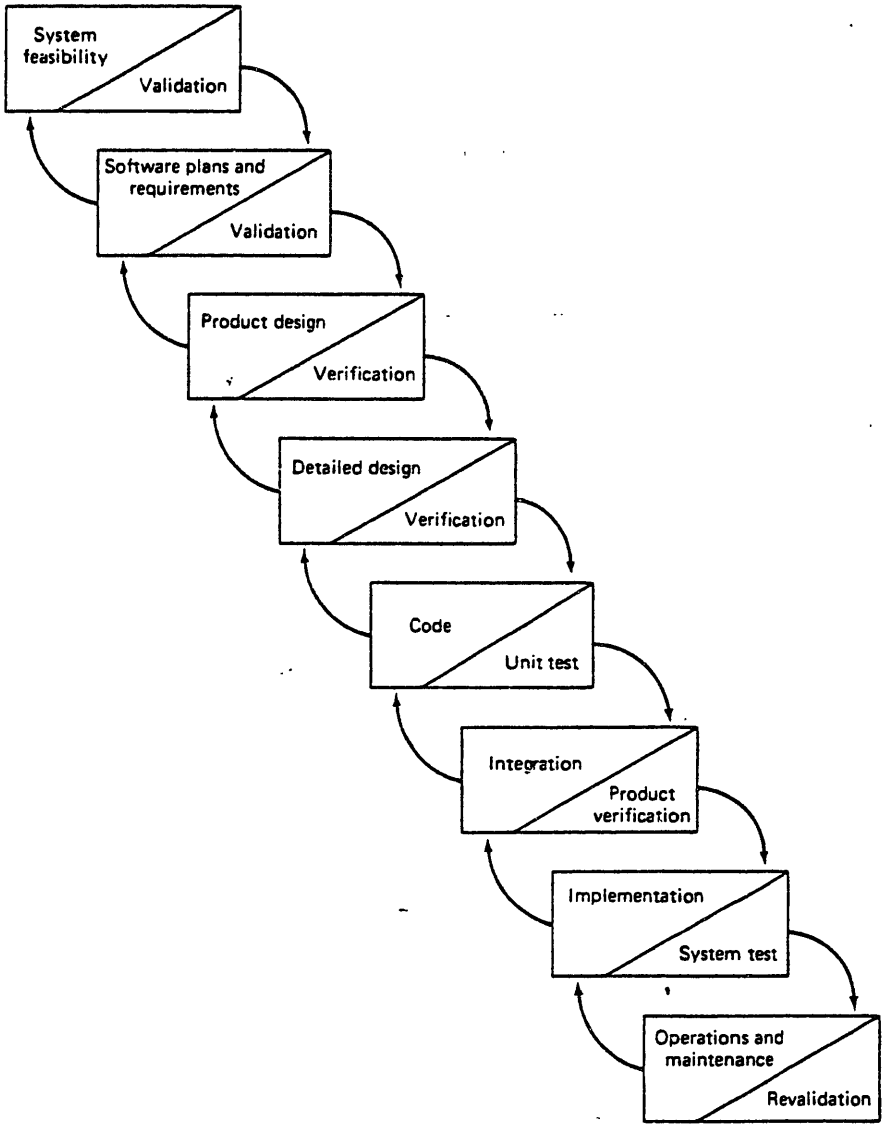
Checkout

Maintenance

The mere enumeration of the phases is not, however, an adequate model of the software life cycle because it "conceals" the iterative nature of the software development process (Artzer and Neidrauer, 1982) (A16). The life cycle is not followed in 1,2,3 fashion, rather "the process is iterative so that, for example, the review after the system design phase may result in going back to the beginning to prepare a new design" (Davis, 1974). Boehm's (1981) "waterful" model, shown in Figure (II.9), emphasizes this highly iterative nature of software development, indicated by the feedback arrows from each phase to its predecessor(s).

In addition to the identification of the component phases and activities in the software development process, it is important to evaluate the relative consumption of resources by each of these activities in order to obtain a proper perspective of the nature of the overall process. Numerous authors have presented figures indicating life cycle resource consumption by phase. In Figure (II.10) a comparison of three author's results done by McKeen (1981) is exhibitted. Commenting on the figure, McKeen stated that:

> Substantial differences do exist particularly in the coding and testing phases of development. These differences may be due to the inherent attributes of the systems being developed, or to terminological variations, or to a combination of both of these. In the absence of a careful description of the systems and



The waterfall model of the software life-cycle

Figure II.9

Comparison of Effort Breakdown by
Activity for Different Authors

Life Cycle Phase/Activity	Percentage Resource Allocation		
	Davis	Zelkowitz	Shaw
Analysis[1]	25	20[2]	25
Design	20	15	10[3]
Coding	25	45[4]	30
System Test	n/a[5]	20	5
Implementation	15	n/a[6]	19

Notes:
1. Analysis encompasses all development activity prior to detailed design.
2. The analysis effort is probably understated. If, as speculated, this data is derived from system developments in a military environment, then initial activity such as feasibility analysis and preliminary systems study has been excluded.
3. Using the authors definitions, the activities of system specifications and technical requirements constitute detailed design activities as used here.
4. Coding effort and module test effort were combined. Programmers are typically responsible for unit, or module, testing each portion of the system they have coded.
5. This activity has been subsumed within the conversion stage by Davis.
6. This activity is not reported.

Figure II.10

> the environment in which they were developed ... the generalization of results beyond the immediate environment in not possible.

The above views are shared by others in the literature. For example, Kustanowitz (1977) supports the notion that system size effects the life cycle resource distribution as shown in Figure (II.11). While Myers (1978) reported on a study in Boeing which showed that "the costs were shifted into earlier stages (of the life cycle) by the use of modern programming practices."

The life cycle resources distribution issue plays an important role in the estimation of resource allocation for software development. This role will be discussed in some detail within our review of the literature on project planning next.

II.2.2. Planning:

In his IEEE Tutorial on Software Management, Reifer (1979) defined planning as follows:

> It is deciding in advance what to do, how to do it, when to do it, and who is to do it. It is setting objectives, breaking the work into tasks, establishing schedules and budgets, allocating resources, setting standards, and selecting future courses of action. It bridges the gap from where we are to where we want to be.

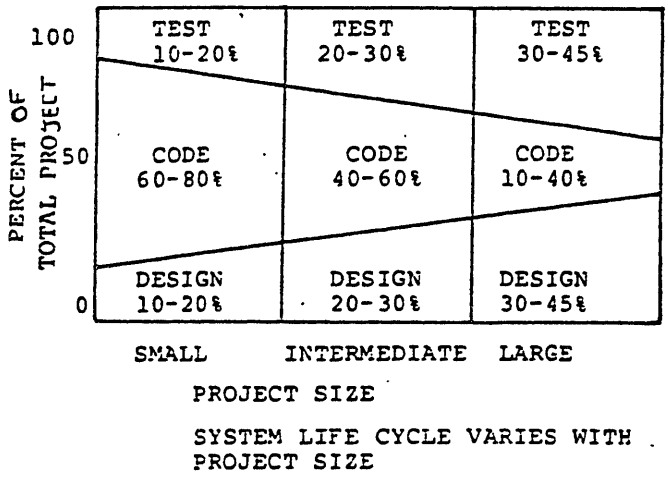

100
50
0
PERCENT OF TOTAL PROJECT
TEST 10-20%
TEST 20-30%
TEST 30-45%
CODE 60-80%
CODE 40-60%
CODE 10-40%
DESIGN 10-20%
DESIGN 20-30%
DESIGN 30-45%
SMALL
INTERMEDIATE
LARGE
PROJECT SIZE
SYSTEM LIFE CYCLE VARIES WITH PROJECT SIZE

Figure II.11

There is abundant support in the software engineering literature for the import of planning in the management of software projects (McGowan, 1978) (Thayer, 1979). Unfortunately, however, there is as ample an evidence for its poor standing (Boehm, 1980), (Jones and McLean, 1970), (Keider, 1974), (Metzger, 1981), (Pressman, 1982), (Thayer et al, 1981). Gehring and Pooch (1980) support both assertions in a single "breath:"

> One universal management principle, for example, has been called the "principle of the primacy of planning." In other words, planning has primacy over the other managerial functions of organizing, staffing, directing, and controlling. Thus, the degree of control over a programming project can be no greater than the extent to which adequate plans have been made for the project ... Inadequate planning is the primary reason for loss of control on many computer programming projects. It is not the comparative newness of the computer programming process, difficulties with programmers, or technical factors --- It is simply that programming projects are not adequately planned in the first place.

When Thayer (1979) surveyed the software engineering literature to identify the "major problems of software engineering project management," he ended up with 20 "hypothesized" problem areas. Of these, a full fifty percent (or 10 problems) were identified as being planning-type problems (see Figure II.5). And when he proceeded to verify his list, the dominance of planning-type problems was even more "impressive:" of the seven problem-areas that were verified, six were planning-type problems (the seventh was in the control area).

In addition, Thayer's work, which incorporated a survey of 60 software projects (in the aerospace industry), shed some light on the planning activity. For example, he reported that:

> * The primary tools or techniques used in planning a software development project were workload charts, work break-down structure (WBS), and the subdivision of the software development into phases or tasks.
>
> * About one-fourth of the (planning) time was spent in developing an overall project plan. An equal amount of time was devoted to planning for the (project) organization, planning on how to staff the organization, and developing control procedures.
>
> * (Contrary to his initial assumption) a separate planning group does not normally perform the planning and scheduling functions. The data showed that in 92% of the cases, planning was done by the future manager of the project.
>
> * The predominant estimation method was "estimation based on a similar project" (used in 67% of the projects), followed by "use of a formula" (40%), "expert opinion" (17%), and "crystal ball" (12%). [Note: Some projects combined methods.]

A further analysis of the data suggested that " ... it makes little differnce what type of technique is used in estimating delivery schedule and project cost. None of the used techniques significantly improved the project manager's ability to deliver the project on time and within cost" (Thayer, 1979).

Software estimation historically has been, and continues to be, a major difficulty associated with the management of software development (Devenny, 1976), (Distaso, 1980),

(Mills, 1976), (Pooch and Gehring, 1980), (Yourdon, 1982), (Zelkowitz et al., 1979), (Zmud, 1980). Farquhar (1970), articulated the significance of the issue:

> Unable to estimate accurately, the manager can know with certainty neither what resources to commit to an effort nor, in retrospect, how well these resources were used. The lack of a firm foundation for these two judgements can reduce programming management to a random process in that positive control is next to impossible. This situation often results in the budget overruns and schedule slippages that are all too common today.

A number of reasons for the difficulty have been suggested in the literature:

1. Software development is a process, that is not yet fully understood by "estimators." (Myers, 1972), (Oliver, 1982), (Gehring and Pooch, 1980), (Synnott, 1981), (Pietrasanta, 1968). This often leads to the overlooking of significant cost factors (Myers, 1972), Canning, 1977), (Boehm, 1981).

2. The phases and functions which comprise the software development process are influenced by a large number of ill defined variables (Gehring and Pooch, 1980), (Devenny, 1976), (Aron, 1976), (Distaso, 1980), (Pressman, 1982), (Oliver, 1982).

3. Most of the activities within the process are still primarily human rather than mechanical, and therefore

prone to all the subjective factors which affect human performance (Gehring and Pooch, 1980), (Pressman, 1982), (Oliver, 1982).

4. The lack of a historical data base of cost measurements (Clapp, 1976), (DeMarco, 1982), (Fox, 1976), (Myers, 1972), (Oliver, 1982), (Zelkowitz, 1979).

5. Little penality is often associated with a poor estimate (Zmud, 1980).

Over the years, estimation of project size and development time and cost has been an intuitive process. Experience and the prevailing industry norms have been used as a basis to develop estimates for any given project (Oliver, 1982), (McKeen, 1981), (Auerbach Inc.), (Gehring, 1976). Myers (1972) has identified several "traps" in the experience mehtod (i.e., basing estimates on actual costs of similar past projects), namely:

1. The relationship between cost and system size is not linear. In fact, cost increases approximately exponentially as size increases. Therefore, the experience method should only be applied when the sizes of the current project and past projects are equivalent.

2. Products with similar names are normally very

dissimilar. For instance, chances are slim that two products titled "Payroll System" have the same development costs.

3. Frequent budget manipulations by management in order to avoid overruns makes historical cost data questionable. For example, the movement of cost from an over-budget account to an under-budget account disguises the real costs and makes future use of this data very dangerous.

In the last two decades, several quantitative software estimation models have been developed. They range from highly theoretical ones, such as Putman's model (1978), to empirical ones, such as the Walston and Felix model (1977), and Boehm's COCOMO model (Boehm, 1981). An empirical model uses data from previous projects to evaluate the current project and derives the basic formulae from analysis of the particular data base available. A theoretical model, on the other hand, uses formulae based upon global assumptions, such as the rate at which people solve problems, the number of problems available for solutions at a given point in time, ... etc.

However,

> Even today, almost no model can estimate the true cost of software with any degree of accuracy. (Furthermore,) it is highly unlikely, that any two will produce the same cost estimate for a given project ... The

> variations in cost estimations are influenced by both the many factors involved and the quantization of these factors by the users of the models. Therefore, in order to estimate a software project and develop appropriate manpower guidelines, it is essential to know the factors that influence the software development process at a given facility (Auerbach Inc.).

Finally, we conclude this discussion by Pietrasanta's (1968), frequently quoted, insights into the estimation problem and its solution:

> ... Many of the problems of resource estimating are symptoms of an underlying ignorance of the program system development for which the estimates are being made. The serious student of estimating must first be willing to probe deeply into the fascinating and complex system development process, to uncover the phases and functions of the process, to highlight the subtle interrelationships of the program system being developed and the project organization doing the developing ... examining the influence variables and their causal relationships is precisely what is required if estimates are ever to be improved. Only then can we do meaningful quantitative research and scientific analysis of resource requirements.

II.2.3. Management of the Human Resource:

People and organizational issues have gained recognition, in recent years, as being at the core of effective software development project management (Semprevivo, 1980). For several reasons:

> Personnel costs are skyrocketing relative to hardware costs. Chronic problems in software development and implementation are more frequently traced to personnel shortcomings. Information systems staff sizes have

> mushroomed with little time for adequate selection and training. It is little wonder that Information Systems (IS) managers find themselves focusing increasing amounts of attention on human resource issues (Bartol and Martin, 1982).

In this section we will review the human resource issues of software project management at two levels: (A) Individuals (e.g., selection, motivation, ... etc.); and (B) Groups (e.g., organization, communication, ... etc.).

(A) Individual Dimensions:

On Motivation: One of the major challenges to managers is to motivate employees to high levels of performance. The few studies that have focused on motivational issues among data processing personnel have mainly concerned themselves with rankings of various job factors (Bartol and Martin, 1982). And the findings have been generally supportive of the notion that the work, achievement, and growth are important job factors for data processing personnel (Couger and Zawcki, 1980).

For example, Fitz-enz's (1978) study provides rankings of the job factors considered most important by the 1500 data processing professionals who participated in the study. The items' rankings were as follows: (1) Achievement, (2) Possibility for growth, (3) Work itself, (4) Recognition, (5) Advancement, (6) Supervision, technical, (7) Responsibility,

(8) Interpersonal relations, peers, (9) Interpersonal relations, subordinates, (10) Salary, (11) Personal life, (12) Interpersonal relations, superior, (13) Job security, (14) Status, (15) Company policy and administration, and (16) Working conditions.

A motivation mechanism which is attracting interest in the software engineering field is "goal setting" (Boehm, 1981). An experiment by Weinberg and Schulman (1974) investigated the motivational value of setting clear goals in a programming environment. In the experiment, five teams were given the same programming assignment, but each team was given different directions about what to optimize while doing the job. One team was asked to complete the job with the least possible effort, another team was to minimize the number of statements in the program, another was to minimize the amount of memory required by the program, another was to produce the clearest possible program, and the last team was to produce the clearest possible output. When the programs were completed and evaluated, the researchers found that each team finished first (or, in one case, second) with respect to the objective they were asked to optimize. They also found that none of the teams performed consistently well on all of the objectives.

On Selection: Programmer aptitude tests are available, but their effectiveness is widely questioned (Schneiderman,

1980). Instruments such as the IBM Programmer Aptitude Test (PAT) or the Test on Sequential Instructions (TSI) for measuring programming ability and the Strong Vocational Interest Blank (SVIB) for measuring interest or motivational level have at best produced very weak correlations with analyst capability or programmer capability (Weinberg, 1971) (Boehm, 1981).

On Performance Appraisal: The general literature on performance appraisal suggests that overall, global judgements regarding individual performance constitute inferior means of measuring and appraising performance (Bartol and Martin, 1982). Instead, performance in most jobs consists of a number of different dimensions (e.g., quality versus quantity or efficiency of program execution versus ease of alteration by another programmer).

Gilb (1977) has suggested a number of possible metrics of performance. Jones (1978) has pointed to the difficulties in using certain standard measures, such as lines of code per programmer-month, and has suggested other approaches, such as separating quality measurements into measures of defect removal efficiency and defect prevention.

On Turnover: Turnover continues to be a chronic problem for software project managers. Willoughby (1977) estimates that annual turnover in the DP field ranged between 15 and

20% during the 1960s, declined to about 5% in the early 1970s, and began to rise again by the end of the decade. More recent studies place the annual turnover rate at 25.1 % (Tanniru et al, 1981), 30 % (Richmond, 1982), and even as high as 34 % (Bott, 1982). As McLaughlin (1979) points out, at such rates the equivalent of a work unit turns over every three to four years --- no minor matter in a profession where it frequently takes 12 to 18 months before a new employee makes significant work contributions.

There are few predictive studies of DP turnover. In one such study, Bartol (1979) investigated the relative importance of two individual factors, personality and professional attitude, versus two organizational factors, professional reward system and tenure, in predicting turnover among computer professionals. Only the professional reward system and tenure variable were found to be significantly predictive for the turnover variable, both in the expected negative direction.

(B) Group Dimensions:

There are two basic issues involving the use of groups in software development. One relates to structural factors (i.e., how the groups are formulated), and the second involves process factors relevant to the ongoing operations and interrelationships of group members.

On Structure Factors: Software development projects are structured in one of three basic organizational forms: (1) Functional form; (2) Matrix form; or (3) Project form (Daly, 1982) (Thayer, 1979). Youker (Y2) suggests that these three organizational forms may be represented as a continuum ranging from functional on one end to project on the other end, with matrix falling in between and including a wide variety of structures from weak matrix near functional to a strong matrix near project. Several authors have presented proposed guidelines or checklists for choosing the "appropriate" organizational form. (e.g., see Green (1982), Youker (Y2), and Daly (1982)).

In a survey of 60 software development projects in the aerospace industry, Thayer (1979) found that "the matrix organization is predominant, with 58% of the projects using this type of organization, 38% of the projects using a project organization, and 4% using a functional organization." He also found that very small projects were split between project and matrix organizations, medium priced projects (between 1 and 5 million dollar) were slightly biased in favor of project organization, while expensive projects (5 million to 50 million) are almost always matrix organization. A comparison of organizational form to "on time" and "within budget" delivery of the software showed that "it made little difference as to what kind of project (organization) type is used."

Thayer's data also showed that the team concept is much in use. About 95% of the projects were handled by teams under the direction of technical leaders of some sort.

Two philosophies for organizing programming teams have achieved a moderate amount of popularity in the data processing field. These are the egoless programming team proposed by Weinberg (1971), and the chief programmer team proposed by Mills (1971) and implemented by Baker (1972).

Little experimental work on programming team and task interaction has been carried out (Mantei, 1981). Weinberg's suggestions are anecdotal and Baker's conclusions are confounded by the team personnel and the programming methods selected.

On Process Factors: The attention here has focused on the communication processes between members of a programming team. In what is probably the most cited reference on the topic, Brooks (1978) suggests that human communication in a software development project is a significant overhead. And that the overhead is made up of two parts, training and intercommunication. Each worker must be trained in the technology, the goals of the effort, the overall strategy, and the plan of work. This training cannot be partitioned, so this part of the added effort varies lineary with the number of workers. Intercommunication, Brooks further

suggests, is worse. It increases as n(n-1)/2 (where n is the number of team members).

The implications of this, is that increasing the size of a software team increases the amount of software produced per unit time, only to a point. Then the problems of communication among the programmers begin to dominate the project and reduce the amount of software being produced (Boebert, 1979). Or in Brooks' words (1978), "Oversimplifying outrageously, we state Brooks' Law: Adding manpower to a late software project makes it later."

The relationship between human communication and programmer productivity was investigated by Scott and Simmons. First, while using the Delphi survey technique to identify project variables that influence programmer productivity, they found that "effect of project communication" to be one of the "eight consensus variables which have an important influence on productivity" (Scott and Simmons, 1974). And in a later study (1975), they used computer simulation to evaluate the communication overhead as a function of a team's communication structure.

Finally, taking a different tack, Parnas (1971) considered the impact of human communication on the product of software development. He suggests that too much communication between the members of a programming team could

negatively affect modularity, because team members would tend to use informal information to bypass structured interfaces.

II.2.4. Control:

Once a plan becomes operational, control is necessary to measure progress, to uncover deviations from plan, and to indicate corrective action (Koontz and O'Donnel, 1972). While in most production environments, control is a standard business practice (Mills, 1983), in the production of software control is a "perilous activity" (Arseven, 1975), (Boehm, 1976), (Fox, 1976), (Gehring, 1977), (Gansler, 1976), (Gehring, 1976), (Lehman, 1979), (Metzer, 1981), (Miller, 1955), (Pooch and Gehring, 1980), (Thayer, 1979). Paraphrasing Mills (1983):

> It is difficult to measure performance in programming. It is difficult to diagnose trouble in time to prevent it. It is difficult to evaluate the status of intermediate work such as undebugged programs or design specification and their potential value to the complete project.

Such a state of affairs has stirred, not only self-criticism within the profession [(Lehman, 1979), (DeRose and Nyman, 1979), (Metzger, 1981), and (Jensen and Tonies, 1979)] but open criticism from the user community as well:

> You software guys are too much like the weavers in the story about the Emperor and his new clothes. When I go out to check on a software development the answers I get

sound like, 'we're fantastically busy weaving this magic cloth. Just wait a while and it'll look terrific.' But there's nothing I can relate to, no way to pick up signals that things aren't really all that great. And there are too many people I know who have come out at the end wearing a bunch of expensive rags or nothing at all.
(A U.S. Government Spokesman quoted in (Gehring and Pooch, 1980).)

The manifestation of poor software project control has more than one form. For example:

1. The "90% Syndrome," (Baker, 1982), (Boehm, 1981), (DeMarce, 1982), (Donelson, 1976).

2. The production of inadequate software e.g., that doesn't meet user requirements (Tansworthe, 1977), (Glass, 1982).

3. Building systems that are inordinately expensive (McKeen, 1981) (Wolverton, 1974) e.g., due to unconstrainted goldplating (Wolverton, 1974), (Boehm, 1981), (Kirby, 1982), (Radice, 1982).

4. Lack of historical software cost data bases (Boehm, 1981) (Thayer, 1979).

Why is it difficult to control software development projects? Two classes of factors have been proposed in

the literature: (1) product-type; and (2) people-type factors.

Product-Type Factors

1. Software is basically an intangible product during most of the development process, and for which there are no visible milestones to measure progress and quality like a physical product would (Wegner, 1980), (Corbato, 1979), (Miller, 1955), (Jones and Mclean, 1970), (Boebert, 1979), (Wolverton, 1974), (Reynolds, 1970), (Gehring, 1976), (Boebert, 1979), (Hales, 1982a). "This invisibility is compounded for large software, for which logical complexity cannot be maintained in one person's mind, and for which development must be partitioned into a number of tasks assigned to different people" (Zmud, 1980).

2. High complexity (McKeen, 1981), (Corbato and Clingen, 1979). "In an overly ambitious project, managers who do not understand the details of what they are managing are easily blustered and misled by subordinates. Conversely, low-level staff may be unable to appreciate the significance of details and fail to report serious problems" (Corbato and Clingen, 1979).

3. Volatility of requirements (Distaso, 1980), (Metzger, 1981), (Tsichritzis, 1977), (Toellner, 1977), (Zmud, 1980). "Since software system modules are not visibly connected, in contrast to hardware systems, the impact of a change is often not readily apparent even to the designers of the system" (Gehring, 1976).

People-Type Factors:

1. The "software wizard syndrome" (Boebert, 1979). This occurs when management abdicates its responsibility to some highly trusted software specialist, whose pronouncements are viewed as correct by definition. The trouble with the syndrome is that software wizards, unlike the mythical kind, are both fallible and mortal.

2. Inaccurate reporting (Boebert, 1979), (Jones, 1979), (Gehring, 1977). In software development, "The employee has control of the resource, his time, and he accounts for the resource on his time sheet. The employee knows that his time sheet is a performance evaluation factor and is a written record. He knows the estimated time for the project serves as a recorded budget. This combination of written records makes a pressure device and 'adjusted amounts' often result" (Reed,

1979), e.g., to hide problems or embarassing situations (Jones, 1979). Another explanation was given by Boebert (1979): "Programmers are paid to program, not to pay attention to progress ... Management should not expect to get progress or status information by asking programmers, the typical programmer doesn't know or care, and will usually give whatever answer is needed to end the meeting and get back to programming."

3. Optimism, (Corbato, 1979), (Oliver, 1982), (Jones and McLean, 1970), (Snyder, 1976), (McKeen, 1981), (Gunther, 1978). "All programmers are optimists," Brooks (1978) remarked, always unjustifiably assuming that "'This time it will surely run' or 'I just found the last bug'" (Brooks, 1978).

The persistence of the industry's difficulties in controlling software development does not seem to be the result of either a scarcity of "advice" from the research community, or a reluctance, on the industry's part to "heed" that advice.

Numerous techniques, often adapted from other industries, have been proposed in the literature. These include: Work Break Structure (WBS) (Tausworth, 1980),

PERT (Boehm, 1981), Gantt Charts (Knutson, 1980), Formal Reviews (Freedman and Weinberg, 1982), Unit Development Folder (UDF) (Ingrassia,1979), and Automated project Management Systems (Canning, 1976).

Furthermore, evidence indicates that most of these "proposed solutions" have been disseminated into the industry (Glass, 1982), albeit at varying degrees. For example, Thayer's survey of software projects in the aerospace industry showed the following:

Technique	% of Projects Using it
Formal Reviews	97 %
WBS	60 %
Automated Project Management System	57 %
PERT	38 %
Gantt	32 %

Thayer further investigated whether the utilization of the above "state-of-the-art" techniques was effective in resolving the control-type difficulties in those aerospace firms surveyed. (Note: Thayer (1979), as well as others (e.g., Lehman (1979), believe that the aerospace industry is the most advanced and experienced in employing software project management techniques.) His results indicated that they, in fact, did not.

Results reported by Lehman (1979), on a survey of

software development projects also in the aerospace industry, were more surprising:

> 17% of the projects had no project control mechanism. And more surprisingly yet, that group fared better than average relative to on-time delivery ...

A similar finding was reported by (Powers and Dickson, 1973). In a study of 20 MIS-type projects they found that:

> With respect to the project control techniques used for the projects in the study, they tended to be dysfunctional to project success. The use of project control methods was not significantly related to any criterion of success, and, indeed, had a negative relationship to the reported quality of project documentation ...
> In general, project leaders appeared to feel an implicit pressure from tight project reporting requirements, to which they responded by cutting corners on documentation and preparations for implementation.

So, what is the prognosis on the status of software project control? Bauer (1980) put it this way:

> We are able to identify the sources of our troubles, but in many cases we have nothing to offer but good advice. We are in the situation of a physician who keeps trying out different pills on his patient in the hope that some will finally cure him (Bauer, 1980).

III. MODEL DEVELOPMENT

III.1. Introduction:

As stated in Chapter I, the objective of this research effort is to develop and test an integrative system dynamics model of software development project management which would provide us with understanding and insight about the general process by which software development is managed.

A system dynamics model of software development can increase our understanding of the process through both the formulation of the model's structure and the analysis of its behavior. Experimentation and analysis of model behavior will be the focus of the next two chapters. In this chapter, on the other hand, our objective is to enhance our understanding of the software development process through model formulation.

Model formulation can enhance understanding in several ways (Schultz and Sullivan, 1972):

1. Confrontation --- vague generalizations crumble when put to the test of modeling.
2. Expansion --- the tendency to a holistic integrative approach in modeling forces a broadening of one's horizon, a looking into other relevant fields for ideas.
3. Communication --- problem-oriented models lead to jumping of disciplinary boundaries, less parochialism.
4. Organization --- organizing data and structuring experience.

In addition, the formulation of the model forces explication i.e., structural relations between variables must be explicitly and precisely defined. This, in Dubins (1971) view, is the "locus of understanding" of a theoretical model:

> A (theoretical model) tries to make sense out of the observable world by ordering the relationships among "things" that constitute the (modeler's) focus of attention in the world 'out there' ... What is gained in understanding ... is achieved by comprehending the law or laws built into the model. The locus of understanding in a scientific model is to be found in its laws of interaction. (That is, the modes of interaction among the variables of the model).

Before relationships are defined, however, one has first to choose the "things" or variables whose relationships are of interest. That is, one has to define the model's boundary. Models have a boundary within which they are expected to "mirror" the empirical world. Beyond that

boundary it may be problematic as to whether the model holds. Our model's boundary is discussed in Section III.3. below. This is then followed by a detailed description of the model's structure and equation formulation in Section III.4.

In the section immediately following this, we discuss the sources of information, on software development project management, we used to construct the model.

III.2. Sources of Information:

To build the model, we went through three information gathering steps:

First, we conducted a series of ten 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 development organizations.

> The system dynamics approach starts with the concepts and information on which people are already acting (Forrester, 1979).
>
> In general sufficient information exists in the descriptive knowledge possessed by the active practitioners --- to serve the model builder in all his initial efforts (Forrester, 1961).

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.

> A model should come first. And one of the first uses of the model should be to determine what formal data need to be collected (Forrester, 1961).

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, 1981C).

The setting for this was a series of 17 interviews conducted between October 7,1982 and July 7,1983 with software project managers at Digital Equipment Corporation, MIT, and General Motors.

In the remaining part of this section, we explain the

above three information gathering steps in more detail.

Step (1):

As stated above, this step constituted a "formulative study." The objective was to increase our familiarity with the software development process, in particular, "the concepts and information on which software project managers are already acting," in order to formulate an initial skeleton system dynamics model of the process.

The technique we used was the "focused interview." In the focused interview, as described by Selltiz, Wrightsman, and Cook (1976),

> ... the main function of the interviewer is to focus attention upon a given (list of topics). Interviewers know in advance what topics, or what aspects of a question, they wish to cover. This list of topics or aspects is derived from a formulation of the research problem ... This list constitutes a framework of topics to be covered, but the manner in which questions are asked and their timing are left largely to the interviewer's discretion.

This type of interview, according to Green and Tull (1978), " ... is useful in obtaining a clear understanding of the problem and determining what areas should be investigated (further)."

Before each interview, two things were done. First the interviewee was briefed, in a telephone conversation, about the objectives of the research. The interviewee was also told that the primary objective of the interview is to find out how software projects are managed in his/her organization. The list of topics shown in Exhibit III.1. was read to each interviewee. The second thing we did, was to mail each interviewee a copy of our internal report titled "The System Dynamics Approach to Designing Software Project Planning & Control Systems: A Research Proposal." The report, written in January 1982, constitued the first "rough" version of our research proposal, and it provided, in addition, a non-technical introduction to the system dynamics methodology.

Ten interviews were conducted in the period between February 5,1982 and April 30,1982. Each interview was, on the average, two hours long. The names of the interviewees, their organizations, their titles, and the dates of the interviews are shown in Exhibit III.2.

All ten interviewees were reached through contacts, primarily those of Sloan faculty members. Each one of the interviewees was currently managing one or more software development projects, had been a software project manager/leader for at least two years, and had managed at least two completed software projects. This, we felt, would

Environment:	o Project types, sizes
	o Hardware environment
	o Organizational structure
Software Production:	o Software tools
	o Standards
	o Error rates
	o QA policy
Planning:	o Estimating
	o Effort Distribution
Control:	o Control tools
	o Milestones
	o Reporting frequency
Human Resources:	o Hiring/firing policies
	o Training
	o Turnover
	o Overtime policy

EXHIBIT III.I

Interview #	Date	Interviewee	Title	Organization
1	2/5/82	John James	Group Leader	MITRE
2	2/10/82	William Stein	Member of Technical Staff	MITRE
3	3/5/82	Clement McGowan	Principal Consultant	MITRE
4	3/15/82	Glen Gage	Project Manager	DEC
5	3/15/82	Joanne Riccardi	Project Leader	DEC
6	3/22/82	Dave Griffin	Project Leader	DEC
7	3/22/82	Jim Doyle	Project Manager	DEC
8	3/29/82	Bonnie Donahue	Project leader	DEC
9	4/7/82	Wayne Babich	Lead Designer & Technical Mgr. Softech Federal System Div.	Softech
10	4/30/82	Francis O'Conner	Group Leader	MITRE

EXHIBIT III.2

provide a level of managerial experience and maturity that would be adequate for gaining insights into the management of software projects.

As is shown in Exhibit III.2., three organizations were represented, namely, Digital Equipment Corporation (5 interviewees), MITRE (3 interviewees), and SofTech (2 interviewees). This provided us with an exposure to three quite different software development environments. In DEC, all five interviewees were involved in developing software for in-house use (e.g., order administration systems). In MITRE, the projects involved the development of embedded-software for the Air Force. And in SofTech, the projects involved a wide range of systems developed on contract for client organizations, both private and public.

The outcome of the above exercise was, as mentioned above, the formulation of an initial simple system dynamics model of software project management. The model is discussed in detail elsewhere (Abdel-Hamid and Madnick, 1982b). This initial model, in addition to serving as a road-map for the succeeding literature reviewing step, was also the "skeleton" for developing our final more detailed version. Which, therefore, means that the information gathered here is also incorporated in the formulation of our final model. This will become more evident when we discuss that model's structure and equation formulation in Section III.4. In

those discussions we will, in many occasions, make reference to the interviews of Exhibit III.2. Such references will always be in the form: (interviewee-name, interview number).

Step (2):

Starting the extensive review of the literature with the initial model serving as the road-map had several important advantages. It was helpful, for example, in organizing the findings, as well as in integrating them. In addition, the integrative nature of our model "prompted" us to broaden our horizon, and look into other relevant fields for ideas. Examples of these "ventures" include: Management Control (e.g., Anthony (1979), and Lawler (1976)), Cybernetics (e.g., Ashton (1976)), Organizations (e.g., Kotter (1978), Schein (1980), and Weick, (1979)), Project Management (e.g, Maciariello (1978)), and Psychology (e.g., (Ingham el al, 1974), Leavitt (1978), and Steiner (1966)).

In discussing the final model's structure and its equation formulation in Section III.4., we will make extensive use of the massive amount of information gathered in this literature review. And, it will then become evident, how effective such a model truly is in organizing and integrating the various bodies of knowledge mentioned above.

Step (3):

> The written record has (a major shortcoming) compared to the mental data from which the written data were taken ... the written record usually cannot be queried. Unlike the mental data base, the written record is not responsive to probing by the analyst as he searches for a fit between structure, policy, and behavior (Forrester, 1979b).

That was one reason to conduct the second set of interviews, which constitued our third information gathering step. That is, there were still unanswered questions that had to be addressed.

The second reason, was to expose the more detailed model that emanated at the end of Step (2) above, to, in Roberts' (1981c) words, "criticism, revise it, expose it again and so on in an iterative process that would continue as long as it proves to be useful."

As a result of these two objectives, the model's structural components became a core around which the interviews were built. The interviews were, thus, more structured in terms of content than those in Step (1). However, the interviews were unstructured in the sense that no standardized set of questions was used. Such a format, according to Isaac and Michael (1971), allows the interviewer to adjust the interview so as to take advantage of an interviewee's personal areas of expertise.

As in Step (1), before each interview, interviewees were contacted by telephone and briefed on the objectives of the research. The topics covered were basically the same as those in the Step (1) telephone briefings, except for an additional brief discussion of the Systems Dynamics methodology. Interviewees were then mailed copies of: (1) "A Model of Software Project Management Dynamics" (Abdel-Hamid and Madnick, 1982b); and (2) "System Dynamics --- An Introduction" (Roberts, 1981c).

It was necessary that this group of interviewees have some understanding of the Systems Dynamics methodology, since one of our objectives was to expose the model to their critique. This was not a major hurdle, however. What was needed was basically an understanding of the feedback concept, and its representation in terms of causal loop diagrams. Both of which are adequately covered in the Roberts' introductory paper. In the interviews, references were made only to "pieces" of the model, and these were always in the form of causal loop diagrams. An example "conversation piece," on the effects of "schedule pressure" on "productivity" and "error generation," is shown in Figure III.1.

Seventeen interviews were conducted in the period between October 7,1982 and July 7,1983. The names of the interviewees, their organizations, their titles, and the

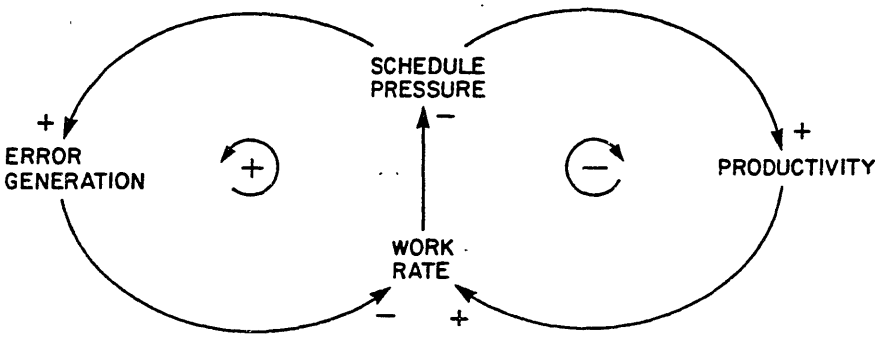

SCHEDULE
PRESSURE
+
ERROR
GENERATION
+
PRODUCTIVITY
WORK
RATE
−
+

Figure III.1

dates of the interviews are shown in Exhibit III.3. below.

A comparison of Exhibits III.2. and III.3. would show that none of the interviewees of Step (1) were among those interviewed later in Step (3). This, we feel, had two positive results. Firstly, it provided us with a larger and more varied pool of experiences and ideas to draw upon, and secondly, it decreased the possibilities for bias in the interviewees' critique of the model.

Except for Mr. Sheldon of MIT who was reached through the personal contacts of an MIT faculty member, this group of interviewees was reached through MIT's Center for Information Systems Research (CISR). Both GM and Digital are CISR sponsors, and occasionally serve as field sites for research in the MIS area. Again, each of the interviewees was "currently managing one or more software development projects," "had been a software project manager/leader for at least two years," and "had managed at least two completed software projects."

Because the discussions at this stage were at a more detailed level than those of Step (1), we needed more time per interviewee. On average, we conducted three two-and-half-hour-long interviews per interviewee.

This battery of seventeen interviews constituted the

Interview #	Date	Interviewee	Title	Organization
11	10/7/82	Mike Landolfi	Mrg. of Planning, Analysis & Control,Finacial&Admin Infosystem	DEC
12	11/3/82	Garrett Sheldon	Mrg. of Business Systems Development	MIT
13	11/4/82	Mike Landolfi	(see above)	DEC
14	11/11/82	Al Chan	Proj. Laeder,NAVO Fin.Syst.	GM
15	11/11/82	Sam Hisamune	Sr. Supervr. Syst. Devt.,CANISCA	Gm
16	11/16/82	Frank Lombardi	Mrg. Revenue Disbursement Syst.	DEC
17	11/23/82	Frank Lombardi	(see above)	DEC
18	11/23/82	Barbara Nichols	Syst. Mrg. Export Services Group	DEC
19	11/24/82	Garrett Sheldon	(see above)	MIT
20	12/15/82	Al Chan	(see above)	GM
21	12/15/82	Sam Hisamune	(see above)	GM
22	1/17/83	Mike Landolfi	(see above)	DEC
23	1/17/83	Frank Lombardi	(see above)	DEC
24	2/16/83	Garrett Sheldon	(see above)	MIT
25	2/18/83	Barbara Nichols	(see above)	DEC
26	6/29/83	Sam Hisamune	(see above)	GM
27	7/7/83	Barbara Nichols	(see above)	DEC

EXHIBIT III.3

third and final information gathering step. And it lead to the formulation of the final model ... which we discuss in the next two sections.

III.3. Model Boundary:

> (Models are) analogues of existing or conceivable systems, resembling their referent systems in form but not necessarily in content. In some way they exhibit, display, or demonstrate structural relationships among elements found in the referent system. At the same time, they are abstractions and idealizations, omitting some aspects of the referent systems and duplicating only those that are of interest for the purposes at hand (Schultz and Sullivan, 1972).

A clear understanding of the purpose of a modeling effort helps to answer questions relating to the system boundary --- i.e., what should be included and what should be excluded.

As was stated in Chapter I, the primary purpose of our model is to "provide us with understanding and insight about the process by which software systems are developed and managed."

Notice that our focus is confined to the development phases of software production. Our model's boundary will thus extend only until the last phase of software development, namely, the testing phase. Not included in our

model are, therefore, the subsequent maintenance activities.

It was also indicated that the model would integrate the managerial functions of planning, controlling, and human-resource management as well as the software production activities of design, coding, and testing. Notice that the model's boundary extends from the beginning of the design phase of the software life cycle, excluding the requirements definition phase. There were two reasons for this. First,

> Analysis to determine requirements is ... distinguished as an activity apart from software development. Technically, the product of analysis is non-procedural (i.e., the focus is functional) while the prime development is the basis for mutual agreement between the customer and the developer as to what the system must accomplish (McGowan and McHenry, 1979).

Secondly, our focus in this study is on the software development organization, i.e., project managers and software development professionals, and how their policies, decisions, actions, ... etc., affect the success/failure of software development. The definition of user requirements is therefore excluded from the model's boundary for the additional reason that it lies beyond the control of the software development group.

Such arguments have also been the bases for excluding the software requirements phase from the "boundaries" of quantitative-type software cost estimation models such as

COCOMO (Boehm, 1981).

Together with excluding the requirements definition phase, we will make the simplifying assumption that once requirements are fully specified (outside the boundary of the model), and the architectural design phase is initiated (within the model's boundary), there will be no significant subsequent changes in the users' requirements. We do realize that changes in users' requirements are frequently blamed for cost/budget overruns in software projects (Aron, 1976) (Boehm, 1980) (Zolnowski and Ting, 1982), and for which the users are often "charged" and found "guilty" (Distaso, 1980), (Thayer, 1979), (Toellner, 1977). However, let us reiterate that our focus in this study is on "the software development group members and <u>their</u> policies, decisions, actions, ... etc." And we suggest that investigating those policies, decisions, and actions which can cause cost/budget overruns <u>inspite</u> of stable user requirements is a more interesting and challenging research endeavor than to answer the question "do changes in users' requirements negatively impact the development process."

Looking within a model's boundary (e.g., at the actions of the software development team) for the causes/cures of problematic behavior rather than outside it (e.g., the actions of the users) is a characteristic of the system dynamics approach. Richardson and Pugh (1981), called it the

"Endogenous Point of View," and elaborated on it as follows:

> ... the system dynamics approach tends to look <u>within</u> a system for the sources of its problem behavior. Problems are not seen as being caused by external agents outside the system ...
> The internal view creates a dramatically different problem focus. The external view places an individual, a firm, a city, or whatever, at the mercy of exogenous events ... The external view is frequently predisposed to search for blame: "instabilities in our workforce and inventory are caused by errotic and seasonal customer orders" (or software projects overrun schedules merely because of changes in user requirements) ...
> The internal view searches (instead) for structures within (the system), which can create or exacerbate the system's problem behavior.

As we mentioned above, our model's focus is on the decisions and actions of the software development group including both project management as well as software development professionals (e.g., designers and programmers). In addition to excluding users (as indicated above), it, therefore, also excludes computer center operators, personnel department personnel, secretaries, higher management, janitors, and so on.

Finally, this model is not a model of small one-programmer-type projects, nor of super-large projects involving hundreds of software professionals over a period of several years. Instead, our domain is that of medium sized projects. Jones (1977) defined "medium-sized" software projects as follows:

> ... (they) range between 16K and 64K lines in size, (and in which) development teams or departments are the norm ... Below the "medium" size range, programming as a business endeavor is often successful: at least the programs tend to work fairly well and insurmountable problems are not often ecountered. At the "medium" size and above, cost and schedule overruns pop up more frequently, and are more serious when they do occur.

III.4. Model Structure:

This section describes the structure of our integrative system dynamics model of software development project management. An overview of the model is first presented, highlighting the four major subsystems of the model, namely, human resource management, planning, controlling, and software production, together with the various flows which connect them. Next, each of the four subsystems, will be described in more detail, in terms of its basic components and relationships. The various assumptions and propositions comprising the model are supported by reference to the literature and to the interviews of section III.2. The outline of the presentation will be as follows:

III.4.1. Model Overview

III.4.2. System Dynamics Schematic Conventions

III.4.3. Human Resource Management

III.4.4. Software Production

III.4.5. Controlling

III.4.6. Planning

A documented listing of each subsystem's DYNAMO equations is included in Appendix (A). DYNAMO is the computer simulation language used. It is a language specifically designed to handle non-linear feedback models of the sort associated with the system dynamics method. (For an introduction to DYNAMO see (Pugh, 1976).)

III.4.1. Model Overview:

Figure III.2. is an overview 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 the interrelatedness of 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, they, as Figure III.2. suggests, both affect and are affected by the other subsystems. For example, the project's "hiring rate" is a function of the "workforce needed" to complete the project on a planned completion date.

Similarly, the "workforce available," has direct bearing on the allocation of manpower among the different software production activities in the Software Production Subsystem.

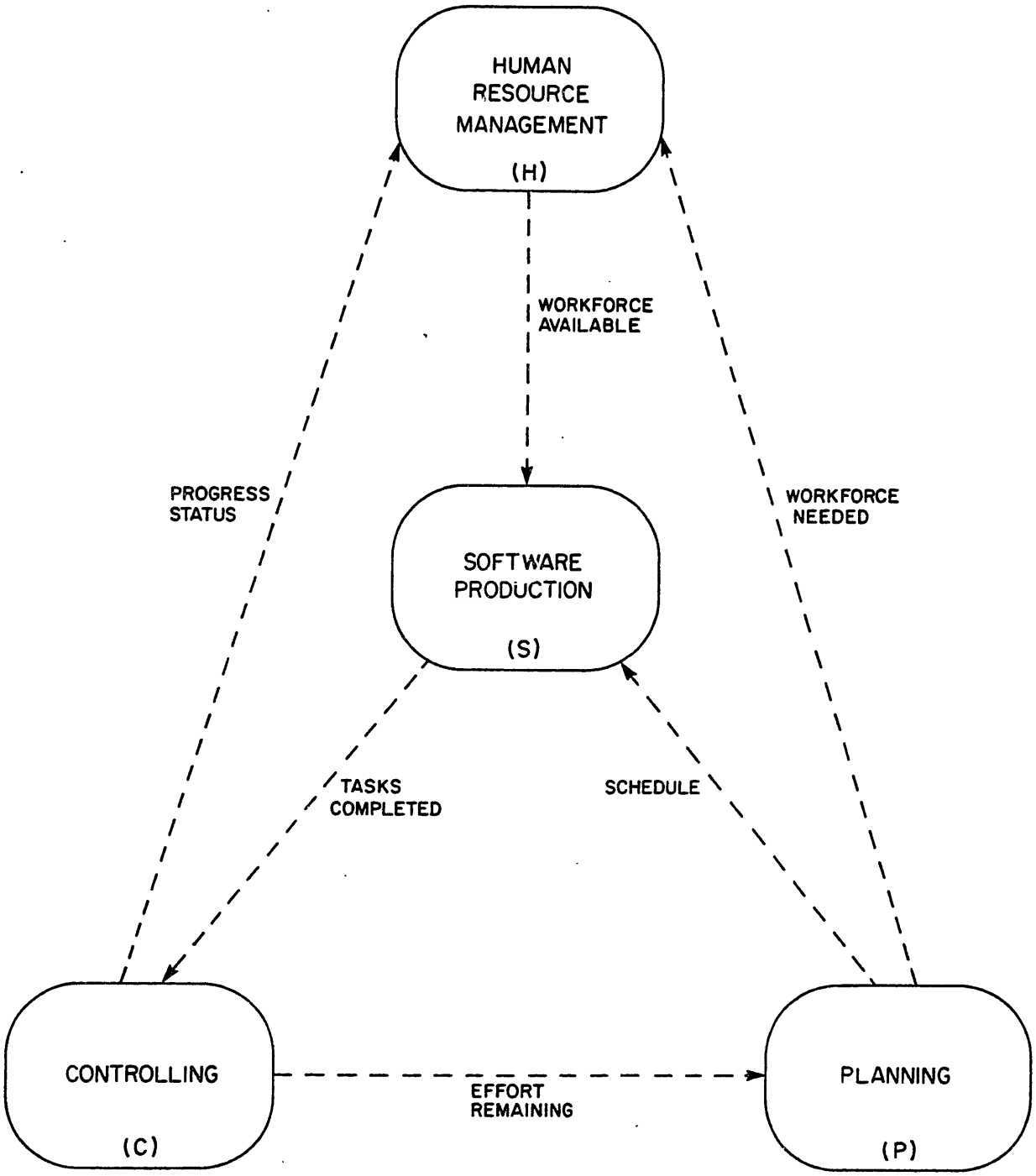

HUMAN
RESOURCE
MANAGEMENT
(H)
WORKFORCE
AVAILABLE
PROGRESS
STATUS
WORKFORCE
NEEDED
SOFTWARE
PRODUCTION
(S)
TASKS
COMPLETED
SCHEDULE
CONTROLLING
(C)
EFFORT
REMAINING
PLANNING
(P)

Figure III.2

The four primary software production activities 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 design/coding errors. Errors detected through such quality assurance activities are then reworked. Not all errors will be detected and reworked, however, some will "escape" detection until beyond the end of development e.g., until the testing phase.

As progress is made, 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 Controlling Subsystem. As was indicated in Chapter II, determining where a software project <u>really</u> is e.g., in terms of % of tasks completed, is not always possible. (E.G., because software is basically an intangible product during most of the development process, and for which there are no visible milestones to measure progress and quality like a physical product would.) 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 to start the project, and then those 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.

With this overview of the model's subsystems, and their interrelationships, we are almost ready to proceed to a more detailed description of each of the four subsystems. Because all the subsystem diagrams will be in terms of the schematic conventions used in system dynamics, we feel it would be useful to preface the discussion of the model's subsystems with an introduction to these conventions.

III.4.2. System Dynamics Schematic Conventions:

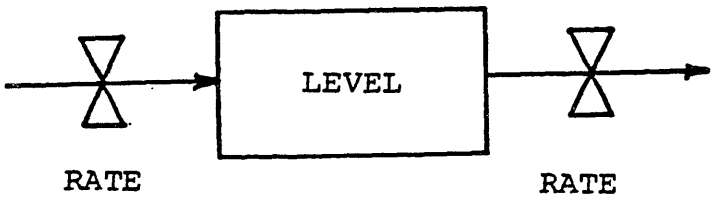
From a System Dynamics perspective all systems can be represented in terms of "level" and "rate" variables, with "auxiliary" variables used for added clarity and simplicity.

A level is an accumulation, or an integration, over time of flows or changes that come into and go out of the level. The term "level" is intended to invoke the image of the level of a liquid accumulating in a container. The system dynamicist takes the simplifying view that feedback systems involve continuous, fluid-like processes, and the terminology reinforces that interpretation.

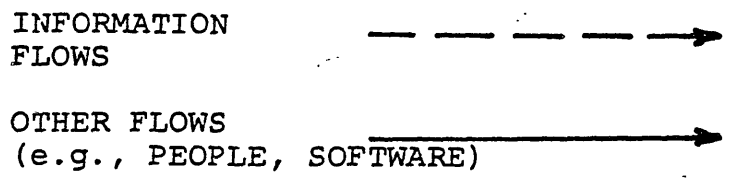
The flows increasing and decreasing a level are called

rates. Thus, a manpower pool would be a level of people that is increased by the hiring rate and decreased by the firing and/or quit rate.

Rates and levels are represented as stylized valves and tubs, as shown below, further emphasizing the analogy between accumulation processes and the flow of a liquid.



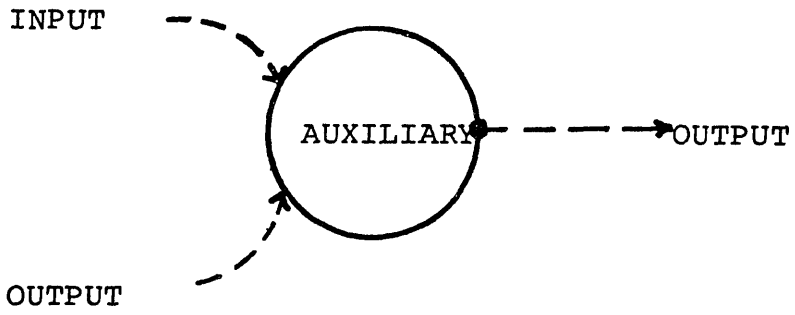
The flows that are controlled by the rates are usually diagramed differently, depending on the type of quantity involved. We will use the two types of arrow designators shown below:



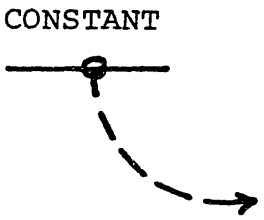
Flows will always, of course, originate somewhere and terminate somewhere. Sometimes, the origin of a flow is treated as essentially limitless, or at least outside the model-builder's concern. In such a case the flow's origin is called a source. Similarly, when the destination of a flow is not of interest, it is called a sink. Both sources and sinks are shown as little "clouds."

All tangible variables are either levels or rates i.e., they are either accumulations of previous flows or are presently flowing. But there is one more type of information variable, which is called an auxiliary. Auxiliary variables are combinations of information inputs into concepts e.g., "desired workforce," or policies e.g., "training policy." Auxiliaries are represented by a circular symbol.

A few other symbols will complete the designation of items included in formal system dynamics diagrams. In addition to the variable symbols shown above, models also



include constant terms, i.e., parameters of the model whose values are assumed to be unchanging throughout a particular computer simulation. Constants are pictured as is shown below, the name of the constant being underlined, with an information arrow going to the variable that is affected by the constant.



Finally, because complex models are often diagramed in multiple displays, situations arise in which variables pictured on one diagram are used in another diagram. These variable cross-references are shown by including the name of the other diagram's variable in parentheses as shown below.

III.4.3. Human Resource Management:

The Human Resource Management Subsystem is depicted in Figure III.3. As the figure indicates, a project's total workforce is comprised of two workforce levels, namely, "Newly Hired Workforce" and "Experienced Workforce." Disaggregating the workforce into these two categories of employees was done for two reasons.

First, newly hired project members pass through an "orientation phase" during which they are less than fully productive (Canning, 1977), (Cougar and Zawacki, 1980), (Weil, 1981), (Wolverton, 1974), (Chrysler, 1978), (Tanniru et al, 1981), (James, 1), (Lombardi, 16), and (Hisamune, 26). (Remember, a reference citation in the form (name, i) where "i" is a number between 1 and 27, refers to one of the 27 interviews of Exhibits III.2. and III.3.) The orientation process has both technical as well as social dimensions. On the technical side,

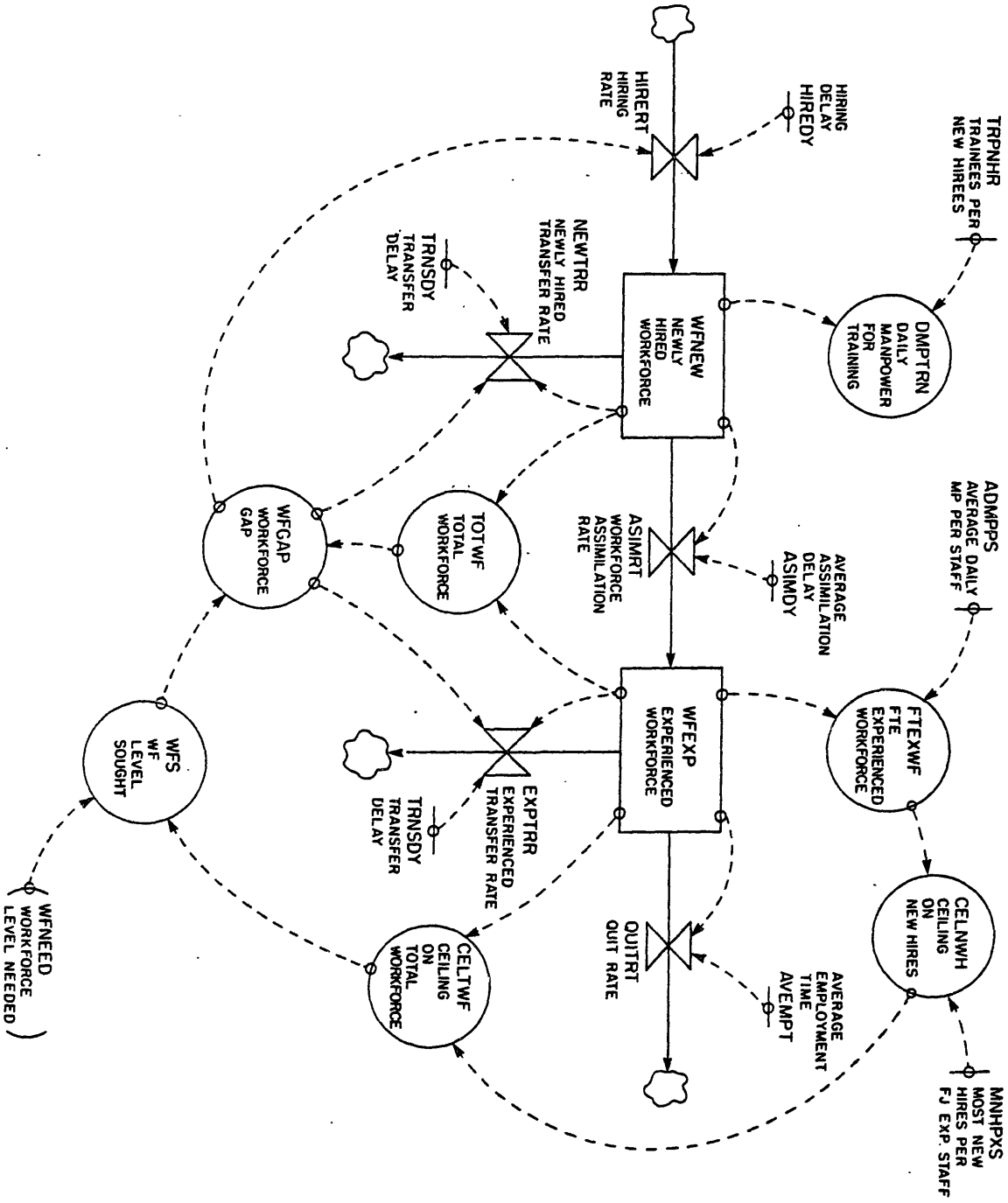

TRPNHR
TRAINEES PER
NEW HIREES
HIRERT
HIRING
RATE
HIRING
DELAY
HIREDY
DMPTRN
DAILY
MANPOWER
FOR
TRAINING
WFNEW
NEWLY
HIRED
WORKFORCE
NEWTRR
NEWLY HIRED
TRANSFER RATE
TRNSDY
TRANSFER
DELAY
ADMPPS
AVERAGE DAILY
MP PER STAFF
ASIMRT
WORKFORCE
ASSIMILATION
RATE
AVERAGE
ASSIMILATION
DELAY
ASIMDY
TOTWF
TOTAL
WORKFORCE
WFGAP
WORKFORCE
GAP
WFEXP
EXPERIENCED
WORKFORCE
FTEXWF
FTE
EXPERIENCED
WORKFORCE
WFS
WF
LEVEL
SOUGHT
EXPTRR
EXPERIENCED
TRANSFER RATE
TRNSDY
TRANSFER
DELAY
CELNWH
CEILING
ON
NEW HIRES
QUITRT
QUIT RATE
AVERAGE
EMPLOYMENT
TIME
AVEMPT
CELTWF
CEILING
ON
TOTAL
WORKFORCE
MNHPXS
MOST NEW
HIRES PER
FJ EXP. STAFF
WFNEED
WORKFORCE
LEVEL NEEDED

Figure III.3

> ... (newly hired) personnel often require considerable training to become familiar with an organization's unique mix of hardware, software packages, programming techniques, project methodologies and so on (Winrow, 1982).

And paraphrasing Schein (1980) on "social orientation:"

> ... (it) refers to the processes of teaching the new recruit how to get along in the organization, what the key norms and rules of conduct are, and how to behave with respect to others in the organization. The new recruit must learn where to be at specified times, what to wear, what to call the boss, whom to consult if he or she has a question, how carefully to do a job, and endless other things which insiders have learned over time.

Of course, not all new project members are necessarily recruited from outside the organization, some might be "recruited" from within e.g., transferred from other projects. For this type of employee, there will still be a "project orientation" period (Brooks, 1978) e.g., to learn the project's ground rules, the goals of the effort, the plan of work, and all the details of the system (GRC, 1977), (Thayer and Lehman, 1977). Although obviously less costly than the "full orientation" needed by an out-of-company recruit, project orientation can still be a significant drag on productivity, especially when a project lacks adequate documentation (Canning, 1977). In a GRC (1977) report it was noted that when workforce additions are made to "rescue" a project e.g., that is behind schedule, it is often the case

that such a project also suffers from sparse and outdated documentation.

The important point to be made here is that, because of the "orientation phase," "Newly Hired Workforce" are, on the average, less productive than the "Experienced Workforce." Later, in our discussion on "Productivity" within the Software Production Subsystem in Section III.4.4., we will take another closer look at this issue in order to quantify this productivity differential.

This productivity differential was the first reason to disaggregate the workforce. The second reason was to capture the training overhead involved in adding new members to a software development project. This training of newcomers, both "technically" and "socially," is usually carried out by the "oldtimers" (T7), (Corbato and Clingen, 1979), (GRC, 1977), (Winrow, 1982), (Bott, 1982), (Lombardi, 16), (Thayer and Lehman, 1977). This is costly, because "while (the oldtimer) is helping the new employee learn the Job, his own productivity on his other work is reduced" (Canning, 1977).

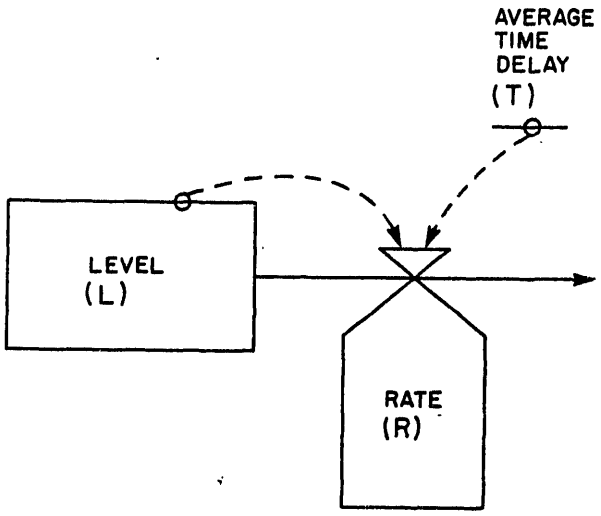
The determination of the amount of effort to commit to the training of new employees is made, we found, on the basis of managerial intuition and organizational custom. There are no proposed formulas in the literature, nor did we find any in the organizations we interviewed in. We did find,

however, rules-of-thumb, and these ranged from committing 15% of an experienced employee's time per new employee (Hisamune, 21) to a 25% commitment (Nichols, 18). In the model, the value of the parameter "Trainers per New Hiree" is set to 0.20 i.e., on the average each new employee consumes in training overhead the equivalent of 20% of an experienced employee's time for the duration of the training or assimilation period.

Estimates for the average assimilation period vary between 2 months (Lombardi, 16) and 6 months (Corbato and Clingen, 1979) (Brandon, 1970). In the model, the "Average Assimilation Delay" is set to 80 days. (Note: "Days" in the model represent working days. One week is five working days, and one year is 48 working weeks.) The assimilation delay is formulated in the model as a first-order exponential delay. Such delays are primary building-blocks of system dynamics models, and they are extensively used in this model. In Exhibit III.4., we show how a first-order exponential delay looks schematically, how it is formulated mathematically, and how it behaves over time.

Thus, if a number say L(0) of project members are recruited at time (0), they will be assimilated into the experienced workforce pool at a rate similar to the one shown in the figure of Exhibit III.4. That is, some will be assimilated quickly e.g., those recruited from within the

(A) SCHEMATIC



(B) MATHEMATICAL INFORMATION

At any time (t),

$$R(t) = L(t)/T$$

Also,

$$\frac{d}{dt} L(t) = - R(t) = - L(t)/T$$

Separating variables and integrating both sides yields,

$$L(t) = L(0)\, e^{-t/T}$$

And it can be shown that the average time spent in in the delay = T

(C) BEHAVIOR

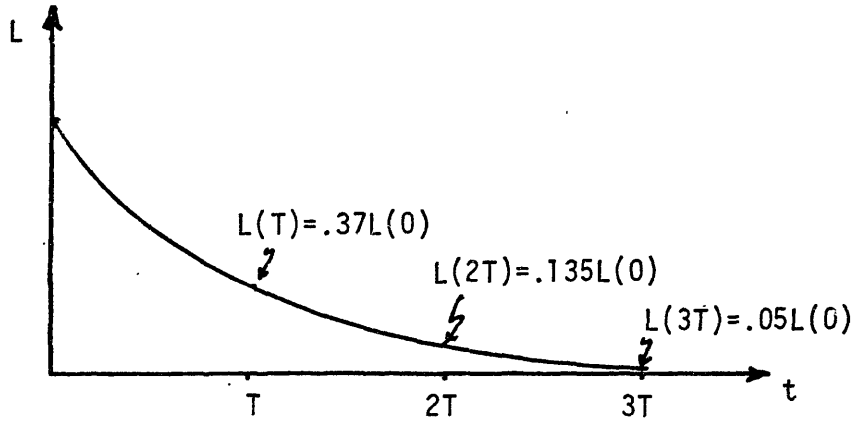


EXHIBIT III.4

company, others will take a much longer time e.g., new hirees fresh from school, while the average new employee will be assimilated at the "Average Assimilation Delay" i.e., in 80 days.

On deciding upon the "Total Workforce" level (i.e., newly hired plus experienced workforce) desired, project management considers a number of factors. One important factor is the current scheduled completion date of the project. As part of the planning function (see Section III.4.6. for details), management determines the workforce level that it believes is necessary to complete the project tasks perceived to be remaining within the scheduled completion time. In addition to that, consideration is also given to the "stability of the workforce." Thus, before hiring new project members, management tries to contemplate the duration of need for these new members. Different firms weigh this factor to various extents. In general, however, the relative weighing between the desire for workforce stability on the one hand and the desire to complete the project on time, on the other, changes with the stage of project completion. For example, toward the end of the project there could be considerable reluctance to bring in new people, even though the time and effort perceived remaining imply more people are needed. It would take too much time to acquaint new people with the mechanics of the project, integrate than into the project team, and train them

There is no text material missing here.
Pages have been incorrectly numbered.

in the necessary technical areas.

As will be further explained in the "Planning Subsystem," based on the above two considerations, management determines the "Workforce Level Needed." This level, however, still does not automatically translate into a hiring goal for the human resource management function. A further consideration is given to the project's ability to absorb new people into, to train them and make them an integral part of a productive team (Brooks, B23). We shall here recognize a policy, formal or more usually implicit, that the rate of hiring of new project members be restricted to that number which project managment feels its fully integrated staff can handle (Landolfi, 22) (Chan, 20).

This restriction is formulated in the model using the variable "Ceiling on New Hirees." Which simply equals the "Full-Time-Equivalent Experienced Workforce" level multiplied by the most number of new hirees that a single full-time experienced staff can be expected to effectively handle. In the model, the value of "Most New Hirees per Full-Time Experienced Staff" is set at 3.

Because in some organizations software developers are assigned to more than one project (i.e., the "Average Daily Manpower per Staff" per project would be less than 1 man-day), the "Full-Time-Equivalent Experienced Workforce"

level can be less than the "Experienced Workforce" level. So, for example, if there are only 2 experienced project members, each of which assigning 50% of his/her time to the project (i.e., "Average Daily Manpower per Staff" = .5) then we'll have .5 X 2 = 1 "Full-Time-Equivalent Experienced Staff." And in that case the "Ceiling on New Hirees" will be 1 X 3 = 3.

The summation of "Ceiling on New Hirees" and the value of the current "Experienced Workforce" level establishes the "Ceiling on Total Workforce." The value of this variable constitutes a ceiling on the number of employees sought i.e., to be hired. That is, "Workforce Level Sought" would be set to the value of "Workforce Level Needed" as long as this is less than or equal to the "Ceiling on Total Workforce." Otherwise, "Workforce Level Sought" is set to the value of the latter.

Thus, the three factors: (1) schedule completion time; (2) workforce stability; and (3) training requirements, all affect management's determination of the "Workforce Level Sought." Once the determination is made, management will face one of three possible situations. First, the "Workforce Gap" between the "Workforce Level Sought" and the current "Total Workforce Level" could be zero i.e., the two levels are exactly equal. In that case no further action is necessary.

A second, more likely, situation would be one where the "Workforce Level Sought" is larger than the current "Total Workforce Level." In this case, new employees will be hired. This, of course, takes time. The delay in hiring software professionals, is on the average, on the order of several months (McLaughlin, 1979). Some recruits are generally available in a short period from elsewhere in the organization, whereas others (especially when the project management is seeking special skills, or new college recruits) will not be available for a much longer time. After averaging these variables, the "Hiring Delay" is set to 40 days (McLaughlin, 1979) (Babich,9) (Hisamune, 26).

The third, and final, possibility would be for the "Workforce Level Sought" to be less than the current "Total Workforce Level." In this case, project members will be transferred out of the project. We will assume that if there are new recruits still in training i.e., in the "Newly Hired Workforce" level, then these will be the first to be transferred out. If still more transfers are needed, they would then be made from the "Experienced Workforce" pool.

Those who are being transferred out require some period of time e.g., for paper work and transfer arrangements, before they actually leave the project. The average transfer delay is set in the model to 10 days (Landolfi, 22).

Finally, there is the effect of turnover on the project's workforce. Turnover continues, of course, to be a chronic problem for software project managers. Willoughby (1977) estimates that annual turnover in the DP field ranged between 15 and 20% during the 1960s, declined to about 5% in the early 1970s, and began to rise again by the end of the decade. More recent studies place the annual turnover rate at 25.1% (Tanniru et al, 1981), 30% (Richmond, 1982) and even as high as 34% (Bott, 1982).

Turnover is captured in the model, through the "Quit Rate" of "Experienced Workforce." That is, we are assuming no turnover among the "Newly Hired Workforce," since it is quite unlikely for a new recruit to quit within 80 days of joining the project (i.e., during the assimilation period).

The annual turnover rate is set in the model to 30%. This translates into an "Average Employment Time" of 673 days. To see why, first notice from Figure III.3. that the "Quit Rate" is (as was the "Workforce Assimilation Rate") a first-order exponential delay. So, we can use the equation of Exhibit III.4.,

$$L(t) = L(0)*e^{-t/T}$$

where,

L = Experienced Workforce (men)

t = time (years)

T = Average Employement Time (years)

For a 30% annual turnover rate,

$$0.70L(0) = L(0)*e^{-1/T}$$

Thus,

$$T = 1/-\ln(.70) = 2.8 \text{ years}$$

Which translates into 673 days, since one year is 240 working days.

III.4.4. Software Production:

There are four primary activities in the Software Production Subsystem, namely, development, quality assurance, rework, and system testing. The development activity comprises both the design and coding of the software. As the software is being developed, it is also reviewed e.g., using structured-walkthroughs, to detect any design/coding errors. Errors detected through such quality assurance (QA) activities are then reworked. Not all errors will be detected during the development phase, however, some will "escape" and remain undetected until the testing phase.

This subsystem is too complex to diagram and explain as one piece. We will, therefore, break it into four sectors, namely:

(A) Manpower Allocation

(B) Software Development

(C) Quality Assurance & Rework

(D) System Testing

These sectors will be connected, not only through information-type variables, but also through flows e.g., software will flow from the "Software Development" sector to the "QA & Rework" sector and from there to the "System Testing" sector. To diagram such inter-sector flows we will make use of a new symbol, a "sector-symbol.." The symbol was proposed by Morecroft (1980), and is shown below:

The shape of the symbol has been selected to avoid any ambiguity or overlap with the standard system dynamics symbols. Figure III.4. shows how, for example, the symbol will be used to depict the flow of software into and out of the "QA & Rework" sector.

(A) Manpower Allocation:

The "Total Daily Manpower" available for the project is

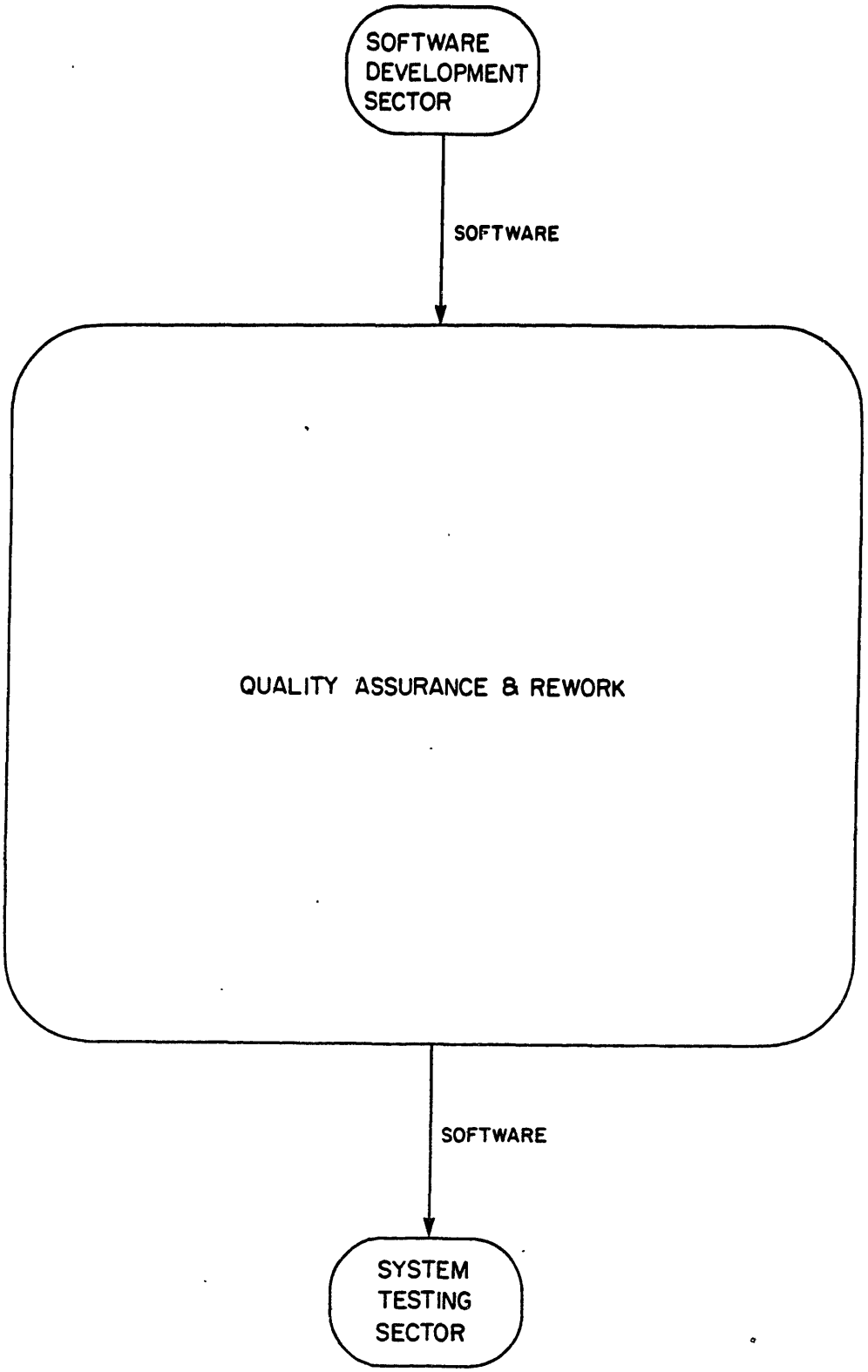

SOFTWARE
DEVELOPMENT
SECTOR
SOFTWARE
QUALITY ASSURANCE & REWORK
SOFTWARE
SYSTEM
TESTING
SECTOR

Figure III.4

simply a function of the "Total Workforce" level and the "Average Daily Manpower per Staff." In some organizations, software professionals are assigned to one project at a time. In such a case the "Average Daily Manpower per Staff" would be 1 man-day i.e., each staff member contributes 1 man-day every day on the project. In other organizations, however, software professionls are assigned to more than one project. So, for example, if on the average each staff member is assigned to say two projects on a 50-50 basis, then the "Average Daily Manpower per Staff," for each of the projects, would be 1/2 man-day.

Part of the available manpower will be consumed in training overhead, as was explained in Section III.4.3. The "Daily Manpower Available after Training Overhead" is what is then allocated to quality assurance, rework, software development and testing.

Quality assurance is defined in Pressman (1982) as a set of activities "... performed in conjunction with (the development of) a software product to guarantee the product meets the specified standards. These activities reduce doubts and risks about the performance of the product in the target environment." Several techniques are used including walkthroughs, reviews, inspections, code reading (a process where code logic and code format is scrutinized by a programmer other than the original designer), and

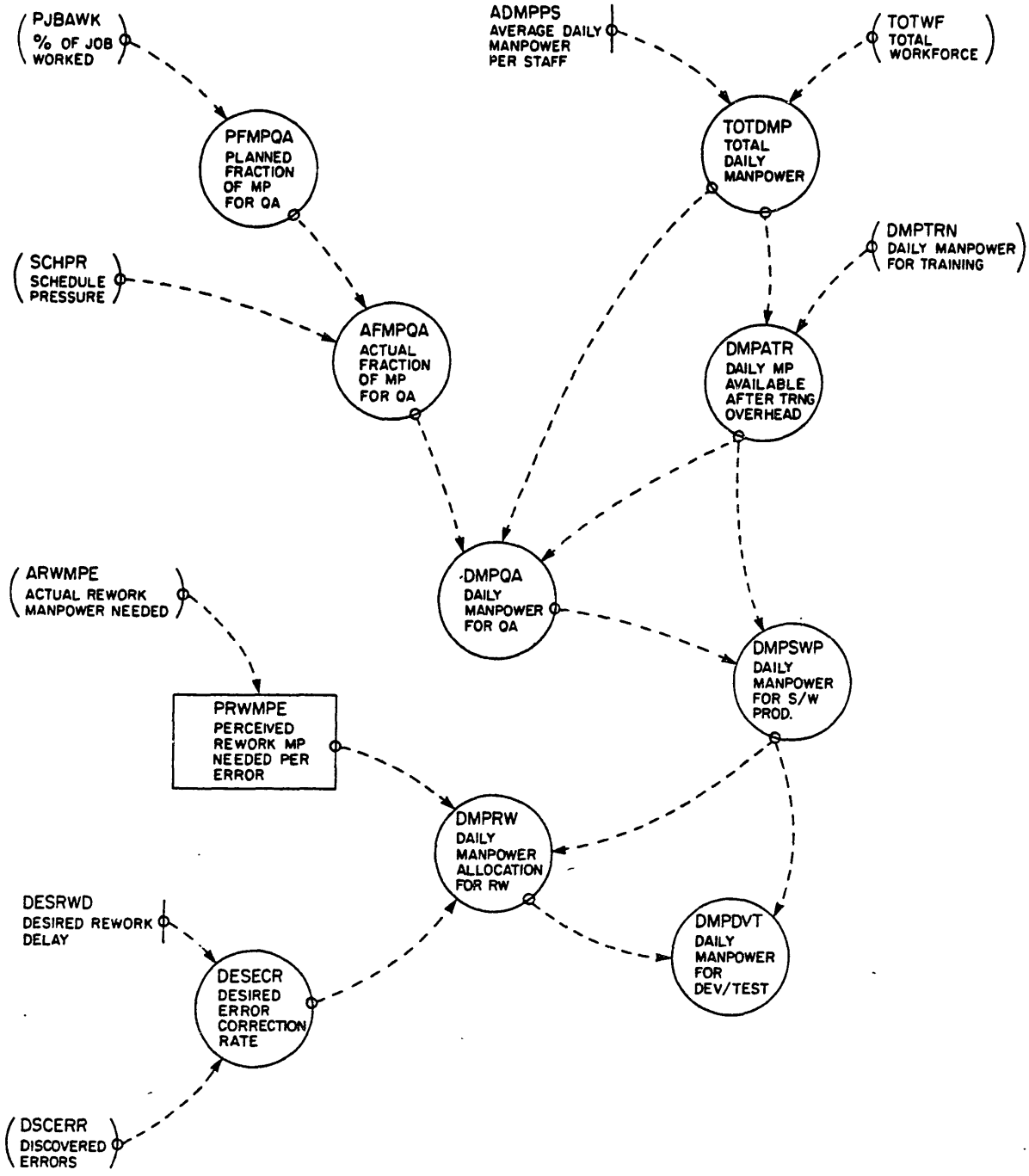

PJBAWK
% OF JOB WORKED
ADMPPS
AVERAGE DAILY MANPOWER PER STAFF
TOTWF
TOTAL WORKFORCE
PFMPQA
PLANNED FRACTION OF MP FOR QA
TOTDMP
TOTAL DAILY MANPOWER
DMPTRN
DAILY MANPOWER FOR TRAINING
SCHPR
SCHEDULE PRESSURE
AFMPQA
ACTUAL FRACTION OF MP FOR QA
DMPATR
DAILY MP AVAILABLE AFTER TRNG OVERHEAD
ARWMPE
ACTUAL REWORK MANPOWER NEEDED
DMPQA
DAILY MANPOWER FOR QA
DMPSWP
DAILY MANPOWER FOR S/W PROD.
PRWMPE
PERCEIVED REWORK MP NEEDED PER ERROR
DMPRW
DAILY MANPOWER ALLOCATION FOR RW
DESRWD
DESIRED REWORK DELAY
DMPDVT
DAILY MANPOWER FOR DEV/TEST
DESECR
DESIRED ERROR CORRECTION RATE
DSCERR
DISCOVERED ERRORS

Figure III.5

integration-testing (Jones, 1982) (Daly, 1977). Not included in this activity is unit or module testing, which is commonly considered to be part of the coding process (Mckeen, 1979).

There is a lack of data in the literature on actual quality assurance effort expenditures. There are, instead estimates, e.g., 6% of development effort (Knight, 1979), and 15-20% (Boehm, 1981).

In the organizations we interviewed in, estimates for the QA effort included 10% (Nichols, 27), 15% (Landolfi, 22), and in one case as high as 25% (Hisamune, 26).

In the model, the "Planned Fraction of Manpower for QA" will be set to a uniform 15% level. Notice, though, that in Figure III.5. the variable "Planned Fraction of Manpower for QA" is shown to be a function of "% of job worked." This will allow us to experiment with other QA policies i.e., ones in which the QA effort is not uniformly distributed through-out the life cycle.

As indicated in Figure III.5., the "Actual Fraction of Manpower for QA" can be different from the "Planned Fraction of Manpower for QA" because of schedule pressures. Several authors have observed that as schedule pressures mount, quality assurance activities are often relaxed (Mitchell, 1980) (Shooman, 1983) (Devenny, 1976) (Ergott, 1979). For

example, paraphrasing Glass (1982):

> Modules and changes were initially inspected in depth but with less severity as work pressure increased and greater risks were taken to meet delivery schedules.

Walkthroughs and inspections are usually the larger casualities. Under schedule pressures, they are not only relaxed, but often they are altogether suspended (Fagan, 1976). Hart (1982) provided an explanation:

> As the project progressed, there were the usual pressures to meet the project deadline. The walkthroughs were a natural area of concern in the schedule, since they represented a significant time commitment before their effectiveness was obviously demonstrated...
> As the deadline neared, there were pressures to hurry the walkthrough and, eventually, to 'temporarily suspended' them.

In the model, "Schedule Pressure" is formulated as follows,

Schedule Pressure = (TMDPSN - MDRM) / MDRM

where,

TMDPSN = Total Effort Perceived to be still needed to complete the project (Man-Days)

MDRM = Total Effort remaining in current plan (Man-Days)

Thus, when the project is perceived as being completely

on target i.e., effort still needed is exactly equal to the effort actually remaining in the project's budget, schedule pressure will be zero i.e., no schedule pressure. But, if the effort perceived still needed is say 150 man-days, while in the project's budget there is only 100 man-days left, then schedule pressure is 0.5. Conversely, if what is perceived to be still needed is less than what is remaining, than schedule pressure will be less than zero i.e., there is a slack.

The effect of schedule pressure on "Actual Fraction of Manpower for QA" is assumed to be as shown in Figure III.6. Such a graph i.e., that depicts a relationship (usually nonlinear) between two variables in a system dynamics model, is called a "Table Function." Table functions are used extensively in system dynamics modeling.

Table functions would be based on measurements, if such measurements are available. In many cases (including this one), however, measurements are not available i.e., there are no published data on the effect of schedule pressure on the QA effort.

> There seems to be a general misunderstanding to the effect that a mathematical model cannot be undertaken until every constant and functional relationship is known to high accuracy. This often leads to the omission of admittedly highly significant factors (most of the 'intangible' influences on decisions) because these are unmeasured or unmeasurable. To omit such variables is equivalent to saying they have zero effect

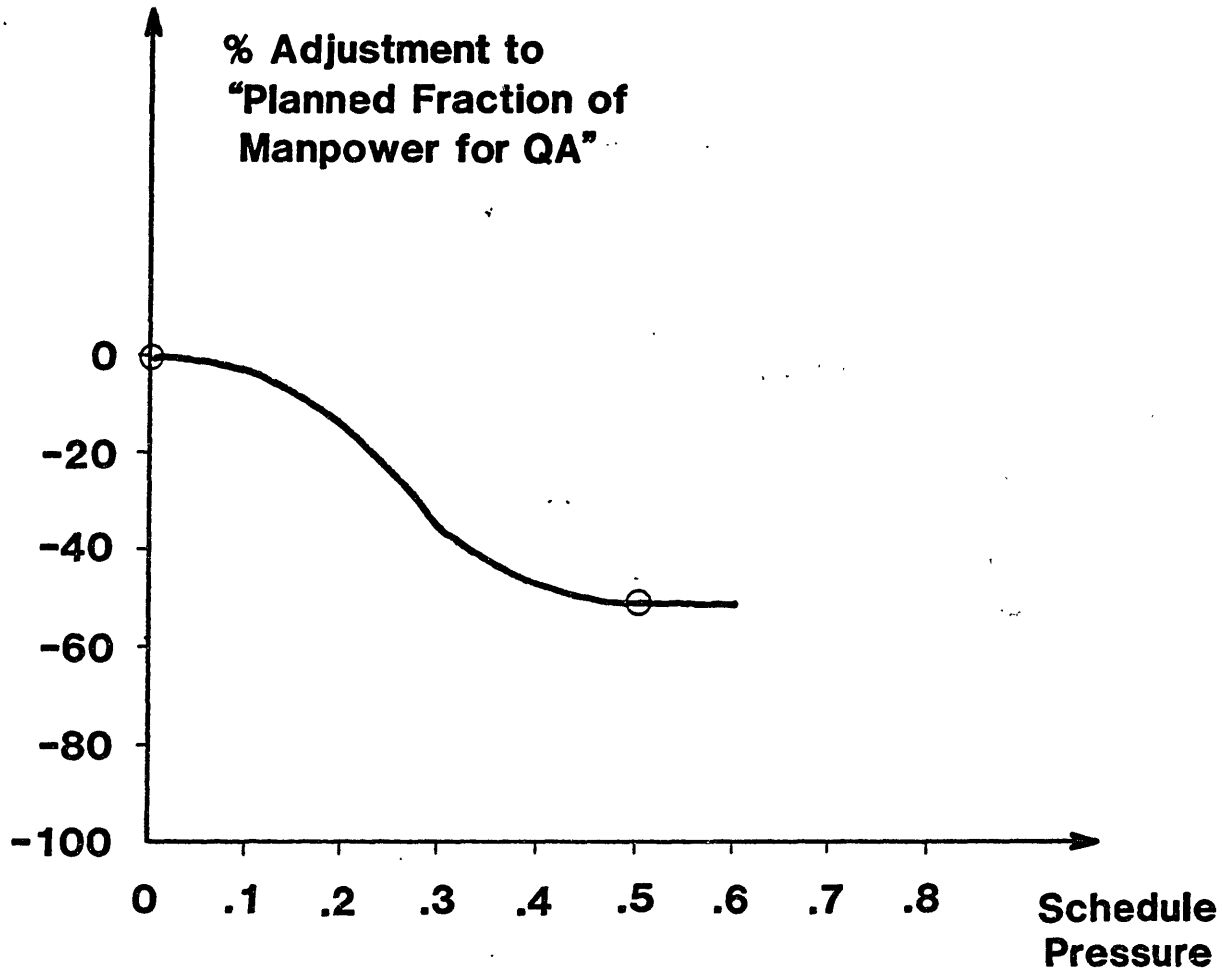

% Adjustment to
"Planned Fraction of
Manpower for QA"
0
-20
-40
-60
-80
-100
0
.1
.2
.3
.4
.5
.6
.7
.8
Schedule
Pressure

Figure III.6

> ... probably the only value that is known to be wrong ...
> A mathematical model should be based on the best information that is readily available, but the design of a model should not be postponed until all pertinent parameters have been accurately measured. That day will never come. Values should be estimated where necessary ... (Forrester, 1981).

Because of the lack of published measurements, it was necessary to estimate the relationship between schedule pressure and the QA effort. To give the reader a flavor of how both judgement and available information are used to formulate a Table Function, we will go through the formulation of Figure III.6. in some detail.

There are three potential considerations in formulating a table function: Slope, one or more specific points, and shape.

The slope of the relationship between schedule pressure and adjustments to QA effort is easy to determine. It must be negative, since, as the above quotes indicate, as schedule pressure increases, QA effort decreases.

We can also identify at least one point on the graph quite straight forwardly. It is the point (0,0) i.e., in the absence of any schedule pressure (i.e., "Schedule Pressure" is Zero), the % adjustment to the planned fraction of manpower effort for QA will be zero i.e., actual QA effort will be equal to the planned effort.

As schedule pressure mounts, quality assurance activities are relaxed i.e., cuts are made into the planned QA effort. QA activities are not, however, eliminated completely e.g., while walkthroughs might be decreased or even temporarily suspended, integration testing might not. In the judgement of the project managers we interviewed, planned quality assurance activities could be cut by as much as 50% under severe schedule pressures, which were defined as situations in which "Schedule Pressure" is equal to or greater than .5 (Gage, 4) (Babich, 9) (Nichols, 25) (Hisamune, 26). On the basis of these judgements (the best available information), the point (.5, -50) of Figure III.6. is identified.

The final step was to figure out the shape of the negatively sloping curve connecting the two points (0,0) and (.5, -50).

It is reasonable to expect the curve flattens out at the two extreme points. As schedule pressure starts to rise, people react, not only by cutting conners, they also start working harder (Boehm, 1981). This absorbs some of the effects of schedule pressure on QA effort allocations at the vicinity of point (0,0). Also, as indicated above, as schedule pressure increases it gradually reaches a saturation point at which it ceases to affect further adjustments to the QA effort i.e., the curve flattens at (.5, -50). And

finally, these two extreme flat parts of the curve are connected by a negatively sloping smooth curve. "Any sharply bent or kinked curve is probably not realistic. A bend or kink implies something special about the exact conditions at which the bend or kink occurs" (Graham, 1980).

Now, we resume our discussion of this section's main topic, namely, the allocation of the project's manpower resource. So far we have accounted for manpower resources consumed in training and quality assurance activities. The remaining bulk of the manpower resource, labelled in Figure III.5. as the "Daily Manpower for Software Production," is to be allocated to software development (i.e., design and coding), testing, and rework.

As software errors are detected through the quality assurance activities, manpower effort is allocated to correct them. The amount of daily effort allocated is a function of both the "Desired Error Correction Rate" i.e., the daily rate at which these discovered errors are to be corrected, and the "Perceived Rework Manpower Needed per Error." In other words, the effort is allocated on the basis of the rework job to be done, and the perceived rework productivity.

The "Perceived Rework Manpower Needed per Error" is diagramed in Figure III.5. as a special kind of a level, namely, one with an input that is not a rate. This is a

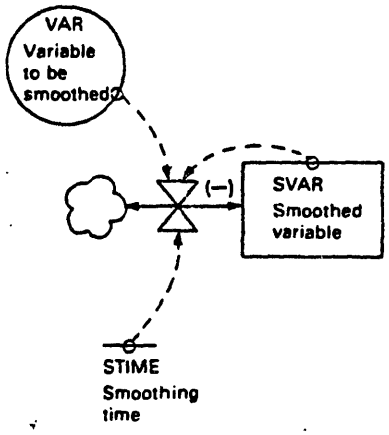
"shorthand notation" for an exponential smoothing operation. That is, "Perceived Rework Manpower Needed per Error" is the exponential smooth of its input, the "Actual Rework Manpower Needed per Error." (Because smoothing or averaging of information accumulates that information, a smoothed variable is represented by a level's rectangular symbol.)

Why smooth? Because, "Full and immediate action is seldom taken on a change of incoming information (e.g., on the sudden drop in yesterday's rework productivity) ... (There is a) tendency to delay action until the change is insistent ..." (Forrester, 1961).

A full schematic representaion of the smoothing operation is shown in Figure III.7., together with its mathematical formulation. (Readers familiar with smoothing formulations may want to observe that the equation for a smoothed variable can be written in the familiar weighted-average form for exponential smoothing.) In Figure III.7., we also show the behavior of the "smoothed variable" in response to a spike in the "variable to be smoothed."

Thus, a sudden change (e.g. increase) in the "Actual Rework Manpower Needed per Error," will not initially affect the project member's rework-manpower allocation decisions. If, however, the increase persists over a period of time, the change will be perceived as permanent (i.e., "Perceived

(A)



(B) $$SVAR_T = SVAR_{T-1} + \frac{\Delta T}{STIME} \left(VAR_{T-1} - SVAR_{T-1} \right)$$

(C)

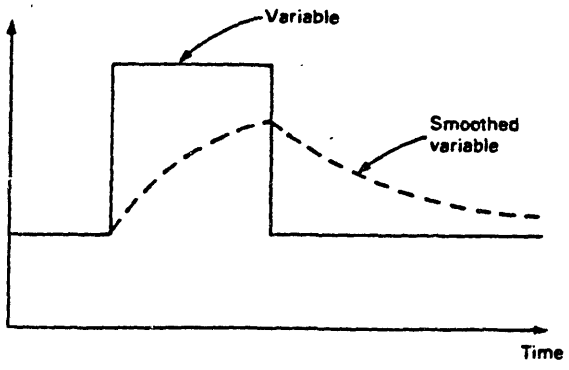


Figure III.7

Rework Manpower Needed per Error" catches up with the actual) and thus incorporated in the allocation decision making process. The smoothing time is set in the model at 10 days.

As mentioned above, the amount of daily effort allocated for rework activities is a function of not only the "Perceived Rework Manpower Needed per Error," but also the "Desired Error Correction Rate," i.e., the daily rate at which the discovered errors are to be corrected. For example, if it is desired to correct one error a day, and if it is perceived than one Man-Day is needed on the average to correct an error, then one Man-day will be allocated daily for rework activities.

The "Desired Error Correction Rate" is the value of the total number of discovered errors divided by a "Desired Rework Delay." When an error is detected, it, usually, is not immediately corrected. Some time elapses before a software professional "deals" with it. In a TRW study (Thayer et al, 1978) this delay was found to be in the range of 8-19 days. The "Desired Rework Delay" is set in the model to 15 days (James, 1) (Lombardi, 16).

As is shown in Figure III.5., after manpower is allocated to rework activities, the remaining (often larger) portion of the "Daily Manpower for Software Production" is devoted to the development (i.e., design and coding) and

testing activities. These activities are discussed in detail below in Sectors (B) and (D) respectively.

(B) Software Development:

Figure III.8. depicts the software development process i.e., the design and coding of the software product. A software project will be defined in terms of a number of "Tasks." Thus, the software development rate will be in terms of "tasks per day," software developed in terms of "tasks" developed, and software development productivity in terms of "tasks per man-day." (A precise definition of a "Task" will be provided shortly, when we discuss nominal productivity.)

As we indicated earlier, after manpower allocations are made for training, quality assurance, and rework activities, the remaining bulk of the available manpower resource is allocated to the development of the software product. This continues until it is perceived that most of the software development tasks are completed, at which point the System Testing phase is initiated. This switch in manpower utilization is affected in the model through the variable "Fraction of Effort for System Testing." The value of "Fraction of Effort for System Testing" is initially set at zero, i.e., no effort is allocated for System Testing. When all development tasks are perceived to be completed, the

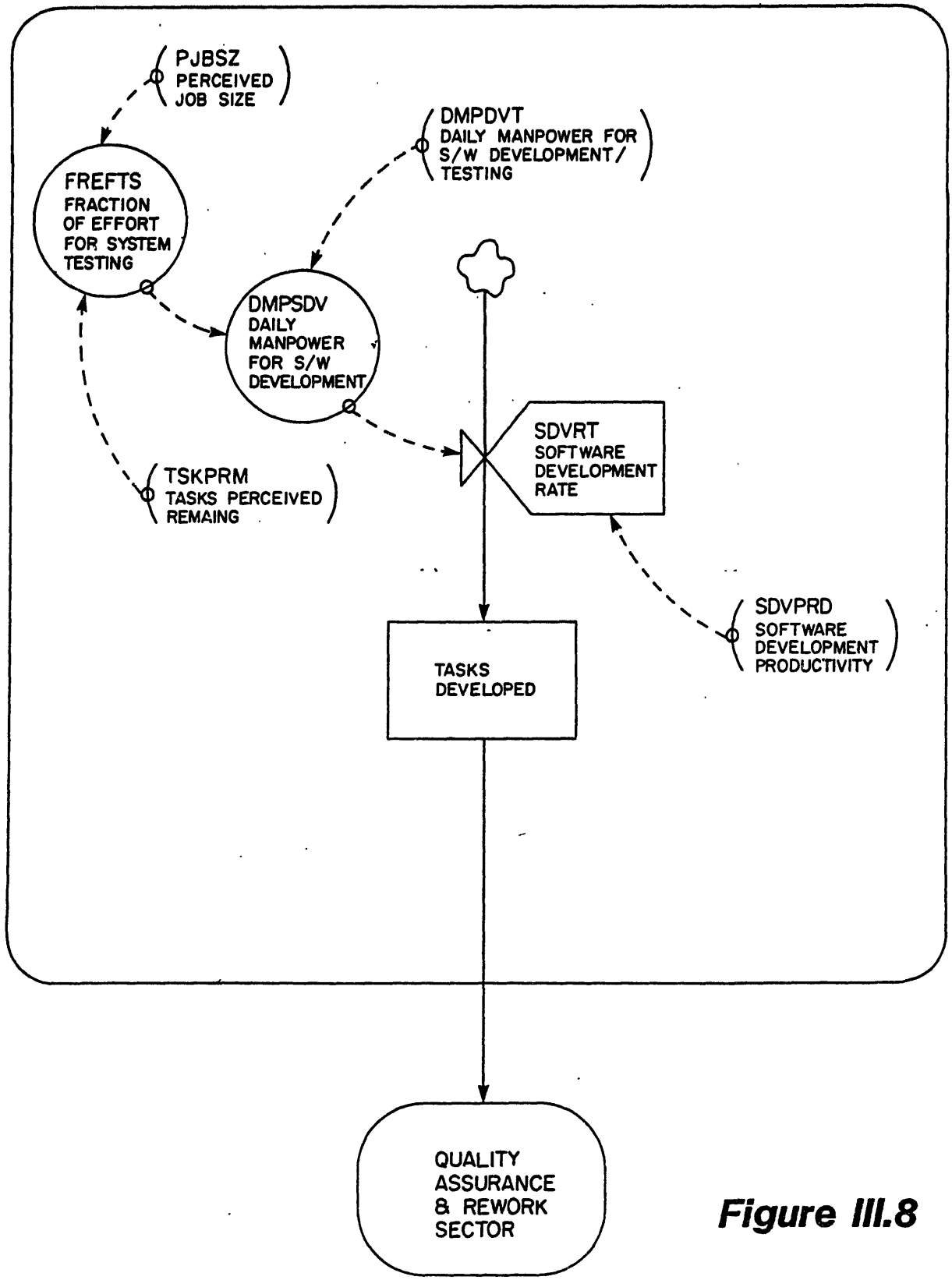

PJBSZ
PERCEIVED
JOB SIZE
DMPDVT
DAILY MANPOWER FOR
S/W DEVELOPMENT/
TESTING
FREFTS
FRACTION
OF EFFORT
FOR SYSTEM
TESTING
DMPSDV
DAILY
MANPOWER
FOR S/W
DEVELOPMENT
SDVRT
SOFTWARE
DEVELOPMENT
RATE
TSKPRM
TASKS PERCEIVED
REMAING
SDVPRD
SOFTWARE
DEVELOPMENT
PRODUCTIVITY
TASKS
DEVELOPED
QUALITY
ASSURANCE
& REWORK
SECTOR

Figure III.8

value of the "Fraction of Effort for System Testing" becomes a 1, i.e., 100% of the effort available for software development/testing is utilized in system testing activities. The switch is not abrupt, however. There is, usually, some overlap between the development and testing phases (Thibodeau and Dodson, 1980) (Daly, 1977) (Hartwick, 1980). For example, the design of test cases usually commences towards (not at) the end of the software development phase (Adrion et al, 1982). This overlap of the phases is captured in Figure III.9. It shows the assumed gradual increase in the value of the "Fraction of Effort for System Testing" as a function of the fraction of development tasks perceived remaining.

During the software development phase, the rate at which the software will be developed will be a function of not only how much manpower is utilized, but in addition, it will also depend on the productivity of the software developers (as is shown in Figure III.8.).

"Software Development Productivity" is a function of a complex set of factors, and as such it comprises a significant portion of the model. We are, therefore, using a separate figure, to provide a detailed depiction of its formulation.

Our formulation of the productivity of the software development group is based on a model of group productivity

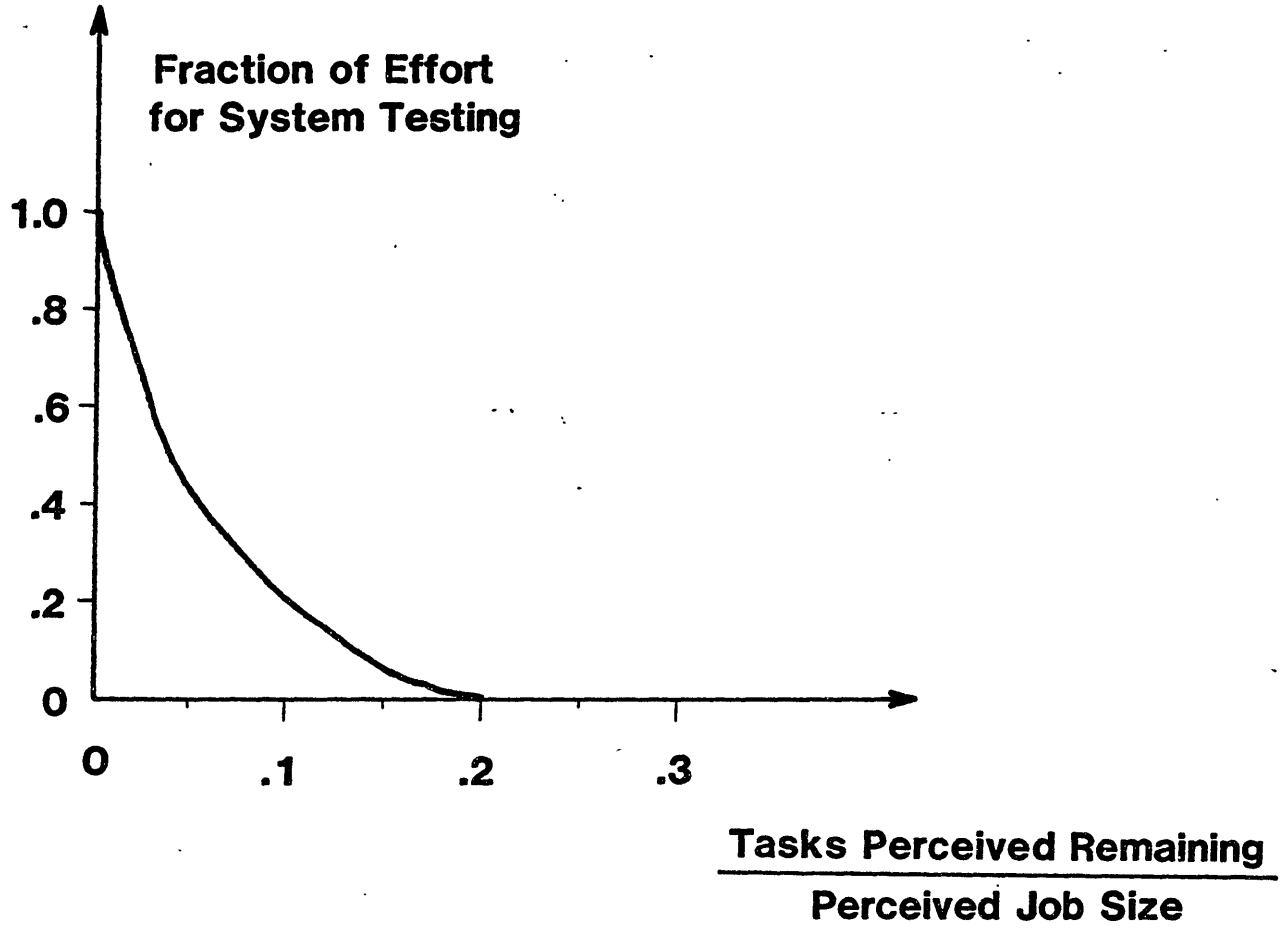

Fraction of Effort
for System Testing
1.0
.8
.6
.4
.2
0
0
.1
.2
.3
Tasks Perceived Remaining
Perceived Job Size

Figure III.9

in the Psychology literature proposed by Ivan Steiner (1966). The model can be simply stated as follows:

Actual Productivity = Potential Productivity - Losses Due to Faulty Process

Where Losses due to faulty process refer basically to communication and motivation losses

> Potential productivity is defined as the maximum level of productivity that can occur when an individual or group employs its funds of resources to meet the task demands of a work situation. It is the level of productivity that will be attained if the individual or group makes the best possible use of its resources (that is, if there is no loss of productivity due to faulty process)... Potential Productivity can be inferred from a thorough analysis of task demands and available resources, for it depends only upon these two types of variables.
> Actual productivity, what the individual or group does in fact accomplish, rarely equals potential productivity. Individuals and groups usually fail to make the best possible use of their available resources. Problems of coordination and/or motivation are responsible for inadequacies in process, and for consequent losses in productivity (Steiner, 1966).

The three pieces of Steiner's model, namely, actual productivity, potential productivity, and communication/motivation losses are all incorporated in the formulation of Figure III.10. Their structures fall in the middle part, the left part, and the right and bottom parts of the figure, respectively.

According to Steiner, potential productivity is a function of two determinants, the nature of the task and the

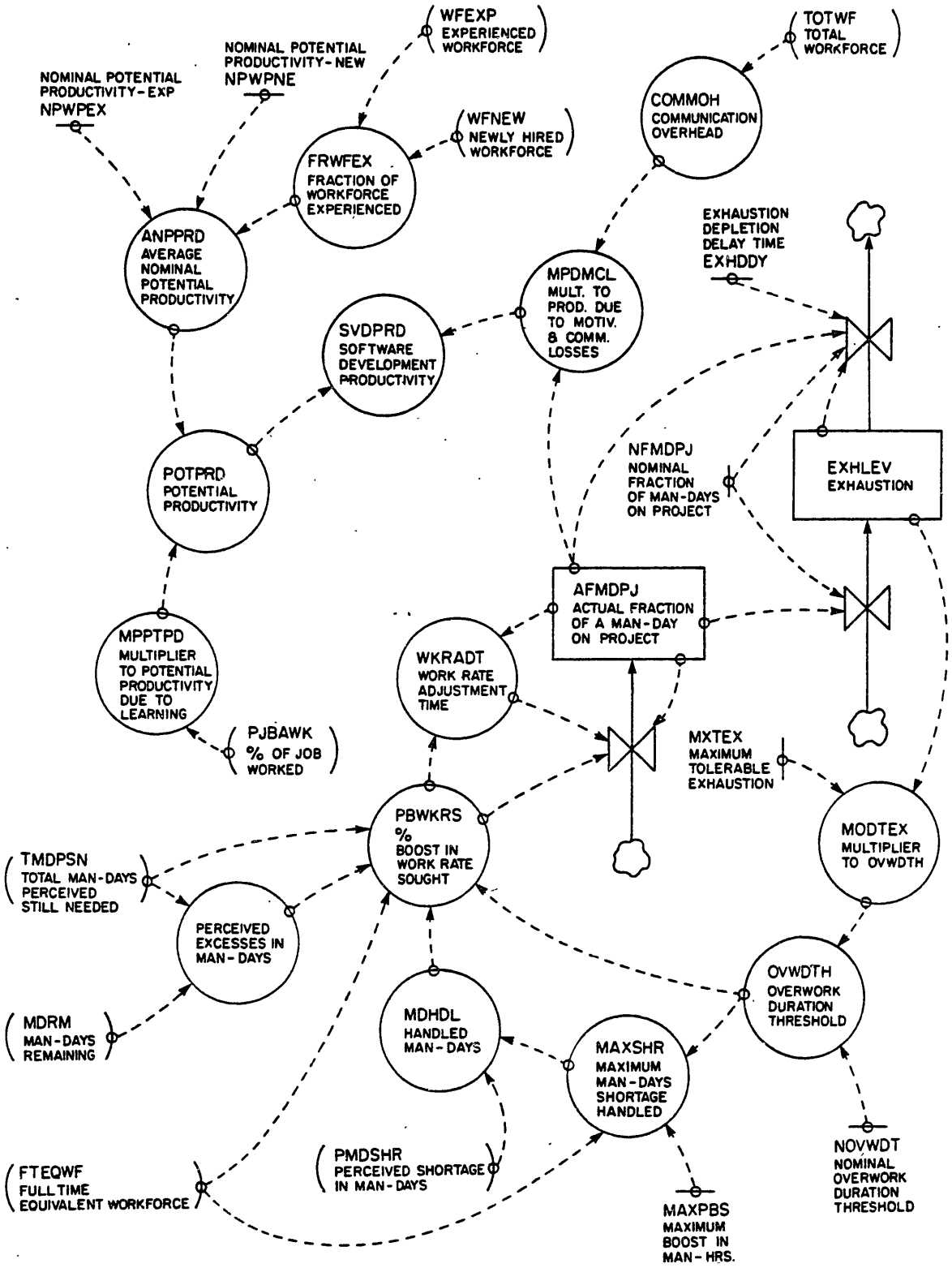


Figure III.10

group's resources. The effects of these two sets of factors on the productivity of software development has been investigated in the software engineering literature. However, because the idea of distinguishing between actual and potential productivity didn't take root in the software engineering literature (yet), in all such studies the dependent variable is always the actual productivity of software development.

For example, Scott and Simmons (1974) used the Delphi technique "to determine what programming project variables have the greatest impact on programmer productivity." They identified three resource-type variables including, the availability of programming tools, the availability of programming practices, and programmer experience, as well as two task-type variables, namely, the programming language and the quality of external documentation, as all having significant influence on productivity.

Boehm's COCOMO software cost estimation model (1981), incorporates the following determinants of productivity:

(1) Task-type Variables: Product complexity, required reliability, memory constraint, and database size.

(2) Resource-type Variables: Software tools available, turnaround time, and personnel experience.

Finally, Chrysler (1978) mapped several research findings into a model that categorizes the determinants of software productivity into 6 categories. Three of the categories were of the task-type, they were "programming problem characteristics," "Source Language," and "Computer Hardware Characteristics." The other three categories included resource-type factors, and they included "Programmer Characteristics," "Organizational Characteristics," and "Programming Mode."

Notice that most of the above factors, while they would vary from organization to organization (e.g., availability of software tools, personnel capability, and computer-hardware characteristics) and from project to project within a single organization (e.g., programming language, database size, and product complexity) they would, however, remain constant within a single project. From our modeling viewpoint, this observation is quite significant. It means that, in modeling the behavior of a single software development project, most of the above variables would remain constant and can, therefore be simply captured by a single constant parameter in the model. Such a parameter would then need adjustments only when modeling different projects and/or different organizations.

This is achieved in the model through the formulation of the "Nominal Potential Productivity" parameter. It

represents the maximum level of software development productivity that can occur when an individual employs his/her fund of resources to meet the task demands for the specific work situation modelled i.e., a specific project within a specific organization.

The value of the "Nominal Potential Productivity" parameter will be defined in terms of a number of "Tasks/man-day." Which, of course, means that its value depends on what we define a "Task" to be. This provides us with two options in modeling different project situations in which the nominal potential productivity differs e.g., due to differences in the degree of complexity of the project. We can either fix in the model what a "Task" is defined to be, and change the value of the "Nominal Potential Productivity" parameter, or we can do the reverse, that is, fix the value of the "Nominal Potential Productivity" parameter to say (X) tasks/man-day, while changing the value of what a "Task" is.

We opted for the second alternative. We will, therefore, define "Nominal Potential Productivity" to be a certain number, (X) (to be specified shortly) of tasks/man-day, and formulate "Task" as a parameter in the model that can be set at different values to reflect different project and resource characteristics.

A "Task" is essentially some unit for sizing up a

software product. In principle, a "Task" can be any arbitrary unit by which we can measure a software project's size e.g., it can be defined in terms of lines of code, function-points, modules, input/output files, ... etc. From a practical point of view, though, the "lines of code" unit is the most attractive alternative. Defining our sizing measure, the "Task", in terms of "lines of code" provides us with direct access to most published results on software productivity measurements.

A "Task" is, therefore, defined in terms of a number of Delivered Source Instructions (DSI). The definition of Delivered Source Instructions (DSI), as provided by Boehm (1981), is as follows:

> Delivered. This term is generally meant to exclude nondelivered support software such as test drivers. However, if these are developed with the same care as delivered software, with their own review, test plans, documentation, etc., then they should be included.
>
> Source Instructions. This term includes all program instructions created by project personnel and processed into machine code by some combination of preprocessors, compilers, and assemblers. It excludes comment cards and unmodified utility software. It includes job control language, format statements, and data declarations. Instructions are defined as lines of code or card images. Thus, a line containing two or more source statement counts as one instruction; a five-line data declaration counts as five instructions.

Let us provide an example to further clarify the concepts of "Normal Potential Productivity" and "Task." Assume two different software development organizations,

(ORG-1) and (ORG-2), have each just completed the development (i.e., design and coding) of a software project. The two projects, (PROJ-1) and (PROJ-2), are two completely different projects (e.g., one is an embedded piece of software for a military satellite and the other a payroll system), except that they are both exactly 8000 DSI in size. Now, let us assume that in (ORG-1) the development effort consumed a total of 400 man-days to design and code the 8000 DSI (PROJ-1), while in (PROJ-2) the development effort was 200 man-days. If for purposes of simplification, we disregard the communication and motivation losses in both organizations i.e., assume that actual productivity = potential productivity, we could then conclude that the potential productivity in (ORG-1) is half that of (ORG-2). This distiction would be realized in the model as follows: The "Nominal Potential Productivity" parameter would be defined in both runs of the model at the same value, say 1 Task/Man-day, but in the (PROJ-1) run we would define a Task to be 20 DSI, while in the (PROJ-2) run a "Task" would be set at 40 DSI. That is, the 8000 DSI project (PROJ-1) will be defined in the first run as a 400 Task project, while the 8000 DSI project (PROJ-2) would be defined as a 200 Task project.

We have thus far only addressed one set of factors that affect the potential productivity on a software development project, namely, those factors which remain constant

throughout a particular project. While most of the factors listed in the literature are of this variety, at least two are not, namely, workforce experience level (Chrysler, 1978) and increases in project familiarity due to learning-curve effects (Crowley), (Shell, 1972), (Weinberg, 1982).

To capture the effect of experience, we will formulate two nominal potential productivity parameters, one to represent the nominal potential productivity of the average experienced staff member, and the second represents that of the average newly hired employee. And at any point in time in the project the "Average Nominal Potential Productivity" for the workforce as a whole would be the weighted average of the two parameters, (in which each parameter is weighted by the fraction of its corresponding employee-type in the total workforce). Thus, while the two nominal potential productivity parameters for the two types of employees remains constant throughout a project, the project's "Average Nominal Potential Productivity" may not, since the mix of experienced and new employees could (and probably would) change.

We will take the nominal potential productivity of an average <u>experienced</u> staff member to be our reference point, and define it to have a value of 1 Task/Man-day. The value of the nominal potential productivity of the average employee within the newly-hired workforce pool is then determined

relative to that 1 Task/Man-day reference point. In the literature, estimates for the productivity of a newly hired staff member relative to that of an experienced staff member included 0.45 (Weiss, 1973), 0.5 (Okada, 1982), 0.6 (Toellner, 1977), and 0.64 (Boehm, 1981) (Benbasat and Vessey, 1980). Estimates provided from interviews ranged from 0.33 (Hisamune,26) to 0.5 (Lombardi, 16). It should be noted, however, that all these estimates are for actual productivities and not potential productivities. But since there is no evidence to suggest that there are significant differences in the communication and motivation losses between the two types of employees, we will accept the above estimates as a "reasonable" approximation for the ratio between the potential productivities of the two groups of employees. The value of the nominal potential productivity for the average newly hired employee is, accordingly, set in the model to 0.5 Task/Man-day.

The second factor affecting potential productivity, in the model, is the increased project know-how due to the learning-curve effect (Crowley) (Shell, 1972) (Weinberg, 1982). "As a project proceeds, the implementers learn their job better. The 'learning curve' is the rate of improvement" (Aron, 1976). Several authors have suggested that an S-shaped type learning-curve characterizes this "rate of improvement" in the software development environment (Crowley) (Weinberg, 1982). Reflecting on his experience at

IBM, Aron (1976) estimates that the total improvement for a medium sized project (e.g., 12-24 months long) would be a 25% improvement in productivity.

In the model the learning curve effect is formulated as the variable "Multiplier to potential Productivity Due to Learning." It is, as is shown in Figure III.11, S-shaped and it is a function of progress in the project, starting with a value of 1 at the beginning of the project, and peaking at a value 25% higher (i.e., at 1.25) towards the end of the development period.

As defined above, potential productivity is the level of productivity that will be attained if the individual or group makes the best possible use of his/its resources (that is, if there is no loss of productivity due to faulty process). However, due to losses caused by communications and motivation problems actual productivity, i.e., what the individual or group does in fact accomplish, rarely equals potential productivity (Steiner, 1966).

In the model, "Software Development Productivity" is formulated as the product of "Potential Productivity" and the "Multiplier to Productivity Due to Communication and Motivation losses." In the absence of any communication and motivation losses the multiplier assumes a value of 1, in which case actual productivity would be equal to potential

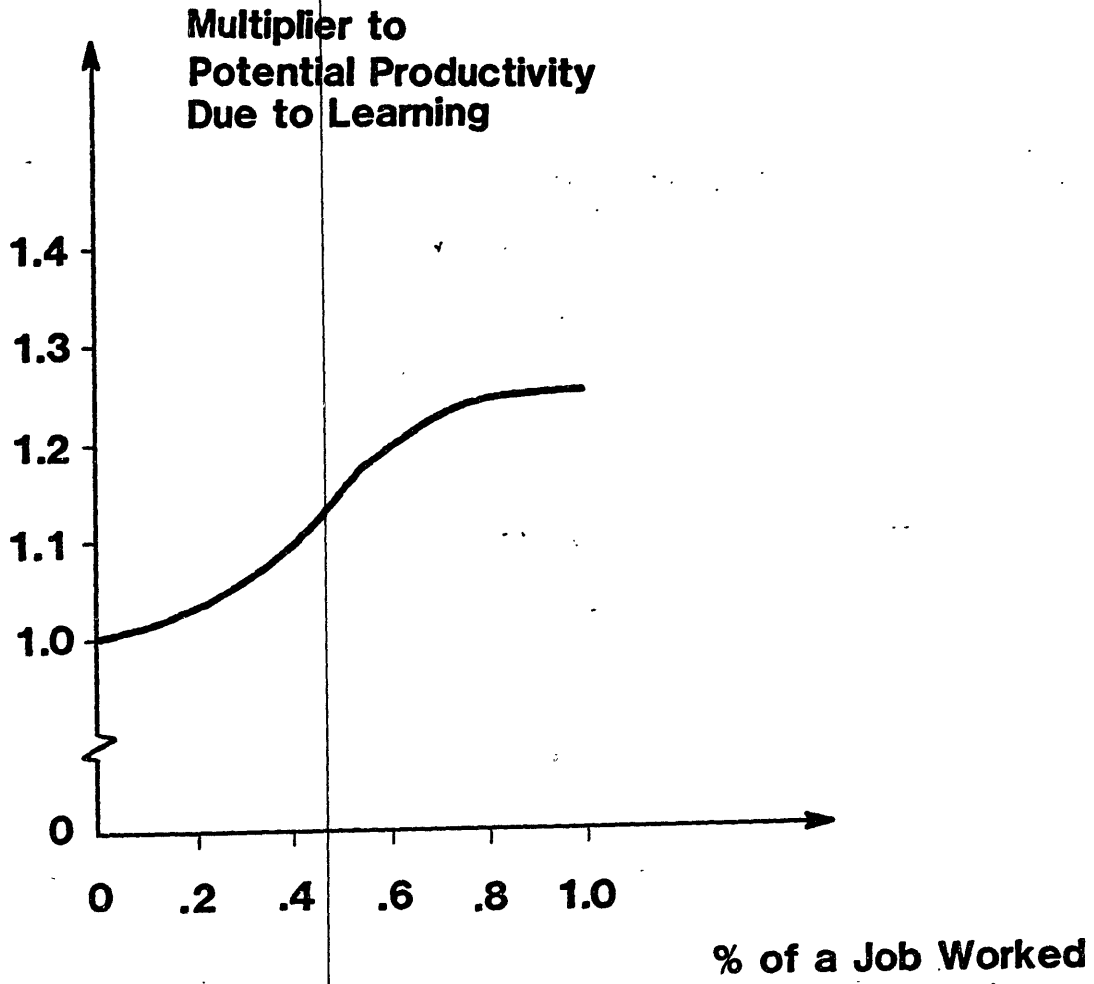


Figure III.11

productivity. However, losses will occur, and these will drive the multiplier to values that are less than 1, thus depressing the value of actual productivity to levels below that of potential productivity.

The "Multiplier to Productivity Due to Communication and Motivation Losses" has the following interpretation. It represents the average productive fraction of a Man-Day. In other words, if the nominal man-day for a full-time employee is 8 hours, because of communication and motivation losses, the daily contribution by the average employee to the project will be less than 8 man-hours. For example, if the communication and motivation losses amount to a 4 man-hour loss per day (for the average employee) i.e., half the nominal 8 man-hour value, then the value of the multiplier would be a 0.5.

The effects of communication and motivation are multiplicative. Motivation factors first determine the fraction of a man-day devoted to project work. This fraction will usually have a value less than 1, since time is often lost on personal matters, coffee-breaks, and other miscellaneous non-project related activities. Communication losses refer to project-type communication losses, and are thus formulated as a fraction of "project hours" i.e., the hours devoted to project work, hence the multiplicative formulation of the two components of productivity loss. The

detailed formulation of the effects of both communication and motivation losses on productivity are shown in Figure III.10.

In considering the effects of motivation losses on productivity we need to make the same distinction we made while formulating the "potential productivity" structure, that is, between those factors that would remain constant during a single project (while possibly varying between projects and between organizations) and those that could change throughout the life of the single project. A reference back to our review of the literature on motivation (in Chapter II) would indicate that most of the motivational factors identified and studied e.g., possibility for growth, advancement, responsibility, salary, company policy and administration, ... etc., are of the former variety i.e., factors that tend to characterize the overall organizal setting and climate. Such invariant factors would therefore be "implicitly" incorporated within the definition of the potential productivity parameters.

"Another motivation approach which is particularly appropriate to the data processing area is goal setting" (Bartol and Martin, 1982). The authors further suggest that project goals and schedules can play a significant motivational role throughout the life of a software development project.

Boehm (1981) went a step further and provided the means to "operationalize" this idea. He suggests that the motivational role of schedule pressures and project deadlines is to expand or contract the project members' "slack time." The slack time being the fraction of project time lost on off-project activities, e.g., coffee-breaks, personal business, non-project communication, ... etc.

The motivation mechanism in the model is designed to capture this motivational impact of schedule pressures on "slack time." That is why, motivation losses are formulated, as indicated above, in terms of man-hour losses.

In the absence of schedule pressures, which can be either positive (i.e., when the project is perceived to be behind schedule) or negative (i.e., when the project is perceived to be ahead of schedule), the fraction of daily hours allocated to project-related work by the average full-time team member is defined by the parameter "Nominal Fraction of a Man-Day on Project." In designating a value for this parameter, we were able to draw upon the experiences of our interviewees as well as that of a large number of authors. And we found that most of the estimates were clustered within the 50-70% range, e.g., 50% (Brooks, 1978) (Nichols, 25), 50-60% (Gehring and Pooch, 1977) (Pooch and Gehring, 1980), 60% (Basili and Zelkowitz, 1979), and 70% (Boehm, 1981). In addition, Stalnaker (1968) reported on the

results of a large study that investigated how software professionals utilize their time. The findings indicated, on the basis of over 7000 observations of a group of production programmers, that 35% of the time was lost on "Personal activities," "being away or out," and other "miscellaneous" non-project related activities. Furthermore, within the remaining 65% of the available working time, there were further losses e.g., time spent on mail, company business, ... etc.

On the basis of the above findings, the value of the parameter "Nominal Fraction of a Man-Day on Project" was set to 60% i.e., in the absence of schedule pressures, a full-time employee would allocate, on the average, 0.6 X 8 = 4.8 hours to the project (assuming an 8-hour day). Under these nominal conditions, therefore, the "contribution" of motivation losses to the "Multiplier to Productivity due to Motivation and Communication Losses" amounts, in effect, to a 40% cut in potential productivity.

The loss in productivity due to motivational factors, does not, of course, remain constant at the 40% level throughout the life of the project. The motivational effects of schedule pressures can push the "Actual Fraction of a Man-Day on Project" to both higher (under positive schedule pressure) as well as lower (under negative schedule pressure) values i.e., leading to motivation losses that would be lower

than the 40% level in the former case, but higher in the latter.

As shown in Figure III.10, the "Actual Fraction of a Man-Day on Project" is formulated in the model as a level variable. Its value is set, at the initiation of the project, to the value of "Nominal Fraction of a Man-Day on Project" i.e., at 60%. And it maintains that nominal value at the absence of any schedule pressures. To see how schedule pressures influence the "Actual Fraction of a Man-Day on Project," let us first consider the effects of positive schedule pressures.

Schedule pressure was previously defined as,

Schedule Pressure = (TMDPSN-MDRM)/MDRM

where,

TMDPSN = Total Effort perceived to be still needed to complete the project (Man-Days)

MDRM = Total Effort remaining in current plan (Man-Days)

Positive schedule pressures arise whenever the project is perceived to be behind schedule. That is, whenever the total effort still needed to complete the project is perceived to be greater than the total effort actually remaining (i.e., when the numerator in the schedule pressure

equation is positive). Such a difference represents a perceived shortage in man-days on the project.

When confronted with such a situation, software developers tend to work harder, i.e., allocate more man-hours to the project, in an attempt to compensate for the perceived shortage and bring the project back on schedule (Larkin) (Ibrahim, 1978) (McGowan, 3) (Babich, 9) (Lombardi, 16) (Nichols, 18) (Sheldon, 19) (Chan, 20) (Hisamune, 21). In one experiment, Boehm (1981) found that the number of man-hours increases by as much as 100%. And he asserts that most of the gains are achieved by "reallocating (i.e., compressing) peoples' slack time." In other words, under schedule pressure, people tend to spend less time on off-project activities such as personal business and non-project communication. This then decreases the man-hours lost per man-day, while increasing the daily man-hours allocated to the project.

Recall that the value of the "Nominal Fraction of a Man-Day on Project" was set to 60%, which translate into 4.8 hours of project work per man-day. This would seem to indicate that, at most, another 3.2 hours per man-day can be gained under schedule pressure (assuming an 8-hour day), i.e., a 67% increase. And since it is quite unlikely that people would in fact allocate every minute of their 8-hour working day to project work, the attainable increase will be

even less than 67%. How then could we explain the 100% increase reported by Boehm?

A 100% increase is attainable because workers, in addition to partially compressing their slack time, may also work overtime hours. For example, by working 12 hours a day at 80% efficiency, a team member would be allocating 9.6 hours to the project i.e., double the nominal 4.8 hours.

In fact, by further compressing the slack time (say to 10 or 15%) and/or increasing the overtime hours, an increase of more than 100% could be achieved. But this would cause actual productivity to be larger than potential productivity, which by definition should not be possible. That is, by the current definitions. To accomodate this situation, we, therefore, amend the definition of potential productivity to be "the level of productivity that will be attained if the individual or group makes the best possible use of its resources under regular working conditions," and define "regular" to exclude overtime working conditions.

To recapitulate, when a project is perceived to be behind schedule, people tend to work harder to bring it back on schedule. They do that by compressing their slack time and/or working over-time, and thus allocating more man-hours to the project. But what if such a situation persists ... would workers be willing to work harder indefinitely? The

answer, according to our interviewees, was overwhelmingly no [(McGowan, 3), (Babich, 9), (Lombardi, 16), (Nichols, 18), (Sheldon, 19), (Chan, 20), and (Hisamune, 21)]. There is, it was indicated, a threshold on how long employees would be willing to work at an "above-normal" rate.

We refer now to Figure III.10. to explain how the above set of findings is implemented in the model.

When the project is perceived to be behind schedule i.e., when the total effort still needed to complete the project is perceived to be greater than the total effort actually remaining in the project's plan, two factors determine the level to which the "Actual Fraction of Man-day on Project" is boosted. The first is the value of the "Perceived Shortage in Man-days" i.e., the value of the difference between what is needed and what is remaining. If this difference is below some "threshold," then it will <u>all</u> be handled, i.e., the employees will boost the hours they allocate to the project (e.g., by compressing their slack time) to what they perceive is necessary to handle <u>all</u> the "Perceived Shortage in Man-days." (How they determine this will be explained shortly.) The second factor is the "Maximum Shortage in Man-Days to be Handled," and it constitutes the "threshold" mentioned above. Thus, if the "Perceived Shortage in Man-Days" is greater than the maximum which the employees are willing to handle, we will assume

that they would be motivated to work harder to handle that maximum value, while arranging with management to extend the schedule so as to handle what exceeds the "Maximum Shortage in Man-Days to be handled." (Such extension to the schedule will be explained in the Planning Section.)

As employees work harder to handle shortages in man-days, their tolerence for working harder decreases i.e., the value of the "Maximum Shortage in Man-Days to be handled" decreases. For if this were not true, e.g., if this maximum value was a constant parameter, then a persistent man-days shortage at moderate levels (i.e., at levels below the maximum value) would lead to an above normal work rate throughout the life of the project. And this, would contradict our finding that "there is a threshold on how long employees would be willing to work at above normal rate."

At any point in the project, the value of the "Maximum Shortage in Man-Days to be handled" is determined by the product of three variables, the "Overwork Duration Threshold," the "Full-Time Equivalent Workforce," and the "Maximum Boost in Man-Hours." For example, if at a point in time the workforce of 10 full-time people on the project is willing to work at an above normal rate for a maximum of 10 days, and they figure that they can boost their work rate by as much as 100% (e.g., allocate 9.6 hours per man-day to the project instead of the normal 4.8 hours) then they would

conclude that during this 10 day period it is possible to handle 10 X 10 X 1 = 100 Man-days worth of backlogged work, over and above the regular work planned for that period.

In the model, the value of the "Maximum Boost in Man-Hours" is set, as in the example above, at a value of 100% (Lombardi, 16) (Nichols, 27).

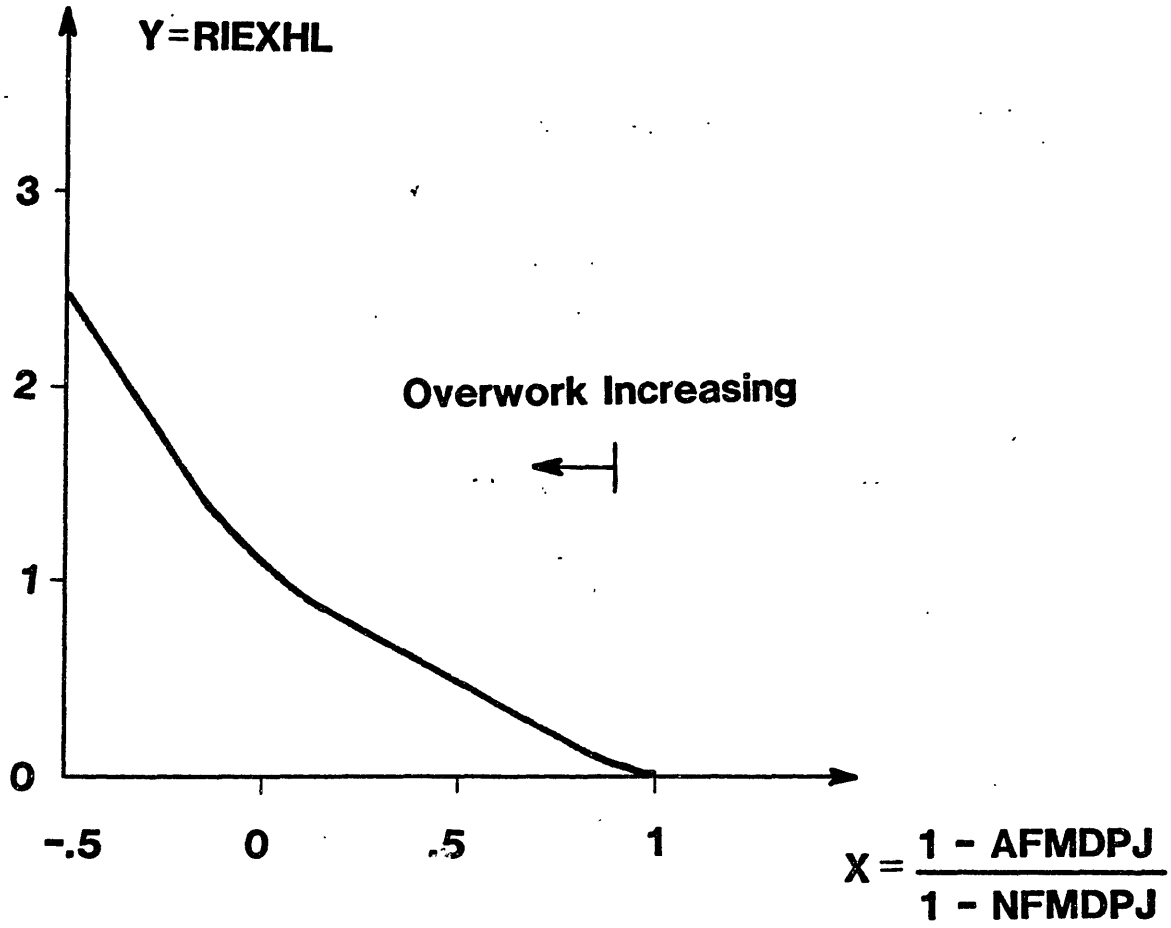
Estimates by the interviewees for a nominal value for the "Overwork Duration Threshold" ranged from 8 weeks (Chan, 20) to 12 weeks (Nichols, 27). In the model we set the nominal value for the "Overwork Duration Threshold" to 50 working days (i.e., 10 weeks). Once people start working harder, their "Overwork Duration Threshold," which at any point in time would represent the maximum remaining duration for which they would be willing to continue working harder, would decrease below the nominal value. Thus the "Overwork Duration Threshold" is formulated as a nominal value (i.e., of 10 weeks) that is adjusted downwards by a multiplier. One option for the multiplier was to have it be a function of the calendar time during which the project members, have been working harder. This option was rejected, though, because it would not differentiate between say a ten day period during which the staff were working 10% harder, and another ten day period in which they worked 100% harder. We wanted the formulation of the multiplier to induce a cut in the "Overwork Duration Threshold" that would be greater at the

end of the latter case.

This was accomplished by formulating the "Multiplier to the Overwork Duration Threshold due to Exhaustion." Where "Exhaustion" is simply a level whose value reflects the level of exhaustion of the workforce due to overwork. The rate at which this level increases needs, therefore, to be a function of some measure of overwork. Such a function is shown in Figure III.12.

Before interpreting Figure III.12., let us first refresh our memories about some assumptions we've made so far. First, we are assuming that a full time employee allocates, on the average, 60% of his or her time to the project (i.e., NFMDPJ = 0.6), which for an 8-hour day amounts to 4.8 hours. Under schedule pressure, more time will be allocated to the project (i.e., AFMDPJ > 0.6). This would be achieved by first compressing the slack time, and then if needed, by working overtime. Furthermore, we are also assuming that there is a "Maximum (Possible) Boost in Man-Hours" of 100% i.e., AFMDPJ can attain a maximum value of 0.6 X 2 = 1.2.

The first thing to note about Figure III.12. is that when AFMDPJ is less than or equal to NFMDPJ (i.e., when X is greater than 1) the value of RIEXHL is zero. That is, when people are working at their normal pace (or slower) there will be no rise in their exhaustion level. This must be so



Where,

AFMDPJ=Actual Fraction of a Man-Day on Project

NFMDPJ=Nominal Fraction of a Man-Day on Project

RIEXHL=Rate of Increase in Exhaustion Level

Figure III.12

by definition, since the "Exhaustion level" in the model is defined to be that of exhaustion due to overwork.

Second, note that the exhaustion rate is really a function of (1-AFMDPJ), since the denominator of (X) i.e., (1-NFMDPJ), is a constant term. Also note that the value of (1-AFMDPJ) is a measure of the average "Slack Time." What we are saying, therefore, is that the exhaustion rate of the workforce is a function of the compression in the average slack time. And the reason is this: the exhaustion of working harder is mostly "psychological," rather than "physiological." That is, people enjoy their slack time (e.g., coffee breaks, social communications, personal business, ... etc.), and they would not tolerate prolonged deprivation of such "breathers." Thus a compressed slack time exhausts them in the sense that it cuts into their tolerance level for continued hard work since that would mean a continued "deprivation" of their slack time.

However, when the value of (1-AFDPRD) approaches zero and moves into negative territory, people would, not only be compressing their slack time, but they would in addition be working overtime. At those values, in addition to the psychological component to exhaustion, there will also be "physiological" exhaustion. And that is why, the curve increases at a faster rate for negative values of (X).

The effects of exhaustion on the "Overwork Duration Threshold" is formulated as the "Multiplier to the Overwork Duration Threshold due to Exhuastion." As we explained previously, the nominal value of the threshold is 50 days. And as people start working harder, i.e., at a rate above their normal rate, that threshold is cut down, until possibly it reaches a value of zero. But notice that setting the nominal value of the "Overwork Duration Threshold" to 50 days is not enough. It is also necessary to specify at what level of overwork, since people might be willing to work for 50 days at a rate 50% above their normal rate, while not willing to do so at a 100% increase. We thus amend our definition of the nominal value for the "Overwork Duration Threshold" to be 50 working days at a rate of 8 hours per man-day (i.e., when AFMDPJ is approximately 1). Notice that when AFMDPJ is approximately 1, RIEXML in Figure III.12. would be also 1 i.e., at such a work rate, each man-day contributes 1 to the Exhaustion level. And after 50 such days, the Exhaustion level reaches a level of 50, which should be enough to drive the "Overwork Duration Threshold" to zero. That level of Exhaustion is termed the "Maximum Tolerable Exhaustion" level. That level of exhaustion could of course be reached in less than a 50 day duration if people are working even harder (i.e., if AFMDPJ is greater than 1), and conversely, if the work rate is less than 8-hours per man-day, it would be reached in more than 50 days. But once reached, it drives the "Overwork Duration Threshold" to zero. This is

accomplished by the formulation of the "Multiplier to the Overwork Duration Threshold due to Exhaustion," shown in Figure III.13.

Once a period of overwork comes to an end, either because the threshold has been reached and/or schedule pressures cease, and the workforce returns to a normal work rate (i.e., when AFMDPJ = NFMDPJ), the workforce's "Exhaustion level" depletes. The "Rate of Depletion of the Exhaustion level" is modeled as a first order exponential delay, with a time delay equal to 4 weeks. The 4 weeks delay time was chosen on the basis of discussion with (Lombardi, 23) and (Nichols, 25).

During the "de-exhausting" period, the workforce remains unwilling to "re-overwork" (Lombardi, 23) (Nichols, 25). This is achieved in the model through the formulation of the variable "Willingness to Overwork." This is a SWITCH variable that can attain one of two values, namely, zero or one, and is multiplied into the formulation of the "Maximum Shortage in Man-Days to be Handled." Whenever the maximum exhaustion level is reached and the "Overwork Duration Threshold" is driven down to zero, the "Willingness to Overwork" variable is switched to zero. The "Willingness to Overwork" variable will remain at that zero level until the workforce is "de-exhausted" i.e., until the "Exhaustion Level" is depleted. And as long as the "Willingness to

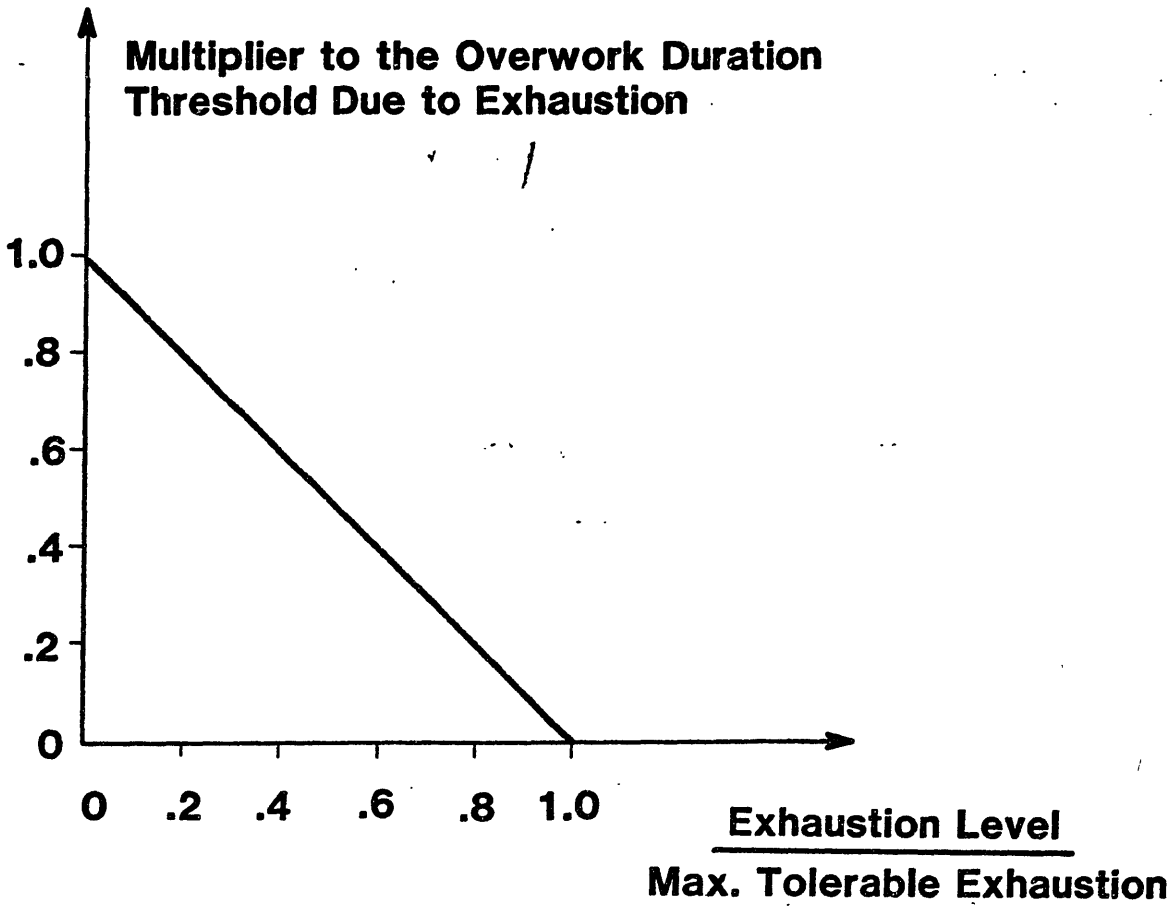


Figure III.13

Overwork" is zero, the "Maximum Shortage in Man-Days to be Handled" will also be zero i.e., the worforce remains unwilling to handle any (further) man-day shortages through overwork. When the "Exhaustion Level" is eventually depleted, the "Willingness to Overwork" is switched back to a value of one i.e., the workforce would again be willing to overwork (if and when the need arises).

Recall that determining the value of the "Overwork Duration Threshold" was necessary in order to determine the value of the "Maximum Shortage in Man-days to be Handled." The latter, in turn, is necessary to determine the value to which the "Actual Fraction of Man-days on Project" is boosted. When the project is perceived to be behind schedule i.e., when the total effort still needed to complete the project is perceived to be greater than the total effort actually remaining in the project's plan, indicating a shortage in man-days, the staff members would then seek to boost their work rate to what they perceive is necessary to handle either all the "Perceived Shortage in Man-Days" or the "Maximum Shortage in Man-Days to be Handled," which ever is smaller. The smaller of the two values would then constitute the "Handled Man-Days." The "% Boost in Work Rate Sought" to handle these man-days is determined by dividing the value of "Handled Man-Days" by the product of "Full-Time Equivalent Workforce" and "Overwork Duration Threshold." For example, if 100 man-days are to be handled by a 10 person team in 50

days, the % Boost would be 100/(10X50) = 0.2. That is the workers would figure that by increasing their work rate by 20% they can handle the 100 man-days of backlogged work in addition to the regular work planned for the 50 day period. Notice our assumption that the backlogged work will always be stretched over the full period defined by the "Overwork Duration Threshold." This should be a good approximation in cases when the value of "Handled Man-Days" is close to the "Maximum Shortage in Man-Days to be handled." When the "Handled Man-Days" is much smaller, though, the team might decide to handle it in a shorter "spurt" of overwork e.g., "to get it over with." However, we will simplify and use a single formulation for all cases (i.e., one in which the backlog is stretched over the "Overwork Duration Threshold" period).

Once the "% Boost in Work Rate Sought" is determined, it defines a work rate goal in terms of the man-hours to be allocated to the project. Such a goal is not achieved instantaneously, since workers take time to adjust their work habits. There is, therefore, a delay before the "Actual Fraction of Man-Days on Project", in fact attains the level sought. The average delay is set in the model to 2 weeks.

So far we have been discussing the effects of positive schedule pressures on productivity. To both complete and conclude this discussion on the effects of motivational

factors on productivity, we turn our attention next to those (probably rare) situations in which the project is perceived to be ahead of schedule i.e., the case of negative schedule pressures.

Such a situation exists whenever the total man-days remaining in the project's plan exceed what the project members perceive to be needed to complete the project. This could happen, for example, if management over-estimates a project's scope. The question we are interested in addessing here is what effects would a perception of such "excesses" have on productivity, if any?

Recall, in the case of positive schedule pressures, the shortage in man-days was handled first by adjustments in productivity and then if needed by additional adjustments in the schedule. Analogous behavior occurs in the negative schedule pressure situation. That is, when project members perceive some "excesses" in the schedule parts, if not all, of those excesses will be "absorbed" by the workers, in the form of "under-work," before downward adjustments are made in the project's schedule (Ibrahim, 1978) (Boehm, 1981) (Griffin, 6) (Babich, 9) (Lombardi, 16) (Sheldon, 19). For example, paraphrasing Boehm (1981):

> ... if the software cost or schedule estimate for meeting a milestone is higher than the ideal, Parkinson's Law indicates that people will use the extra time for ... personal activities, catching up on the

mail, etc.

Again, analogous to the positive schedule pressure situation where there was a limit on how much backlog could be handled, there are limits on how much "fat" employees would be willing, or allowed, to absorb. And beyond those limits, excesses would be translated into cuts in the project's schedule.

The above ideas are captured in the table function of Figure III.14. The dashed 45° line represents full disclosure of schedule excesses, and thus the complete translation of any excesses into schedule cuts. A more realistic project behavior is the one depicted by the Solid Curve. At the upper right corner excesses are small i.e., "Man-Days Perceived Still Needed" is slightly less than "Man-Days Remaining" in the plan. Under such conditions most of the slack will be absorbed (not reported) i.e., reports will show that the project is <u>on</u> (not ahead of) schedule i.e., "Man-Days Reported Still Needed" will be equal to "Man-Days Remaining." As we move towards conditions of larger and larger excesses those large excesses will be only partially absorbed, and the balance translated into cuts in the project's schedule.

Absorbed excesses will mean, as was indicated above, a larger slack time, which in turn means a lower "Actual

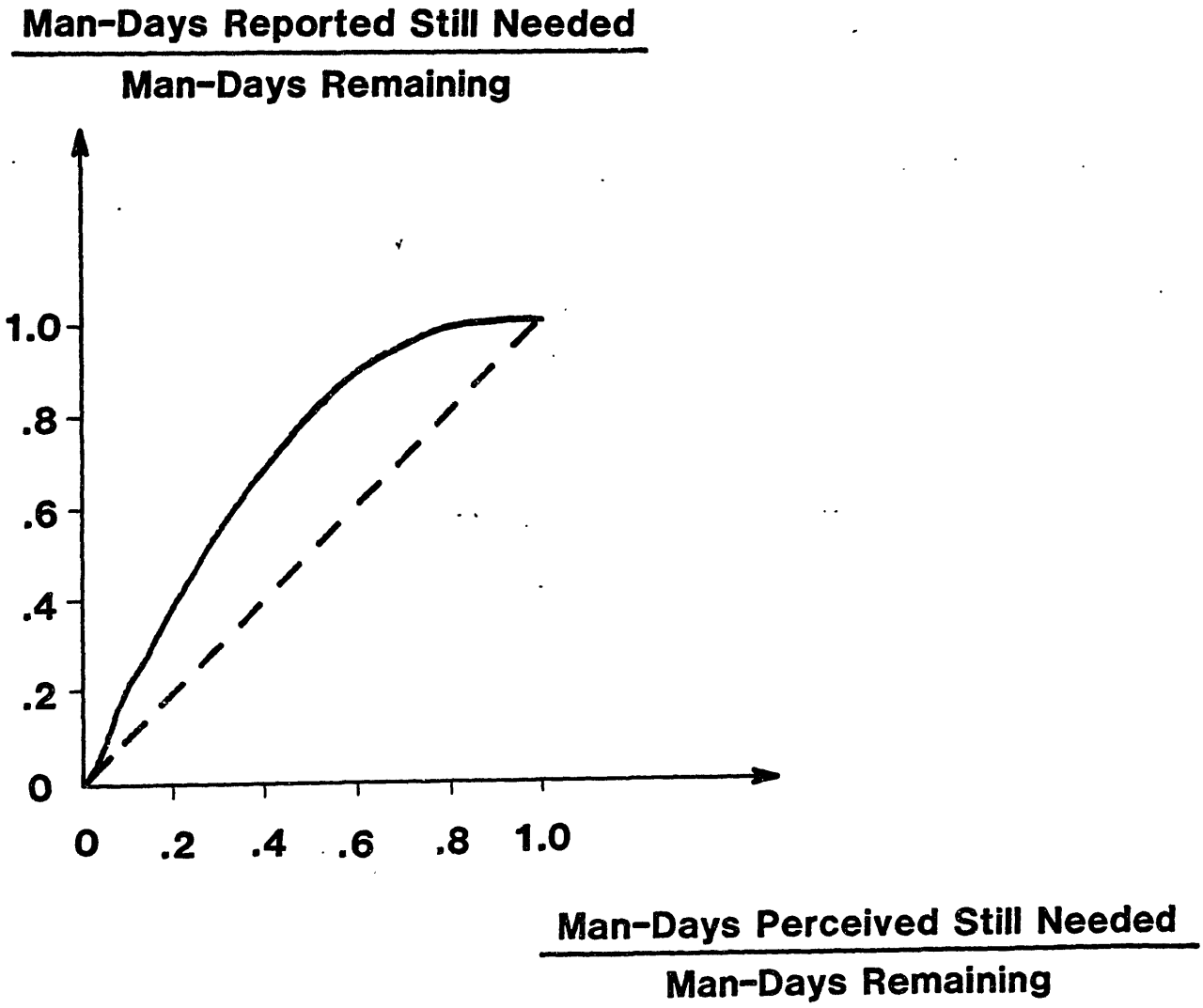


Figure III.14

Fraction of a Man-Day on Project." This is brought about in the model through the same mechanism used to increase the "Actual Fraction of a Man-Day on Project" under positive schedule pressure, namely, through an adjustment to the value of the variable "% Boost in work Rate Sought." In this case, however, the % boost will be a negative value.

There are, in addition, two more differences between the two cases. In calculating the % boost, we will assume that the workers will stretch their absorption of the perceived excesses over the remaining life of the project. That is, instead of a short lived and drastic dip in their work rate, workers are assumed to adjust to what they perceive would be a stable, albeit comfortably lower, work rate.

Once the "% (DIP) in work rate Sought" is determined, it defines a work rate goal in terms of the man-hours to be allocated to the project. As in the positive schedule pressure situation, such a goal is not achieved instantaneously, since workers take time to adjust their work habits. It is reasonable to expect, though, that the delay to adjust one's habits to a more comfortable state would be a smaller delay than that of adjusting to a less comfortable state. We, therefore, will assume that the average delay in adjusting to a "% Dip" is 7.5 days i.e., 25% lower than that of adjusting to a "% Boost" under positive schedule pressure.

The value of the "Actual Fraction of a Man-Day on Project," once determined under various schedule pressure conditions, becomes an important determinant of the actual software development productivity. It represents, as indicated above, the losses in productivity due to motivational factors. It is not the only determinant, though. Additional losses in productivity are incurred due to the communication overhead.

As is shown in Figure III.10., "Software Development Productivity" is formulated as the product of "Potential Productivity" and the "Multiplier to Productivity Due to Communication and Motivation Losses." The multiplier represents the average productive fraction of a Man-Day, i.e., that fraction of the "Actual Fraction of a Man-Days on Project" that remains after accounting for communication overhead. For example, if the "Actual Fraction of a Man-Day on Project" is 0.6 i.e., a full-time employee allocates on the average .6 X 8 = 4.8 hours to the project, and if the project communication overhead consumes 25% of that, then the average productive fraction of a Man-Day would be 0.75 X 0.6 = 0.45 i.e., 3.6 hours.

What is communication overhead? There are those who might argue that human communication is an essential component of any software development effort, and is, therefore, actually part of the "job" ... not an overhead.

Even though human communication is indeed an essential (and even useful) component of software development, it does constitute an overhead. To see why, let us examine what happens when a software system rather than being developed by a team is instead developed by one person.

Two things usually happen. First, time lost in human communication is avoided. When a team is developing the software,

> ... it is necessary that each individual spend part of his time communicating with each of the other team members. For example, the designer must confer with the coder to resolve any questions the coder may have about the design; both of these must talk to the individual testing the code to give him the benefit of their experience with the program; each of these must talk to the documentor to assure that the documentation is proper and complete; and so on (Tausworthe, 1977).

Such human communication is, obviously, unnecessary when the software is developed by a single person.

Second, the amount of work itself usually increases when software is developed by a team, vis-a-vis a single person. This increase in the work load takes two forms. The first, and obvious one, is that the amount of documentation increases e.g., in a one-person environment the programmer could get away with sketchy notes to merely augment his "mental documentation" (Tausworthe, 1977). The second less obvious increase is in the form of an increase in the size of

the software product itself (Gagliardi, 1980) (Conway, 1968). For example, when a program is developed by two people instead of one, it might be designed as a two-module program instead of a single-module program necessitating an inter-module interface that has to be agreed upon and developed.

On the basis of the above observations, we can now answer the question we posed above; namely, "what is communication overhead?" The answer: It is the drop in the productivity of the average team member below his nominal productivity due to team communication. Where communication includes verbal communication, documentation, and any additional workload e.g., due to interfaces.

It is widely held that communication overhead increases in proportion to n^2, where n is the size of the team (Brooks, 1978) (Shooman, 1983) (Mills, 1976) (Zelkowitz, 1978) (Scott and Simmons, 1975). Such a relationship is shown in the table function of Figure III.15. Thus, communication overhead, as is formulated in the model, is zero when the software is developed by one person, but as the workforce size (n) increases, communication overhead increases in proportion to n^2. For example, at n=30, the communication overhead is approximately 50%. This means that if the "Actual Fraction of a Man-Day on Project" is 0.6, i.e., 4.8 hours are allocated daily, on the average, by the full-time

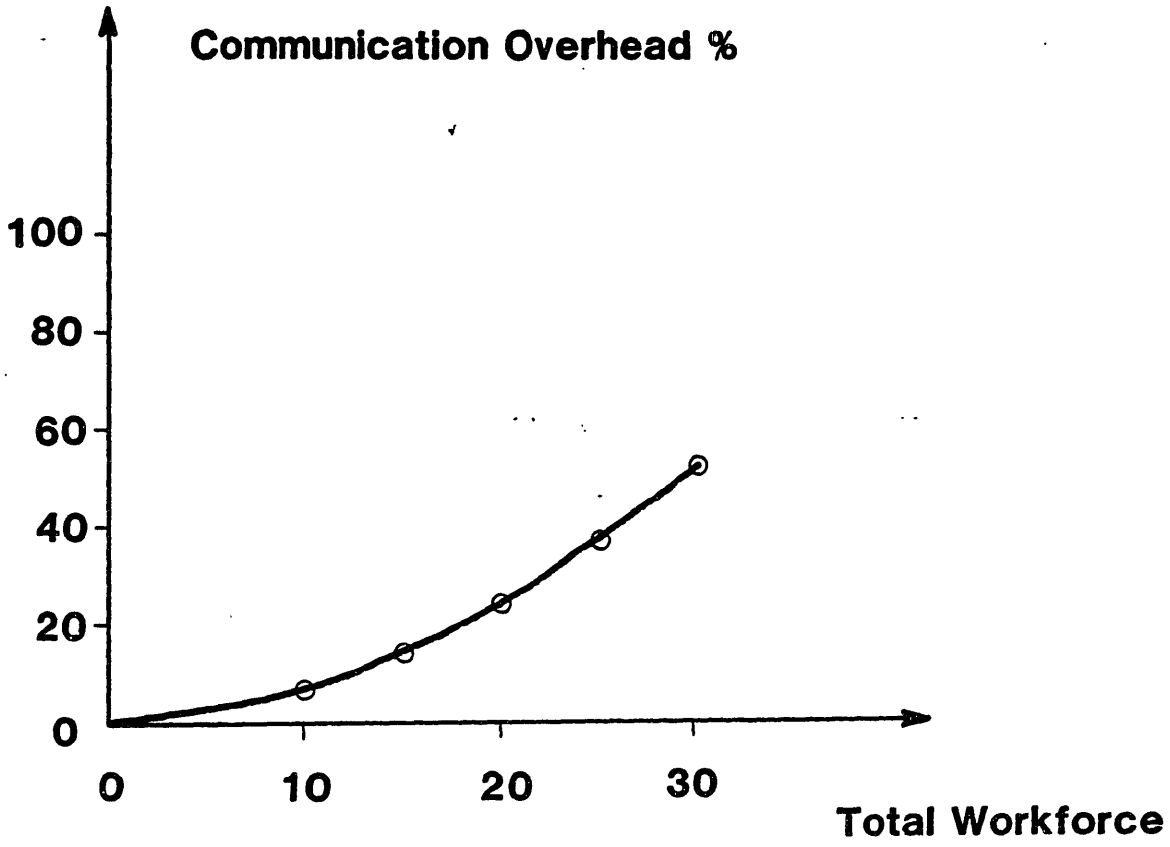


Figure III.15

team member, 50% of these, or 2.4 hours, will be effectively lost due to communication overhead. In other words, the "Multiplier to Productivity Due to Motivation and Communication Losses" would be 0.6X0.5=0.3. Which means that "Software Development Productivity" would be 30% of the value of "Potential Productivity." For example, if the latter is 1 Task/Man-Day, then "Software Development Productivity" would be 0.3 Tasks/Man-Day (after accounting for motivation and communication losses).

(C) Quality Assurance and Rework:

> The development of software systems involves a series of production activities where the opportunities for interjection of human fallibilities are enormous. Errors may begin to occur at the very inception of the process where the objectives of the software system may be erroneously or imperfectly specified, as well as during the later design and development stages where these objectives are mechanized. The basic quality factor for software is that it performs its functions in the manner that was intended by its architects. In order to achieve this quality, the final product must contain a minimum of mistakes in implementing their intentions as well as being void of misconception about the intentions themselves. Because of human inability to perform with perfection, software development is accompanied by a quality assurance activity (Deutsch, 1979).

Software quality assurance is approached by two distinct and complementary methodologies. The first is that of assuring that the quality is initially built into the product. This involves emphasis on the early generation of a coherent, complete, unambiguous, and nonconflicting set of requirements. Then as the product is designed and coded, review and testing of the product, the second quality tool, are encountered (Deutsch, 1979).

In this section we will discuss the generation, detection, and correction of errors during the development phase. As we indicated in Section III.3. (on "Model Boundary") the development phase includes both the design and coding activities, but excludes the requirements phase. It was also indicated then, that we will be assuming that

software design commences (within the model's boundary) at the "successful completion" of a software requirements review (outside the model's boundary), and that there would be no subsequent changes or modifications in the system's requirements.

In this section, therefore, our concern is with the generation of design and coding errors, and with the second quality tool above, namely, the review and testing of the product.

Errors come in many different, "flavors." Summarized below are what Nelson (1974) delineated and described as the most prominent software design and coding errors:

* Misinterpretation of specifications
* Errors in developing the logic to solve the problem
* Algorithm approximations that may provide insufficient accuracy or erroneous results for certain input variables
* Data structure defects either in the data structure design specification or in the implementation of the specification
* Singular or critical input values to a formula that may yield an unexpected result not accounted for in the program code
* Misinterpretation of language constructions by the

programmer

In a system dynamics model such as ours, it is quite feasible, from a technical point of view, to disaggregate a variable such as errors into different error types. However, it is not always necessary or useful.

> There are two (and only two) considerations for reformulating a level (variable) as a sequence of two or more levels: policy analysis and model behavior. First, is the disaggregation required in order for the model to be able to address particular policy issues? ...
> The second reason for disaggregating a level involves the dynamics of the system. Does the disaggregation of a level into two or more levels have the potential to change significantly the behavior of the model? ... The final arbiter should be model-based policy analysis. If the change in behavior has the potential to alter policy conclusions, then the disaggregation is essential (Richardson and Pugh, 1981).

Since our model's policy focus is on the managerial-type policies of software development, as opposed to say the technical issues of software reliability, an explicit disaggregation of errors into more than one type is, on the basis of the policy analysis criterion, clearly unnecessary. On the other hand, there are significant behavioral differences among error types that had to be accounted for. For example, findings in the software engineering literature indicate that errors are generated at different rates at different points in the life cycle e.g., design errors, in the earlier design phase, are generated at a higher rate than are coding errors (Martin, 1982). Such a factor is obviously

of dynamic significance. For example, it could have a direct bearing on the allocation of the manpower resource.

Such differences will be implicitly captured in the model. That is, while errors will be formulated as a single type, "Errors," the generation, detection, and correction characteristics of errors will be allowed to vary throughout the development life cycle. For example "Errors" will be generated at a higher rate in the earlier portions of the life cycle (as design errors do) and they will, on the average, be "harder" to detect and correct (as design errors are).

Figure III.16. depicts how the generation, detection, and correction of errors are formulated in the model.

What factors affect the "Error Generation Rate" in a software project? There are two sets of factors. The first set includes: organizational factors e.g., the use of structured techniques (Alberts, 1976), the quality of the staff (Belford et al, 1977), ... etc., and project-type factors (Shooman, 1983) e.g., complexity, size of system (small, medium, or large), language, ... etc. Notice that even though such factors can differ from organization to organization and from project to project, they do, however, remain invariant during the life of a single project. The cumulative effect of all such factors can, therefore, be

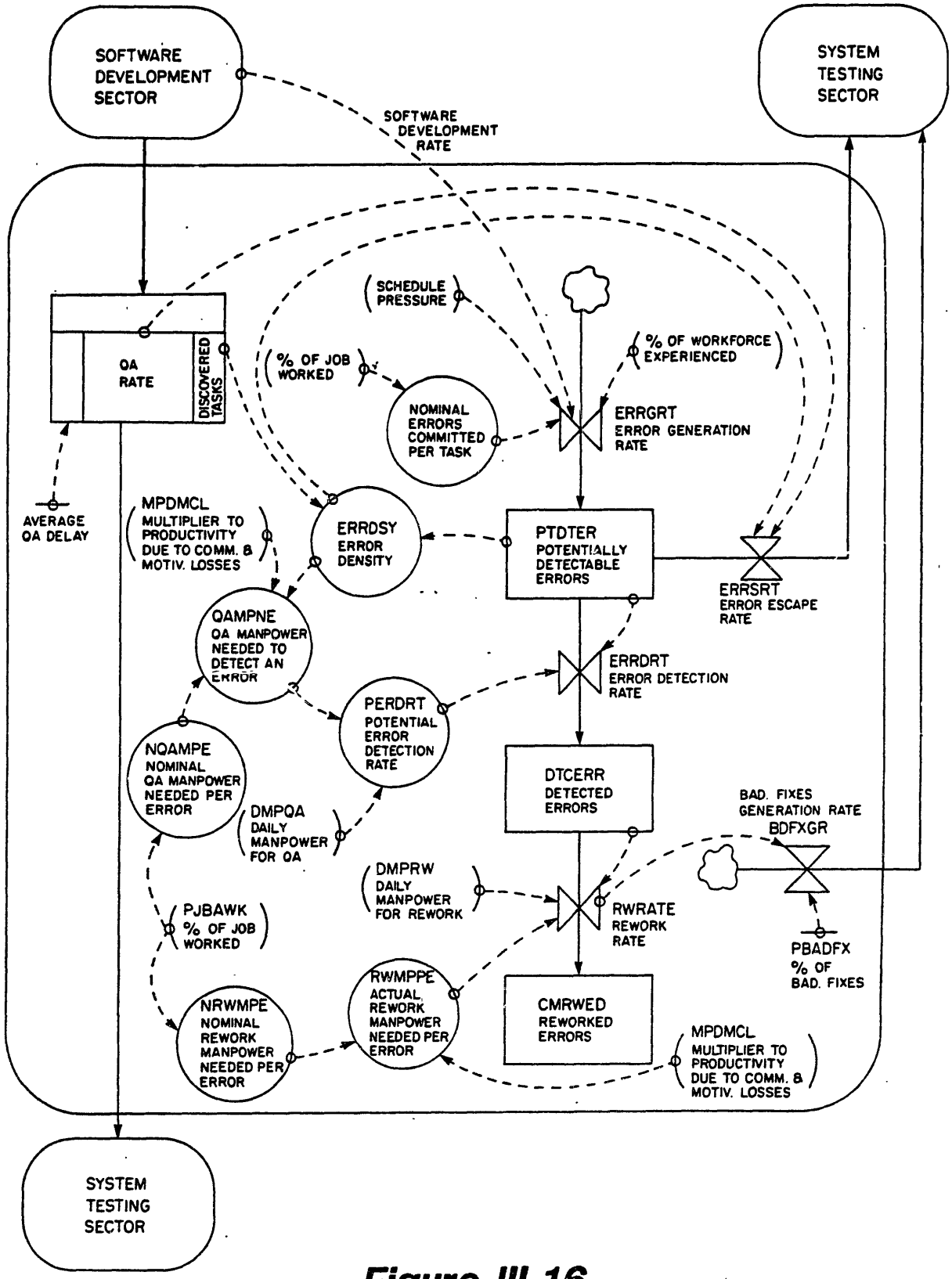


Figure III.16

captured in the model in the form of a single nominal variable, namely, the "Nominal Number of Errors Committed per Task." The nominal error generation rate would then simply be the product of the "Software Development Rate," i.e., how much tasks are developed per unit of time, and that "Nominal Number of Errors Committed per Task." However, since this single nominal variable is modeling the generation of different error types (within the single project that is within a particular organization) it is not formulated as a constant number, but rather as a variable that changes over the project's life.

The formulation of the "Nominal Number of Errors Committed per Task" is, therefore, serving two purposes: First, its shape over the project's life reflects our own modeling assumptions about the relative generation rates of different error types throughout the life of a project. These assumptions, as all others in the model, are expected to apply to all project situations to which the model is applied. Hence, this shape will always remain the same, even when modeling different project situations. The second purpose of the formulation, namely, its absolute value, reflects the different error generation characteristics of different project situations (i.e., the software product's characteristics as well as those of the organization in which it is developed). This, obviously, would generally change when modeling different projects.

The formulation of the "Nominal Number of Errors Committed per Task" used in the base model is shown in Figure III.17. Notice that the number of errors is defined in terms of KDSI i.e., "thousand delivered source instructions" rather than "Tasks." Both definitions are, of course, equivalent since a "Task" is itself defined in terms of DSI. However, it is more convenient to represent error generation in terms of KDSI since most published data on error rates are in terms of KDSI.

The error rates range in value from 25 errors/KDSI to 12.5 errors/KDSI, with an average value for the project of approximately 19 errors/KDSI. [Published error rates in the literature include: 10-20 errors/KDSI in (Thayer et al, 1978), 15-25 errors/KDSI in (Boehm, 1981), 30-35 errors/KDSI in (Jones, 1978).]

As we mentioned above, the shape of the curve over the project's life reflects the relative generation rates of design-type errors versus coding-type errors. Thus, before we can specify the shape of the curve we need first to delineate design versus coding activities within the development life cycle. We will assume in the model that the development phase will be equally divided between design (including architectural and detailed design) and coding activities. [This approximates data reported by (Boehm, 1981), (Gaffnery, 1982), and (Zelkowitz, 1978).] The diagram

at the bottom of Figure III.16. is meant to indicate that the transition between the two activities is not abrupt i.e., there will be a period over which both activities will overlap (McKeen, 1981) (Thibodeau and Dodson, 1980).

Estimates for relative generation rates of design versus coding errors were provided by several authors. For example,

Design : Coding Errors	Reference
3.8 : 1	(Martin, 1982)
2.0 : 1	(Alberts, 1976)
1.8 : 1	(Jones, 1981)
1.7 : 1	(Boehm, 1981)
1.6 : 1	(Thayer et al, 1978)

As shown in Figure III.17., the ratio assumed in the model achieves a maximum value of 2:1 i.e., at the beginning of design the nominal number of errors committed is 25 errors/KDSI, while towards the end of coding it drops to 12.5 errors/KDSI. The average rates for the design and coding phases are approximately 23 and 14.5 errors/KDSI respectively i.e., a 1.6:1 ratio.

The formulation of the nominal error generation rate captures, as we mentioned above, the cumulative effect of one set of factors effecting error generation, namely, the organizational and project-type factors. Such factors remain invariant during the life of a single project. There is a second set of factors, however, which do play a dynamic role during software development. These include the workforce-mix and schedule pressures.

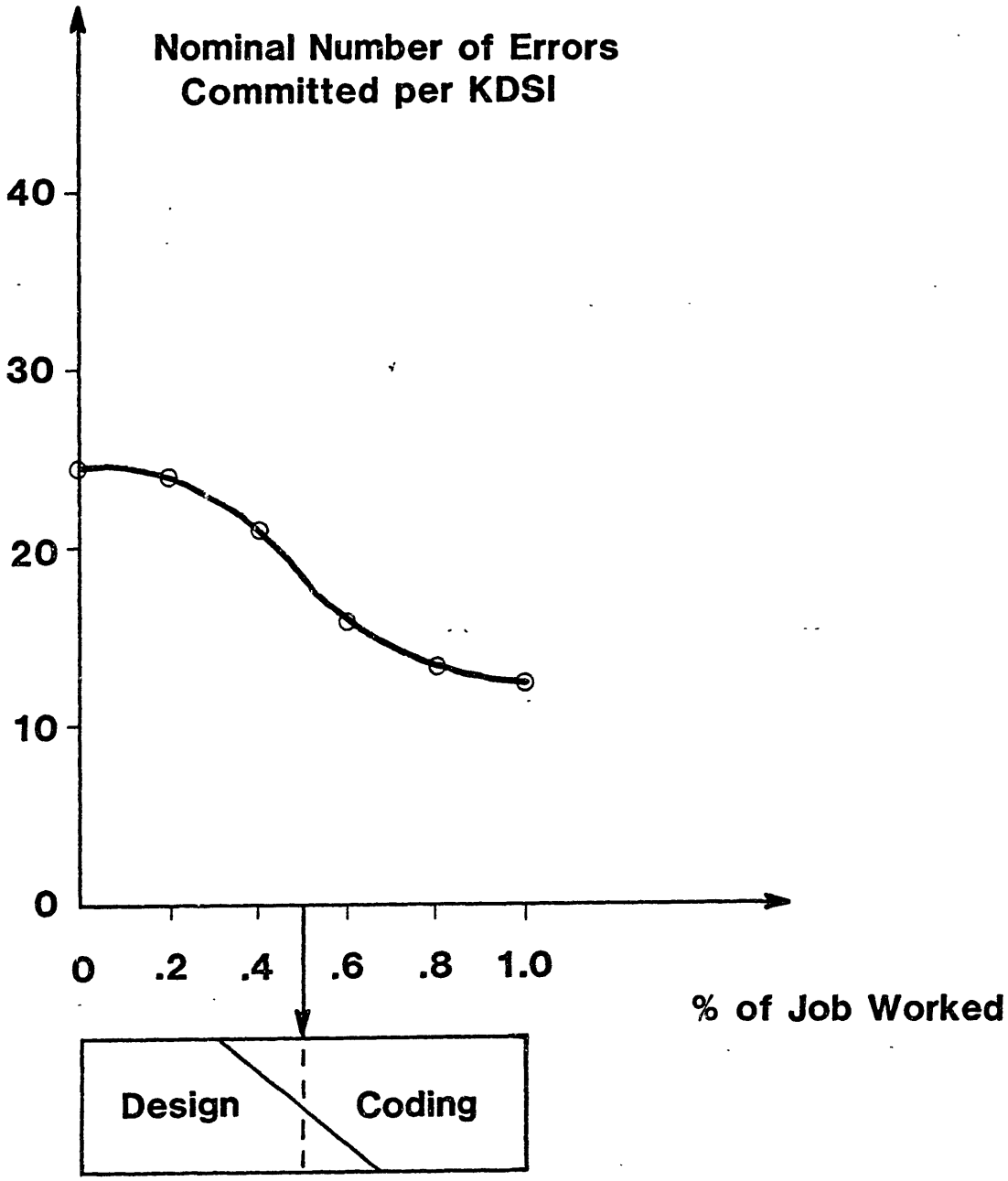


Figure III.17

As was stated in the discussion on Human Resource Management, the workforce in the model is disaggregated into two types of employees, newly hired and experienced. It was also indicated that new hires pass through an "Orientation Phase" during which they are less than fully productive. The orientation process brings them "up to speed" through training that covers both the social as well as the technical environments of the project. For example, on the technical side, newly hired project numbers "often require considerable training to become familiar with an organization's unique mix of hardware, software packages, programming techniques, project methodologies, and so on" (Winrow, 1982).

While not yet fully trained (during this orientation period) newly hired employees are, not only less productive on the average, but also more error-prone than their experienced counter-parts (Endres, 1975) (Myers, 1976). We will assume in the model that a newly hired employee is twice as error-prone as an experienced employee would be (Chan, 20) (Nichlos, 25). To model the effect of this factor on error generation we formulate the "Multiplier to Error Generation due to Workforce Mix" as a function of the "% of Workforce that is Experienced." When the workforce value is comprised of only experienced staff, the value of the multiplier is set to 1 i.e., it would have a neutral effect on the nominal error generation rate. In other words, what we are defining to be nominal, is defined with respect to the average error

generation rate of the experienced-type employee. And as the fraction of new hires increases, the multiplier increases in a linear fashion, as shown in Figure III.18., until it attains a maximum value of 2 , if the workforce is comprised of only new hires.

The second factor that can drive the error generation up is schedule pressure (Putman and Fitzsimmons, 1979) (Mills, 1983) (Radice, 1982) (James, 1) (Riccardi, 5) (Doyle, 7) (Nichols, 18) (Sheldon, 19) (Chan, 20).

> 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).

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

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

Another explanation was provided by Thibodeau and Dodson (1980). They suggest that schedule pressures often result in the "Overlapping of activities that would have been accomplished better sequentially." and this can

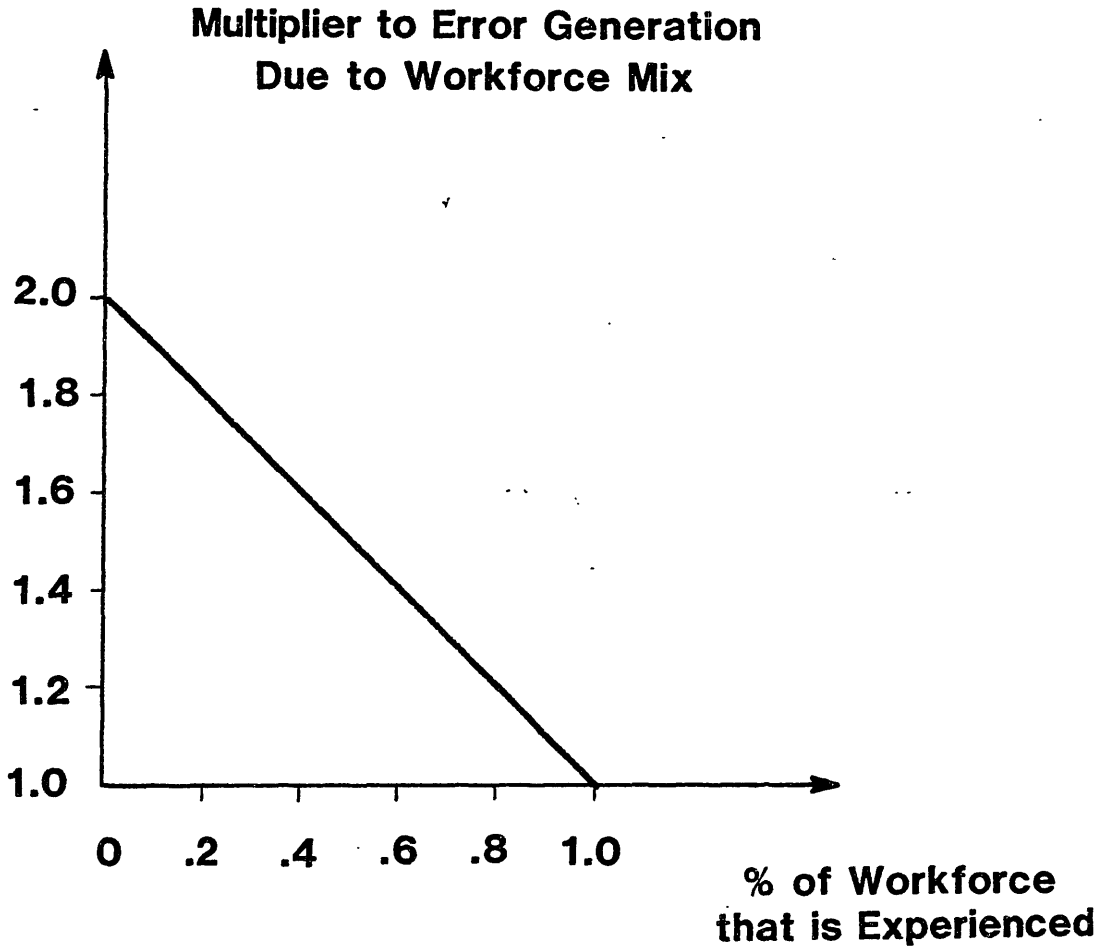


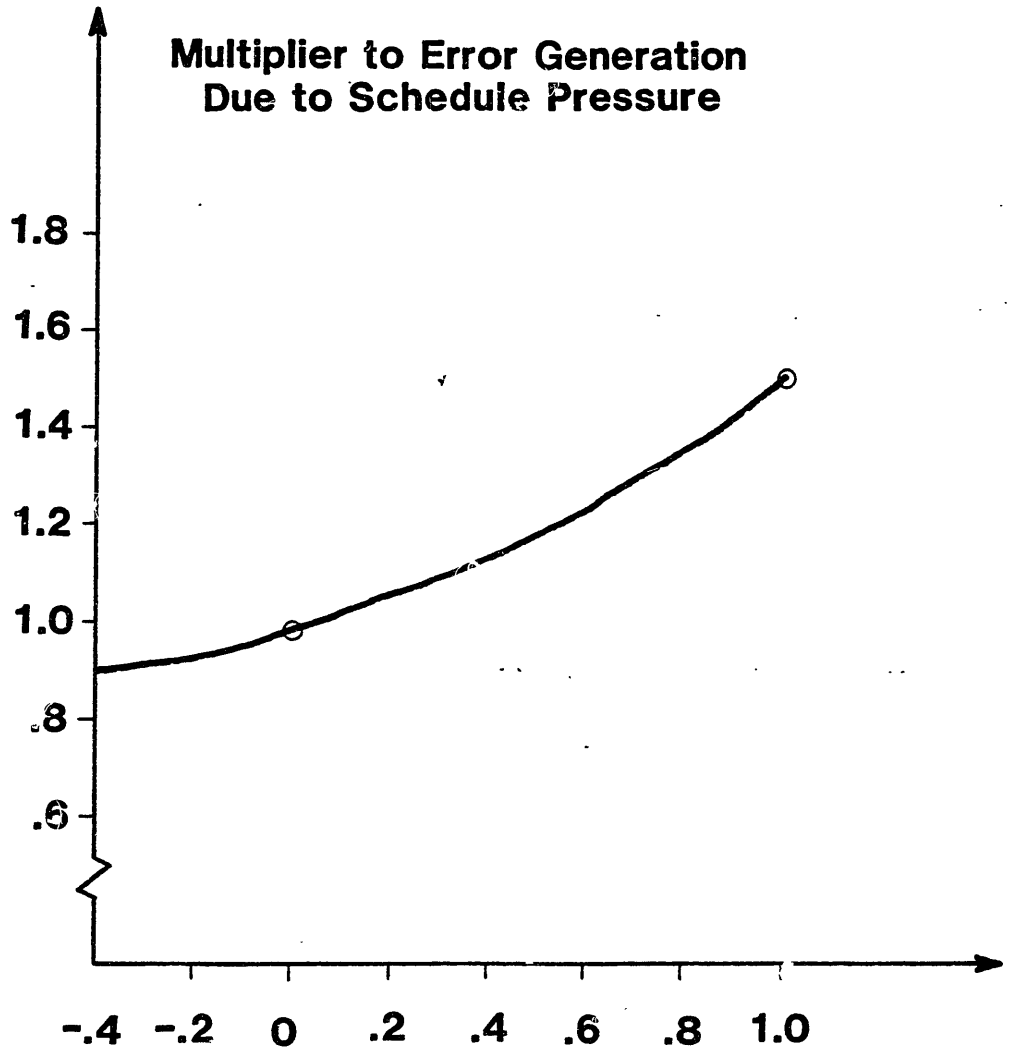
Figure III.18

significantly increase the chance of errors. For example,

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

The effect of schedule pressure on error generation is formulated in the model as shown in Figure III.19. Under nominal conditions there would be no schedule pressures, and the multiplier assumes a value of 1. As schedule pressures increase, the multiplier increases exponentially leading to higher error-generation rates. As shown in the Figure, error-generation can increase by as much as 50% under severe schedule pressures. Notice also, that we are assuming that errors will be generated below the nominal rate under the "relaxed" conditions of negative schedule pressures.

Thus, as software tasks are developed, errors are committed within those tasks. Errors within a developed task remain as "Potentially Detectable Errors" until the task is reviewed and tested, at which point some of the errors do get detected, and those are then reworked. Usually, however, not all errors will be detected, some will "escape" and pass undetected into the subsequent phases of software development. In the next section we will see how those



Where,

$$= \frac{\text{TMDPSN} - \text{MDRM}}{\text{MDRM}}$$

TMDPSN=Total Man-Days Perceived Still Needed

MDRM=Man-Days Remaining

Figure III.19

errors are eventually "caught," albeit at a relatively high cost.

The detection of errors is the objective of the Quality Assurance (QA) activities. Quality Assurance is defined in Pressman (1982) as:

> (A set of activities) performed in conjunction with the (the development of) a software product to guarantee the product meets the specified standards. These activities reduce doubts and risks about the performance of the product in the target environment.

Several techniques are used including walkthroughs, reviews, inspections, code reading (a process where code logic and code format is scrutinized by a programmer other than the original designer), and integration testing (Jones, 1982) (Daly, 1977). Not included in this activity is module or unit testing, which is commonly considered to be part of the coding process (McKeen, 1979).

The "QA Rate," of Figure III.16., has a non-characteristic type of a formulation, namely, that of a third order delay. The "characteristic" way to formulate a rate of doing something, e.g., the rate of developing software or reworking errors, is as a product of the effort allocated and its productivity. However, what we found, and what the third order delay formulation actualizes, is that the QA Rate is independent of the QA effort and its productivity! What we found happening [based on discussions

with (Gage, 4) (Landolfi, 13) (Chan, 14) (Lombardi, 16) (Nichols, 18)] is this: QA effort is planned and allocated, usually in the form of a fixed schedule of periodic group-type functions (Mitchell, 1980). For example, a 2-hour walkthrough for the 5 members of team (A) is scheduled for every Friday. During these periodic "QA Windows," all tasks developed since the previous one are supposed to be processed. And what we were surprised to find was that, in an almost perfect realization of Parkinson's Law, irrespective of how much tasks need to be processed within the specified "QA Window" they almost always do. No backlogs, therefore, develop in the QA pipeline. Even when QA activities are relaxed or suspended because of schedule pressure (as we indicated they might in Sector (A)), no backlogs develop. That is, when walkthroughs are suspended for a while on a project, the requirement for a "walkthrough" is also suspended, not postponed (Hart, 1982).

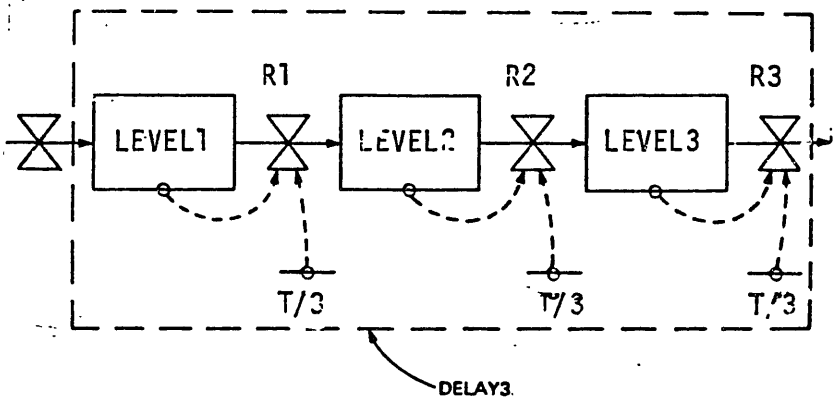
We can propose an explanation for how and why this happens. Since the objective of the QA activity is to detect invisible errors, invisible that is until they are detected, it becomes almost impossible to tell whether the QA job was completly done (i.e., that all those invisible errors were in fact detected). By the same token, it is as difficult to tell that the job has not been completely done (except much later in the life cycle). Under such circumstances it becomes quite easy to rationalize both to oneself and to

management that the QA job that was possible to do, was not insufficient. Furthermore, the QA effort that is possible to expend (i.e., in terms of available time and effort), is usually what is expended and not more (e.g., even if called for due to a larger than expected workload of developed tasks) because there seems to be no significant incentives to do otherwise. Firstly, at the psychological level, there are actually dis-incentives for working harder at QA, since it only "exposes" more of one's mistakes (Weinberg, 1971). And secondly, at the organizational level there are seldom any reward mechanisms in place that promote quality or quality-related activites (Cooper and Fisher, 1979).

The formulation of the "QA Rate" as a third order delay, provides, we feel, a good approximation of the "Parkinsonian" execution of the QA activity as described above. (In Exhibit III.5., we show how a third-order delay looks schematically, how it is formulated mathematically, and how it behaves over time.) That is, software tasks that are developed will always be QAed (or considered QAed) after a certain delay, and which is (assumed to be) independent of the QA effort allocated. In the model, the "Average QA Delay" period is set to 2 weeks (i.e., 10 working days) (Nichols, 25).

However, while the rate at which tasks are QAed (or considered QAed) can proceed under QA policies and procedures independently of the actual QA effort allocated, the

(A) SCHEMATIC



T is the "Average Time Delay"

LEVEL = LEVEL1 + LEVEL2 + LEVEL3

(B) MATHEMATICAL FORMULATION

At any time (t),

$$R_i(t) = LEVEL_i(t)/(T/3)$$

$$\frac{d}{dt}(LEVEL_i(t)) = -R_i(t)$$

$$= -LEVEL_i(t)/(T/3)$$

(C) BEHAVIOR

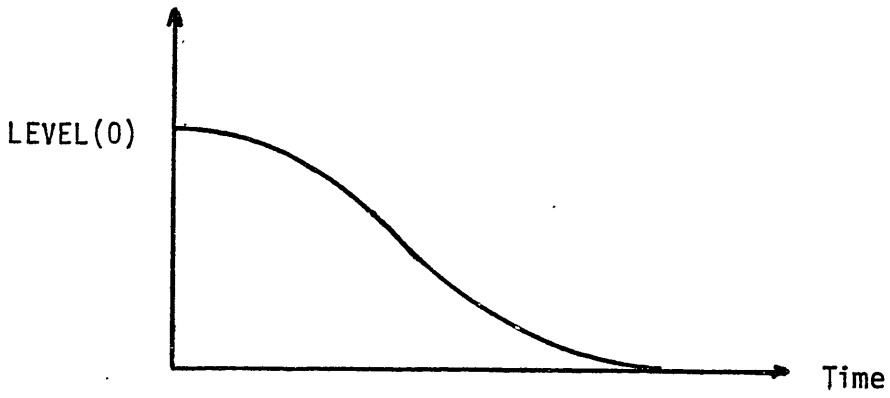


EXHIBIT III.5

effectiveness of QA will, obviously, depend on that effort. That is, the amount of errors detected will be a function of how much QA effort is allocated for error detection.

In the model (see Figure III.16.) we define a variable called "Potential Error Detection Rate." It represents, the maximum number of errors that could be detected at a point in time, and is determined by dividing the value of the QA effort allocated by the value of the QA effort that is needed, on the average, to detect an error. That is, if say 5 man-days are allocated per week to QA, and the "QA Manpower Needed to Detect an Error" is, on the average, 1 man-day, then the "Potential Error Detection Rate" would be 5 errors per week.

What are the determinants of the "QA Manpower Needed to Detect an Error?" First and foremost, it is a function of error-type i.e., whether an error is a design or a coding error. Thus, even if a project proceeds under some invariant set of nominal conditions, the QA manpower that would be needed, on the average to detect an error would change simply because the errors to be detected change from design-type errors to coding-type errors.

The value of the QA effort needed per error as a function of the project's phase and hence of error-type i.e., design errors versus coding errors, are shown in Figure

III.20. Design-type errors are not only generated at a higher rate (as we saw in Figure III.16), they are also, as Figure III.20. indicates, more costly to detect (Myers, 1976) (Alberts, 1976) (Boehm, 1975). Alberts (1976) estimates that design errors are 2.5 times more costly (i.e., to detect and correct). In the formulation of Figure III.20., we are assuming that, on the average, a design error is 1.6 as costly to detect as a coding error. Furthermore, in terms of absolute values, the average detection effort per error is 0.3 man-days. Thus, on the average it would take approximately 2.4 man-hours (30% of an 8-hour man-day) to detect an error. In the case of walkthroughs and inspections, this effort would include, not only the effort expended during the walkthrough/inspection itself, but also the effort expended in preparation for it (e.g., reviewing documentation and gaining familiarity with product). Estimates in the literature for the error detection effort per error include: 3 man-hours (Mitchell, 1980), 2.36 man-hours (Shooman, 1983), and 0.5-1.25 man-hours (Shneiderman, 1980).

The actual QA manpower needed to detect an error, in addition to being a function of error-type, must also depend on the efficiency of how people work. In our discussion on productivity we indicated that a full-time employee's work day does not translate into an 8 man-hour input to the project. Man-hours are lost on communication and other

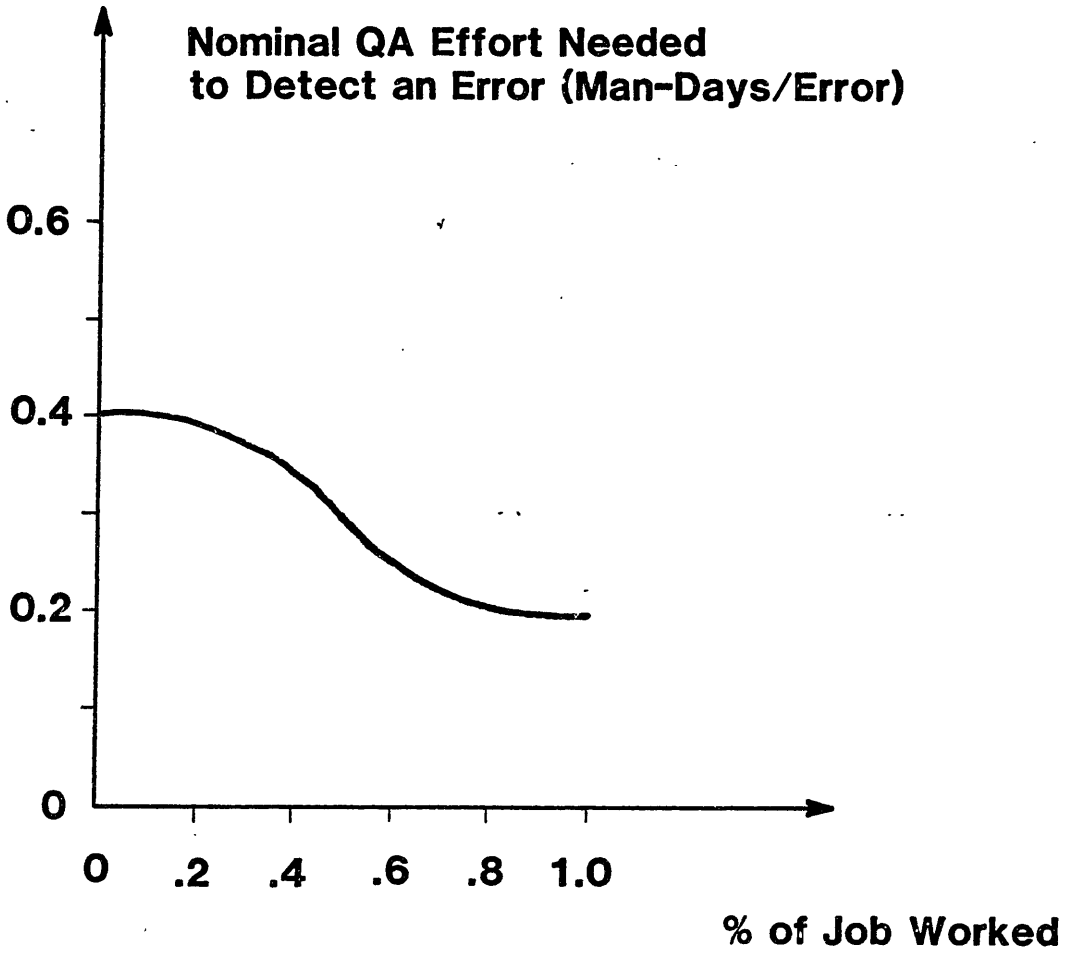


Figure III.20

non-project activities (e.g., personal business). These two types of losses are captured in the "Multiplier to Productivity Due to Communication and Motivation Losses," which simply represents the average productive fraction of a man-day. In other words, if the communication and motivation losses amount to a 4 man-hour loss per day (for the average employee) i.e., half of the nominal 8 man-hour value, then the value of the multiplier would be a 0.5. Under such circumstance, the actual QA manpower needed to detect an error becomes twice what is nominally needed. That is, if a design error requires, under nominal conditions (i.e., under conditions of no losses), 0.4 man-days to be detected, it would actually require (under the above conditions) 0.4 X 2 = 0.8 man-days.

Finally, evidence suggests that "In any sizable program, it is impossible to remove all errors" (Shooman, 1983). Thus, even when generous effort allocations are made to QA, it would still be unlikely that all errors will be detected (Boehm, 1981). One reason, for example, is that "... some errors manifest themselves, and can be exhibited only after system integration" (Shooman, 1983). At any point in time, one could, therefore, view the collection of "Potentially Detectable Errors" as constituting a hierarchy of errors, in which some are more subtle, and therefore more expensive to detect than others. Empirical results reported by Basili and Weiss (1982), suggest that the distribution is pyramid like,

with the majority of errors requiring approximately a few hours to detect, a few errors requiring approximately a day to detect, and still fewer errors requiring more than a day to detect. Notice that the results show that those few subtle errors are an order of magnitude more expensive to detect.

We will assume in the model, that as QA activities are performed, the more obvious errors will be detected first. And as these are detected, it then becomes more and more expensive to uncover the remaining more subtle (although less predominant) errors. This is realized in the model through the formulation of the "Multiplier to detection Effort due to Error Density," shown in Figure III.21. At moderate to large error densities, the multiplier assumes a neutral value of 1. But as those "obvious" errors are all detected, and a few "subtle" errors remain, the multiplier increases in an exponential fashion, such that at a density level of 2-4 (subtle) errors per KDSI, it becomes an order of magnitude more expensive to detect an error.

To recapitulate, the "QA Manpower Needed to Detect an Error" is a function of error-type, work efficiency and error density. As the value of this needed effort increases, e.g., due to a decrease in error density, the number of errors that can be detected, at some level of QA effort, decreases. At any point in time, the "Potential Error Detection Rate"

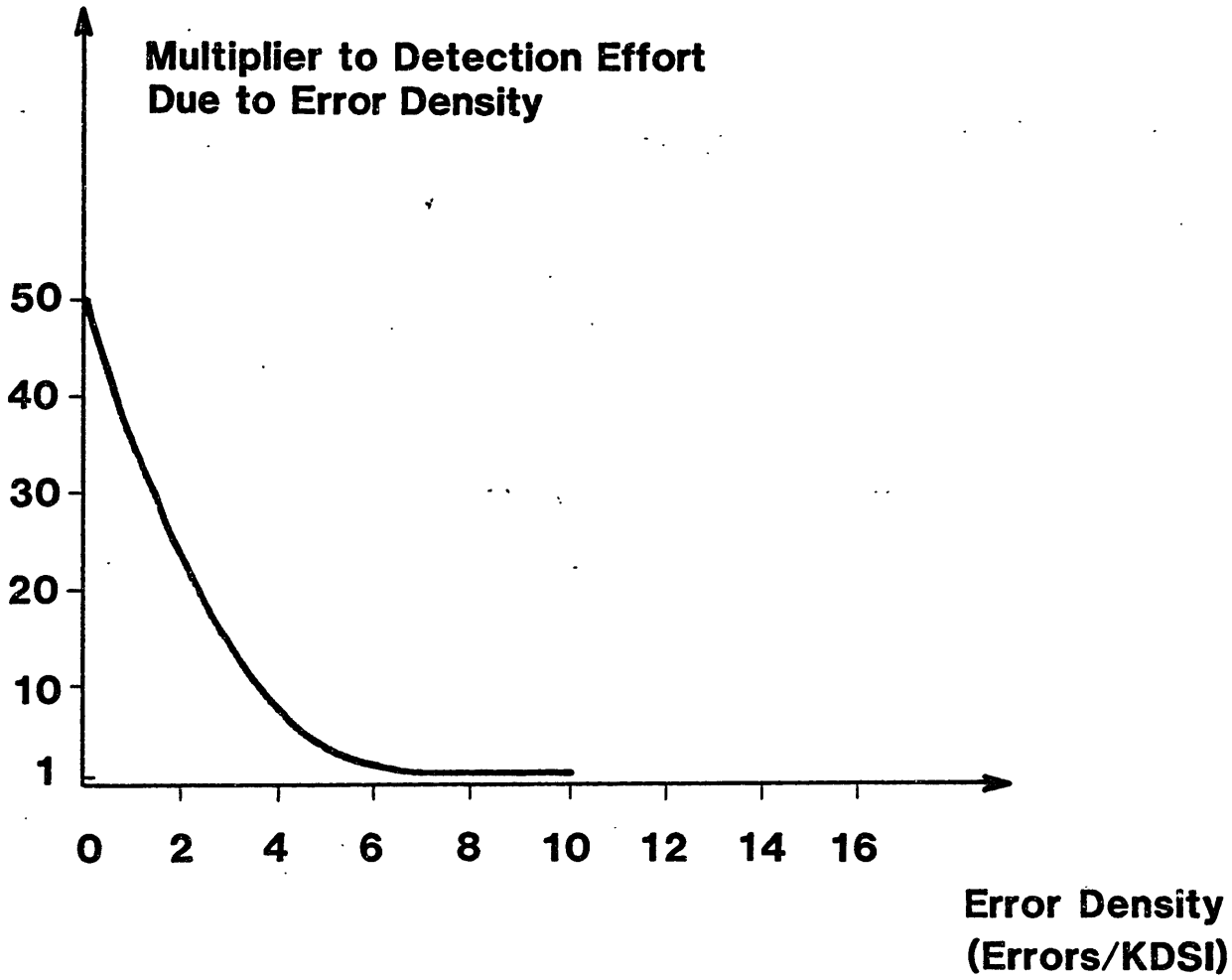


Figure III.21

(determined by dividing the value of the QA effort allocated by the value of the "QA Manpower Needed to Detect an Error"), represents the maximum possible number of errors that could be detected. Because manpower allocations to QA are often "modest," this maximum value is seldom large enough to secure the detection of all errors generated. And even when effort is allocated generously to QA, a few subtle errors will be so prohibitively expensive to detect, that whatever the effort allocated, it will not be quite enough to detect all errors. As a result, as shown in Figure III.16., some errors will "escape" and pass undetected into the subsequent phases of software development. In the next section we will deal with those errors, and show how they are eventually "caught."

On the other hand, those errors that do get detected through QA activites, are then reworked. The rework rate is a function of how much effort is allocated to rework activities, and the rework manpower needed per error. For example, if the project members commit 10 man-days per week to rework detected errors, and the "Actual Rework Manpower Needed per Error" is, on the average, 1 man-day, then errors will be reworked at the rate of 10 per week.

The "Actual Rework Manpower Needed per Error" has two components. The first is the "Nominal Rework Manpower Needed per Error." As in the case of error detection, this nominal component is a function of error-type i.e., design versus

coding errors.

The values of the nominal rework effort needed per error as a function of the project's phase, and hence of error-type, are shown in Figure III.22. Design-type errors, in addition to both being generated at a higher rate and being more costly to detect, are also more costly to rework (Myers, 1976) (Alberts, 1976) (Boehm, McClean, and Urfrig, 1975). As the formulation of Figure III.22. indicates, we are assuming that, on the average, a design error is approximately 1.5 more costly to correct than a coding error. Under nominal conditions, a design error would require, on the average, 0.54 man-days to be corrected, while the average correction effort for a coding-type error is assumed to be 0.36 man-days. For the nominal 8-hour working day, these averages translate, into 4.3 man-hours/error and 2.9 man-hours/error, respectively. These values were chosen on the basis of the empirical results reported in (Weiss, 1979) and (Basili and Weiss, 1981), which suggest that the average rework effort (for all errors) is in the range of 0.25 to 1.0 man-days per error.

The actual rework man-power that would be needed to correct an error, in addition to being a function of error-type, must also depend on the efficiency of how people work. That is, we need to account for the communication and motivation losses incurred. For example, if the "Multiplier

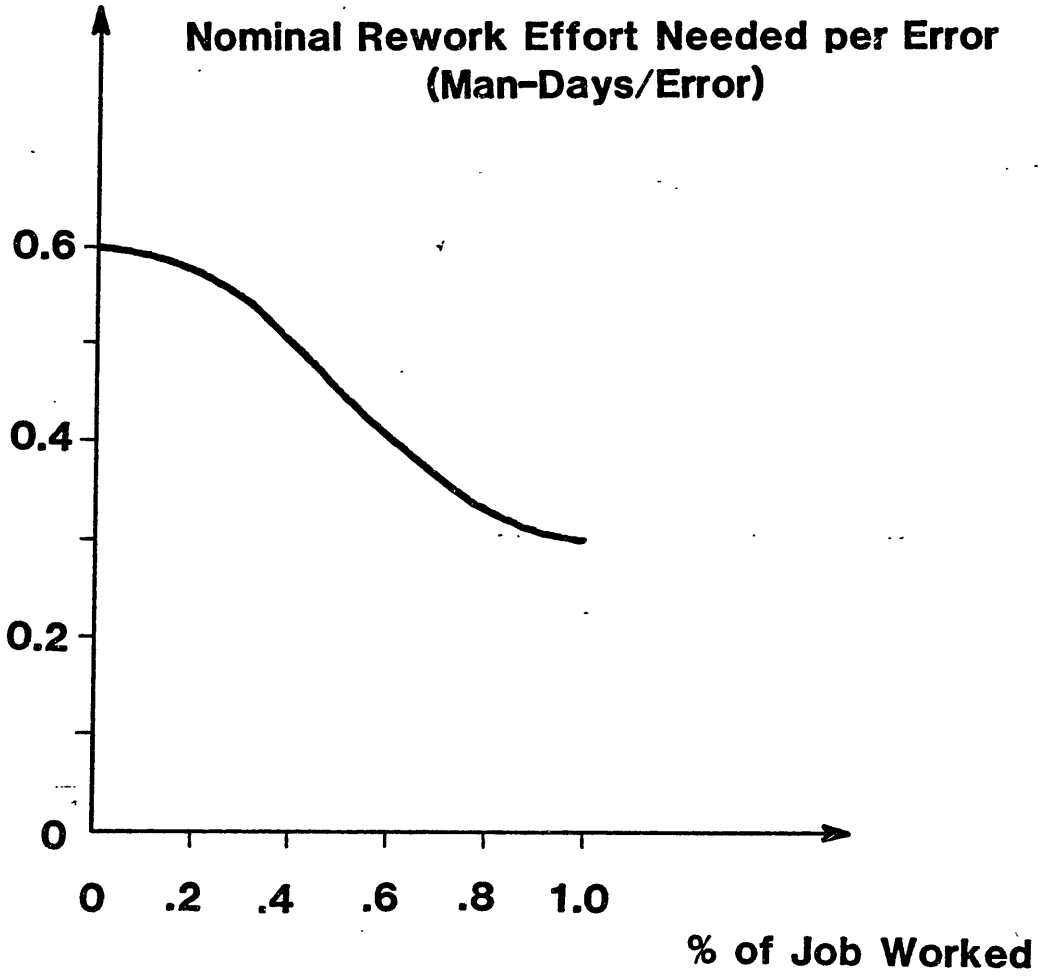


Figure III.22

to Productivity due to Communication and Motivation Losses," which represents the average productive fraction of a man-day, is 0.5, then the actual rework manpower needed to correct an error becomes twice what is nominally needed. A design error that would have required under nominal conditions (i.e., under conditions of no losses), 0.5 man-days to be corrected, would actually require (under the above condition) 0.5 X 2 = 1 man-day.

To recapitulate, as errors are detected through the QA activities, they are reworked. The rate at which errors are reworked is a function of the manpower committed to the rework activity and the rework effort needed per error. The "Actual Rework Manpower Needed per Error" is, in turn, a function of two things, error-type (i.e., design versus coding errors) and work efficiency.

The reworking of software errors is not, itself, an errorless activity:

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

The problem of "bad-fixes" is widely documented in the literature (e.g., (Jones, 1978) (Shooman, 1983), (Myers, 1976), (Endres, 1975), (Fagan, 1976), and (Thayer et al, 1978)). Shooman and Natarajan (1977), suggested some of the

ways in which bad-fixes may be generated:

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

Thus, as detected errors are reworked, some fraction of the corrections will be bad-fixes. Unfortunately, there are no published data on how large that fraction is. However, there are results that indicate that bad-fixes constitute 6.5 - 10% of all errors caught at the system testing stage (Jones, 1981) (Fries, 1977). The balance of the errors is comprised of those errors that escape detection, through QA, during development. If we assume that 50-60% of errors are detected and reworked during development, and that most of the remaining errors together with bad-fixes are later detected at the system testing phase, then the above findings on bad-fixes imply that between 4.5-11% of corrections will be bad-fixes. The "% Bad-Fixes" is, therefore, set in the model to 7.5%.

The detection and correction of bad-fixes as well a those errors that escape QA detection, is the topic of the

next section.

(D) System Testing:

We will assume that undetected errors i.e., those that QA activities (e.g., walkthrough, inspections, code reading, ... etc.) fail to detect while the software is being designed and coded, as well as those bad fixes created as a result of faulty rework, will all remain undetected until the system testing phase. Further, we will assume that all such errors will get detected and corrected at the system testing phase. Thus, even though in practice some errors often remain in a software product after system testing is completed (i.e., as the product becomes operational), e.g., because system testing activities fail to detect them, or they result from bad fixes at the system testing phase, all such errors will be excluded from our formulation. The primary reason for their exclusion is that the generation, detection, and correction of these errors are all issues of maintenance of the operational system, which are, as we previously stated, beyond the boundary of our model and thus the focus of this study.

The second justification for their exclusion, is that errors that escape detection at the system testing phase are generally a "small" fraction of all the errors handled at that phase (Deutsch, 1979). This assertion might sound

surprising to many, since it is common to assume that the maintenance activity is as costly as it is primarily because of the costs incurred in handling such "lingering" errors. What empirical results have shown, however, is that corrections of such errors consumes only a relatively small portion of the software maintenance activity (Lientz and Swanson, 1978). The major portion of the software maintenance effort is, instead, devoted to software updates (e.g., enhancements for users, adaptation to new data or hardware, ... etc.) (Parikh and Zvegintzov, 1983).

The System Testing Sector is shown in Figure III.23. As shown in the figure, this sector models two sets of processes, namely, the growth processes of the undetected error populations and the processes of system testing, i.e., the detection and correction of those errors.

The population of undetected errors is comprised, as we said, of errors that escape the detection of the QA actions as well as those bad fixes created as a result of faulty rework. This group of errors does not remain dormant awaiting detection and correction at the system testing phase. They, instead, lead an "active existence" reproducing more and more errors in the system. For example, a design error that remains undetected until the system testing phase often instigates further errors in the code, user and maintenance manuals, training material, ... etc., (Boehm,

1981).

In a study by Shooman reported in McClure (1981), it was determined that detecting and correcting a design error during the design phase (i.e., through the QA activities) is one-tenth the effort that would be needed to detect and correct it later during the system testing phase because of this additional inventory of specifications, code, user and maintenance manuals, ... etc., that would require correction in the later case. This 10:1 ratio was also supported by data in Boehm (1981), but only for larger projects. For smaller projects, the escalation in cost-to-fix was in the range of 4:1, because, Boehm argued, "The smaller size meant that there was a relatively smaller inventory of items to fix in later phases."

But, besides such static estimates on cost-to-fix escalations at different points in the software life cycle, no data are available in the literature to describe the dynamics of these "error-reproduction" processes. That is, even though we do know that an undetected design error reproduces enough errors in code, documentation, ... etc., to become 4 to 10 times more expensive to fix at the system testing phase, we still do not have the data that explain exactly how and when these reproduction processes occur.

When the dynamic relationships are not well understood

(that is, when theory is not well developed), as it is in this case, then "the best one can do is attempt to imitate the change process itself in the hope of learning more about such relationships. Thus the model becomes an aid to theory development" (Schultz and Sullivan, 1972). Our "proposed theory" of the error reproduction process is depicted in Figure III.23.

As shown in the figure, we are assuming that errors that escape QA detection, together with those generated due to faulty rework, will develop into either "Active Errors" i.e., active in reproducing more errors, or "Passive Errors." Because design specs are the blue prints of the system's code, any errors in design will get translated into coding errors. Thus, all undetected design errors should be of the active type. As development moves into the coding stage, a mixture of active and passive errors would be expected. If we assume, for example, that the system is coded in a top-down fashion, then in the early parts of the coding stage most of the errors committed (i.e., in the high-level modules) would be of the active type. As development proceeds to the lower level modules, the reverse should be true, since the errors become more and more localized in nature. These assumptions on how the mixture of active and passive errors changes over the project's life are realized in the model through the formulation of the variable "% Active Errors" shown in Figure III.24.

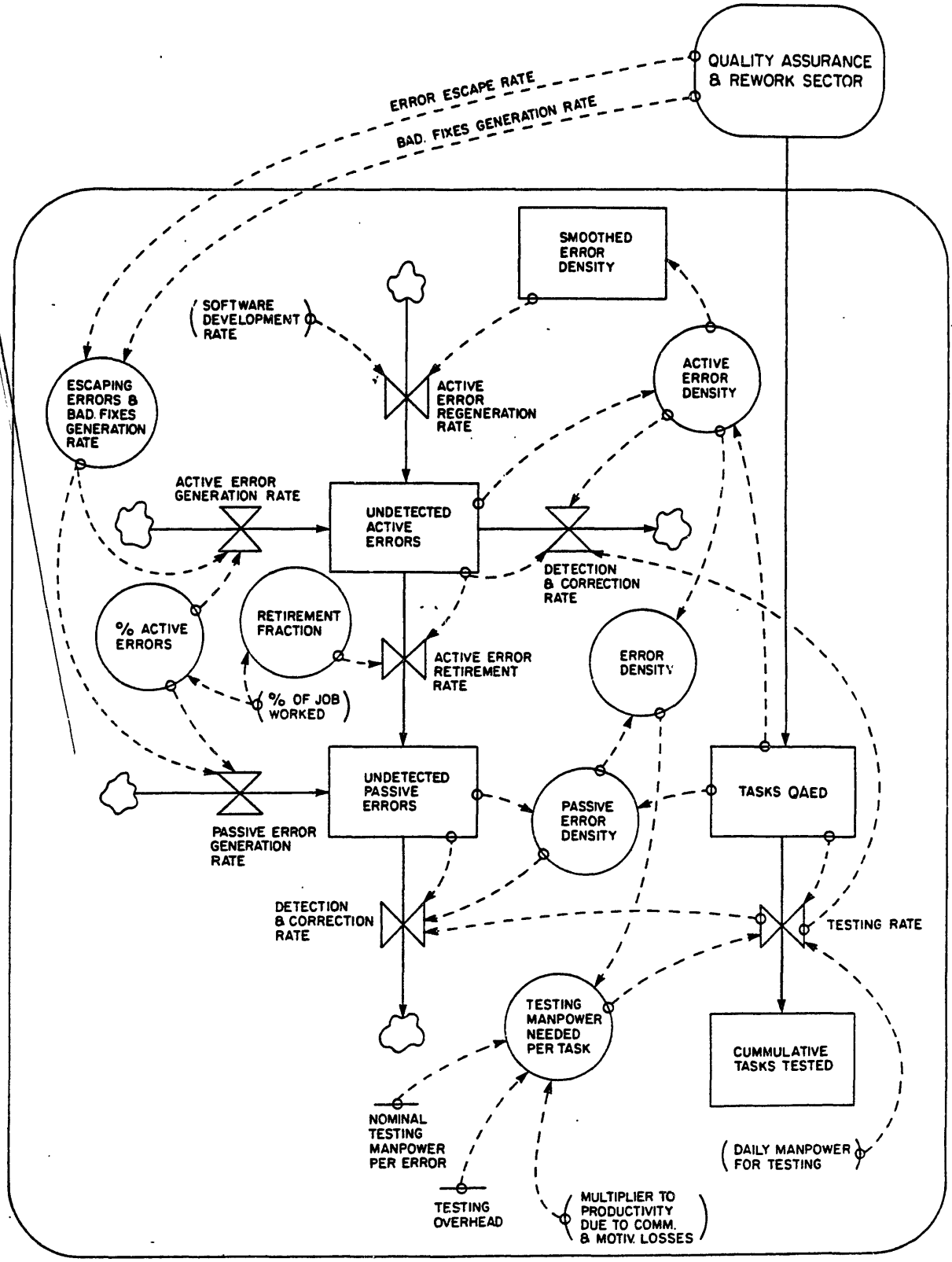


Figure III.23

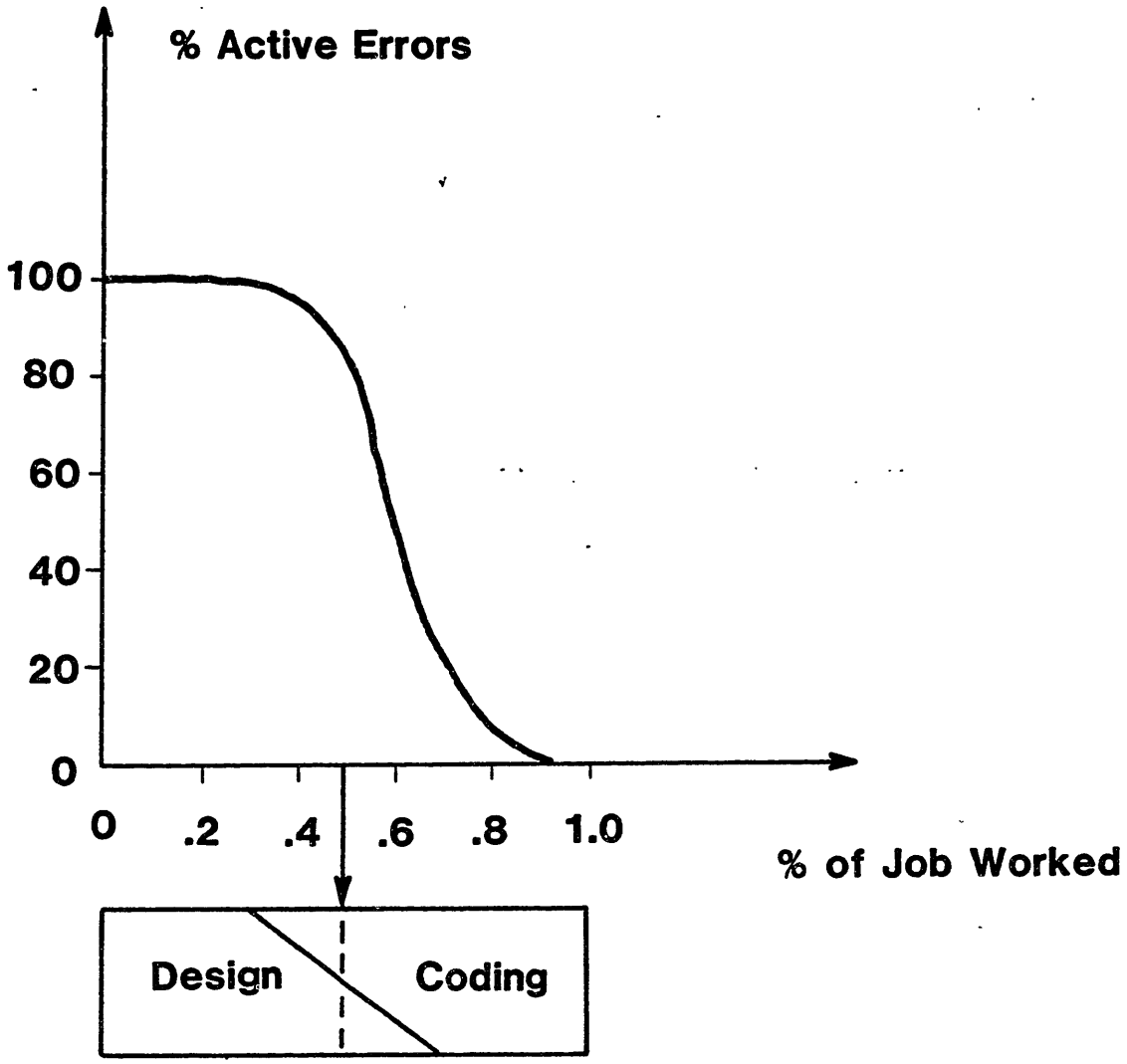


Figure III.24

"Undetected passive errors," as Figure III.23. illustrates, remain in a dormant state until they become detected and corrected in the system testing phase. The "Undetected Active Errors," on the other hand, provide a greater cause for concern, since they reproduce more and more errors into the system. This error reproduction process is a continuous one that keeps "feeding" on itself, that is, an error reproduced will itself reproduce further errors, and so on. For example, an undetected design error could lead to errors in the code, which in turn could lead to errors in the system's documentation and/or user manuals. This continuous reproduction process is formulated in the model through the "classic" positive feedback loop in which an increase in the "Undetected Active Errors" level leads to an increase in the "Active Error Regeneration Rate," leading to further increases in the level, and so on.

We now take a closer look at this positive feedback loop. First, notice that the "Active Error Regeneration Rate" is a function of the "Software Development Rate," since errors can only be generated as <u>new</u> tasks are developed. And if the development activity stops, no errors can be generated. Second, the regeneration rate is a function of the "Active Error Density," which is simply the number of existing active errors divided by the tasks developed so far. More precisely, the generation rate is a function of the SMOOTH of the "Active Error Density." This is because when

errors are committed in one part of the system, they would not, in general, affect other parts that are being developed in parallel. Errors, instead, propagate through the succeeding tasks that build on one another e.g., coding tasks developed on the basis of the design specs. Thus, there is a delay before an error would reproduce further errors. This average delay is set in the model to three months.

As was indicated by the studies cited above, a design error could be 4 to 10 times more costly when left undetected until the system testing phase. And, as was also indicated, this escalation in cost-to-fix results because of an additional inventory of various error types that would be reproduced and that would require correction. In the model, though, we do not disaggregate errors into different explicit types, e.g., errors are not disaggregated into errors in data structures, syntax, logic, ... etc. There is only one explicit error-type, namely, "Error." (This aggregation, as opposed to disaggregation, of error-types, has been justified elsewhere.) As a result, the escalation in the cost-to-fix of an undetected "Error" is realized in the model only through the number of "Errors" that the "Error" reproduces. For example, if an "Error" at the early phases of the project, reproduces (over several generations) a total of 9 more "Errors," then at testing time instead of dealing with one (the original) "Error," it would now be necessary to deal with 10 "Errors," i.e., a 10 fold escalation in cost.

The escalation in the number of active errors is achieved in the model through two mechanisms. Firstly, it is partially achieved through the "feeding on itself" characteristic of the reproduction positive feedback loop we explained above. This mechanism ensures that the earlier the undetected-error is, the more "generations" of errors it will reproduce, and thus the more costly it will end up being.

Secondly, escalation is achieved through the "Multiplier to Active Error Regeneration due to Error Density." The interpretation of this multiplier is a simple one, it represents the average number of new errors that a single active error reproduces in one generation. (That is, it is a measure of "Error Fertility!") The multiplier is formulated as a table function, and is shown in Figure III.25.

First, notice that the multiplier's value will always be greater than one. That is, an undetected error will always generate more than one more error (in a single generation). Second, the value of the multiplier increases as the density of active errors increases. Studies have shown that errors are not homogeneously distributed throughout the modules of a software system (Myers, 1976) (Endres, 1975), instead systems studied were found to be "characterized by the presence of 'error-prone modules' that show a high frequency of the system's total error content" (Jones, 1981). For example, if there are say 5 undetected errors in a system that is

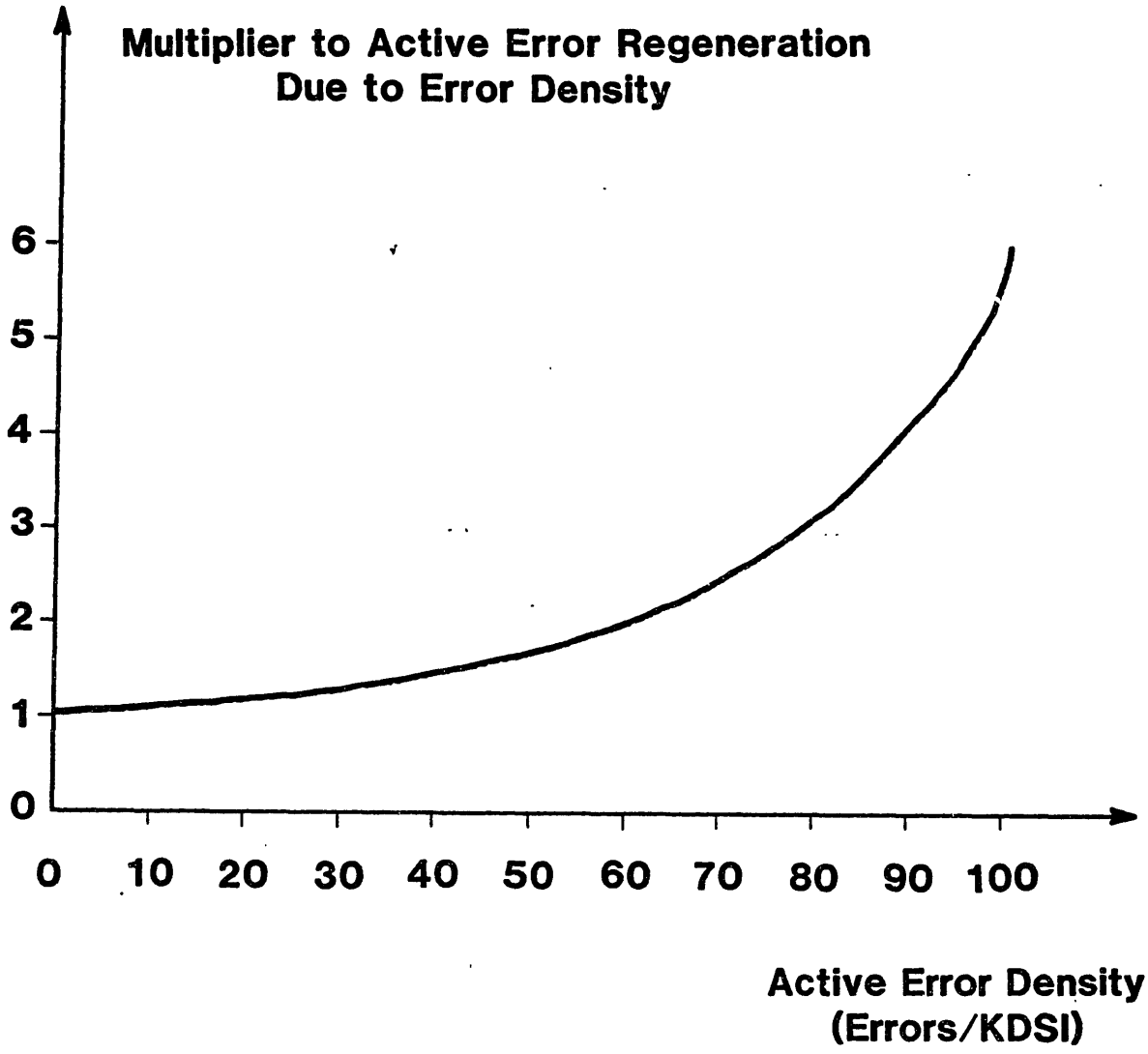


Figure III.25

comprised of 5 modules, it is quite possible that all 5 errors will be clustered in one error-prone module, as opposed to being evenly distributed among the 5 modules. If there is a much larger number of undetected errors (e.g., 100), though, it would be quite unlikely then that all the errors would still be clustered in what would be a single extremely-error-prone module. Such a situation is unlikely because we are dealing here with modules that have already "passed" some QA testing. Thus, as the error density increases, the distribution of errors among the system's modules would generally also increase. As this happens, i.e., as errors become less localized, they also become more expensive to detect and correct. For example, because of the set-up cost of testing any single module, it is generally less expensive to fix 10 errors that all reside within a single module, than fixing an equivalent set of 10 errors that are distributed among two or more modules. Thus, higher densities of undetected errors mean a wider (but not necessarily an even) distribution of errors among the system modules, which leads to an escalation in the cost to fix those errors. And since, as was indicated above, the escalation in the cost-to-fix of an undetected error is realized in the model through an increase in the number of errors that the error reproduces, higher error densities should lead to a higher error reproduction rate (per error). This is achieved through the higher values of the "Multiplier to Active Errors Regeneration due to Error Density," at

higher error densities.

As was stated above, "Undetected Active Errors" can potentially continue to reproduce new errors as long as new tasks are being developed e.g., up until the last system module is coded. Not all of the active errors will do so, however. That is, for some errors the reproduction activity will not continue up until the end of the development phase. It, instead, might cease after the reproduction of one or two "generations" of errors. For example, an error in a high-level module might reproduce a number of interface errors at some lower level, without necessarily leading to any further errors in say the user manuals. When undetected active errors cease to reproduce, they effectively become "Undetected Passive Errors." The rate at which this occurs is termed the "Active Error Retirement Rate," as shown in Figure III.23. This rate is regulated through the "Retirement Fraction," which is the fraction of active errors that retire (i.e., become passive) every unit of time. This fraction is a function of the development phase as shown in Figure III.26. Notice that, because any design error must translate into coding error(s), the "Retirement Fraction" remains at a zero level during the design phase i.e., no active design errors will retire and become passive since every design error will reproduce at least one generation of coding errors. As the project progresses towards the last stages of development e.g., the coding of the lower level

functional modules, opportunities for error propagation quickly decrease, and as a result the "Retirement Fraction" increases sharply, and reaches a value of 1 at the end of development.

As the project progresses towards the last stages of development, something else happens, namely, the System Testing activities are initiated. The objective of system Testing is to verify "that all elements (of the system) mesh properly and that overall system function and performance are achieved" (Pressman, 1982). The System Testing activities are also depicted in Figure III.23.

As was explained in Section (B) on "Software Development," the switch in manpower allocation from development to testing is effected in the model through the variable "Fraction of Effort for System Testing." The value of this variable is initially set to zero i.e., no effort is allocated for System Testing. When development (i.e., the coding and design) is perceived to be completed, the value of the "Fraction of Effort for System Testing" becomes a one, i.e., 100% of the manpower effort available for development/testing is allocated to the testing function. The switch is not abrupt, however. There is, usually, some overlap between the development and testing phases (Thibodeau and Dodson, 1980), (Daly, 1977), (Hartwick, 1980). This overlap of the phases was captured (in Section (B) above) in

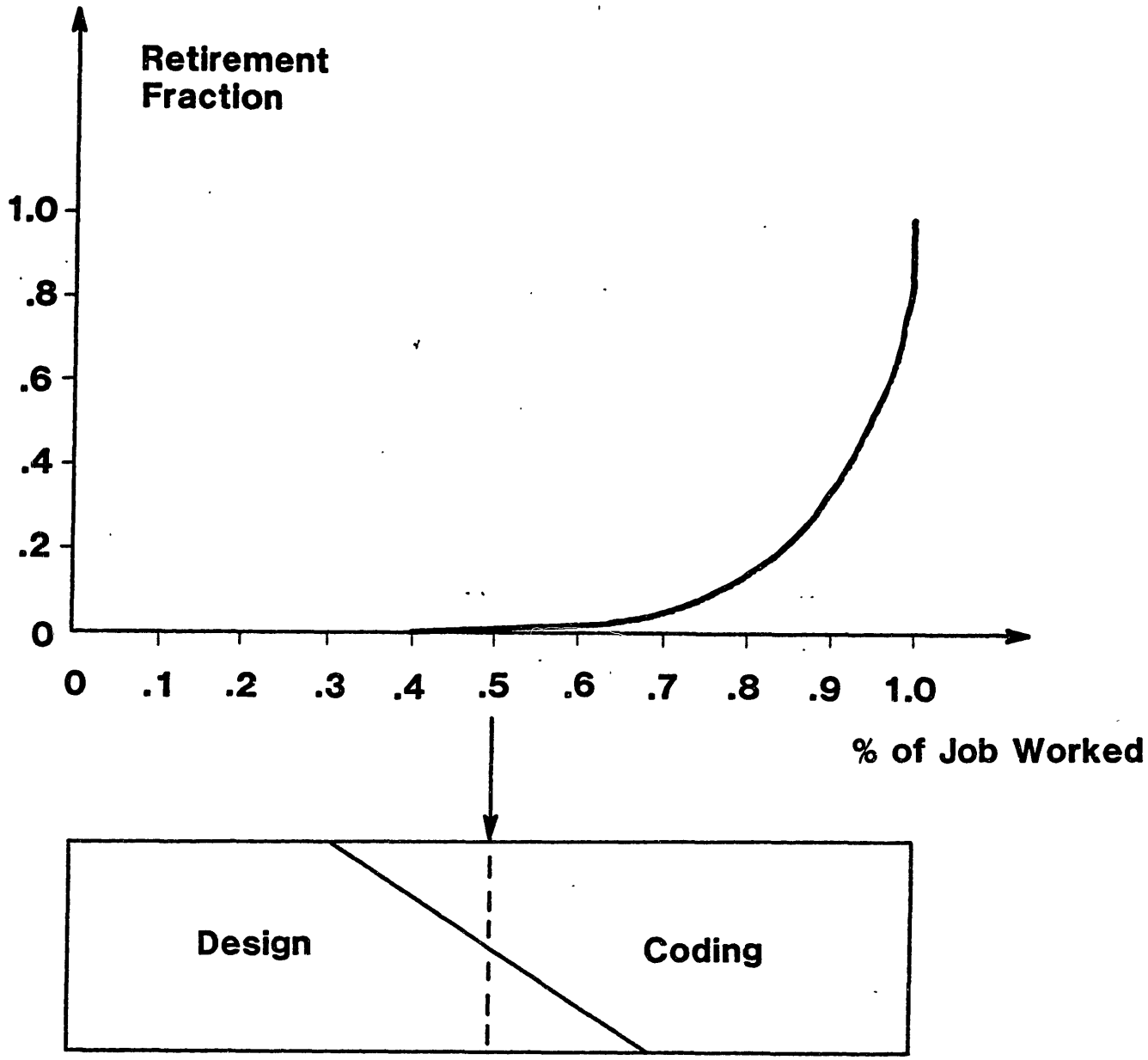


Figure III.26

Figure III.9., which shows the assumed gradual increase in the value of the "Fraction of Effort for System Testing" as a function of the fraction of development tasks perceived remaining.

The objective of System Testing stated above is operationalized in the model as follows: Test all tasks that have been developed to detect and correct any remaining (active and/or passive) errors.

The rate at which (developed) tasks are tested is determined by dividing the "Daily Manpower for Testing" by the "Testing Manpower Needed per Task." For example, if 5 man-days are allocated daily to the system testing activity, and it takes, on the average, 1 man-day to test a task, then 5 tasks will be tested a day.

The "Normal Testing Manpower Needed per Task" has two components, a fixed component and a variable one (Alberts, 1976), (Herndon and Lane, 1977). The variable component is a function of the number of errors in a task, and it represents the testing effort that would be consumed in the actual detection and correction of errors. The fixed component, on the other hand, is independent of the number of errors. It involves overhead-type activities such as developing test plans, installing test tools, designing test cases, ... etc.

The "Nominal Testing Overhead" (i.e., the fixed component) is defined in the model in terms of nominal man-days/KDSI. Estimates reported in Boehm (1981) suggest that this overhead effort is in the range of 2 man-days/KDSI. For example, for a 32 KDSI project, Boehm's estimate for the above overhead functions (which he labelled "Test Planning") amounted to 64.41 man-days. If we assume that motivation and communication losses will, on the average, result in a 50% loss in productivity, then Boehm's estimate translates into an overhead of 1 nominal man-day/KDSI.

This constant parameter, could then be transformed in a straightforward manner into an equivalent value of nominal man-days/task. For example, if in a particular run of the model, a "task" is defined to be, say 100 DSI, the nominal testing overhead would be 0.1 man-day/task.

In addition to the overhead incurred in testing a task, effort is needed to detect and correct any remaining errors. This needed effort to detect and correct the errors remaining within a task is formulated as the product of the "Error Density" and the "Nominal Testing Manpower Needed per Error." The value of the former is obtained by dividing the sum of both the active and passive errors still remaining by the number of tasks yet to be tested. It represents the average number of errors per task. The value of the "Nominal Testing Manpower Needed per Error," on the other hand, is set to 0.15

Man-Days/Error. For the nominal 8-hour working day, this translates into 1.2 Man-Hours/Error. This value was chosen on the basis of empirical results reported in (Shooman, 1983) and (Herndon and Lane, 1977).

Finally, the actual testing effort needed per task, in addition to being a function of testing overhead and error density, must also depend on the efficiency of how people work. That is, we need to account for the Communication and Motivation losses incurred. For example, if the "Multiplier to productivity due to Communication and Motivation losses," which represents the average productive fraction of a man-day, is 0.5, then the actual manpower needed to test a task becomes twice what is nominally needed.

The testing activity continues until all the tasks that have been developed are all tested. When this is accomplished, the project is declared completed. (Remember, our model's boundary extends only until the end of the testing phase.)

With the completion of the testing activties, we also complete our presentation of the software production processes in the model. We have discussed the allocation of the manpower resource in part (A), the development activties (i.e., coding and design) in (B), Quality Assurance and Rework in part (C), and finally, System Testing in this final

part (D). In the next two sections, we turn our attention to two managerial functions of software developement, namely, controlling and planning.

III.4.5. Controlling:

Any control function has at least three elements (Anthony and Dearden, 1980):

1. Measurement. To detect what is happening in the activity being controlled.
2. Evaluation. To assess the significance of what is happening, usually by comparing information on what is actually happening with some standard or expectation of what should be happening.
3. Communication. To report what has been measured and assessed, so that behavior could be altered if the need for doing so is indicated.

These three elements are captured in our formulation of the control function of software project management depicted in Figures III.27. and III.29. As work is accomplished in a software project, progress is measured through the amount of resources consumed, tasks completed, or both. Based on such measurements, a determination is made on the "Total Man-Days Perceived to be Still Needed" to complete the project. This includes man-days perceived to be still needed to develop and

QA tasks, to rework any detected errors, and to complete system testing. Once this is determined, the effort perceived to be still needed is compared to the actual "Man-Days Remaining" in the project's plan. Thus, if 100 man-days are perceived to be still needed to complete the project but only 50 man-days are remaining, the project would be perceived to be behind schedule. Conversely, if only 25 man-days are what is perceived to be still needed, while 50 man-days remain available in the project's plan, then the project would be perceived to be ahead of schedule. Once an assessment is made of any man-day shortages or excesses, behavior on the project could be altered if the need for doing so is indicated. For example, if the project is perceived to be behind (ahead of) schedule, i.e., if it is experiencing a man-day shortage (excess), then project members could be motived to work more (less) hard, the project's schedule could be extended (trimmed), or a combination of both of these could happen. In the remaining part of this section, we will explain in detail how all these control processes are formulated in the model.

At any point in the project, the amount of project work that will be perceived as still remaining will, in general, be a combination of three things: (1) work needed to develop and QA new tasks; (2) work needed to rework any detected errors; and (3) work needed to conduct the system testing activities. Thus, the "Total Man-Days Perceived to be Still

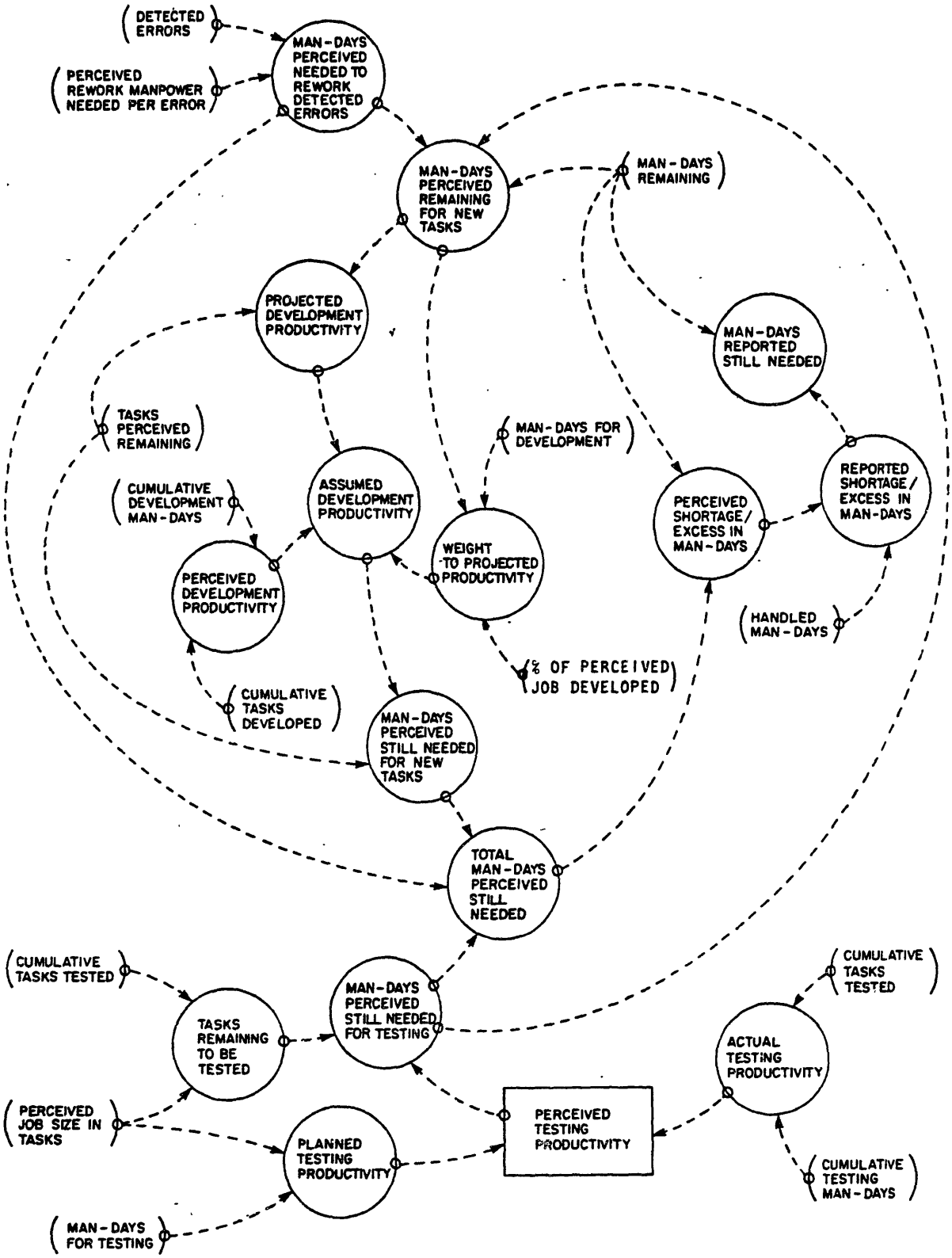


Figure III.27

Needed" to complete the project is formulated as a summation of three respective components, namely, "Man-Days Perceived Still Needed for New Tasks," "Man-Days Perceived Needed to Rework Detected Errors," and "Man-Days Perceived Still Needed for Testing."

Because software is basically an intangible product during most of the development process, and for which there are no visible milestones to measure progress like a physical product would, "It is difficult to measure performance in programming ... It is difficult to evaluate the status of intermediate work such as underdebugged programs or design specification and their potential value to the complete project" (Mills, 1983). How, then, is progress in a software project measured? Our own interview findings corroborate those reported in the literature, namely, that progress, especially in the earlier phases of software development, is measured by the rate of expenditure of resources rather than by some count of accomplishments (Putnam and Fitzsimmons, 1979), (Keider, 1974), (DeMarco, 1982), (Devenny, 1976), (Baber, 1982), (Griffin, 6), (Donahue, 8), (O'Conner, 10), (Lombardi, 16), (Chan, 20). For example, a project for which 100 man-days has been estimated is 10% complete when 10 man-days have been expended; when 50% of the man-days have been expended, it is 50% complete. Paraphrasing Baber (1982):

> 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.

As progress is measured, during the early phases of development, by the rate of expenditure of resources, status reporting ends up being nothing more than an echo of the original plan (McKeen, 1981), (Baber, 1982), (DeMarco, 1982), (Devenny, 1976). In other words, "Man-Days Perceived Still Needed for New Tasks" will be equal to the "Man-Days Perceived Remaining for New Tasks."

As the project develops, though, and the work becomes relatively more visible, discrepancies between % of tasks accomplished (remaining) and % of resources expended (remaining) become increasingly apparent. For example, while it might not be too apparent that a project that has consumed 50% of its estimated resources is only 25%, rather than 50%, complete, any such descrepancy becomes quite obvious when the allocated resources are almost used up. At the same time, and as the project advances towards its final stages, project members become increasingly able to perceive how productive the workforce has actually been (McGowan, 3), (Nichols, 18). As a result, the value of the "Man-Days Perceived Still

Needed for New Tasks" ceases to be a function of what the "Man-Days Perceived Remaining for New Tasks" is, and, instead, is determined on the basis of what the project members perceive to be the amount of work that is still remaining.

These differring modes of measuring progress, are captured in the model through a single formulation of "Man-Days Perceived Still Needed for Needed Tasks." As shown in Figure III.27., "Man-Days Perceived Still Needed for New Tasks" (MDPNNT) is determined by dividing the value of "Tasks Perceived Remaining" (TSKPRM) by the "Assumed Development Productivity" (ASSPRD). That is,

$$MDPNNT = TSKPRM / ASSPRD \quad (1)$$

Where "Assumed Development Productivity" (ASSPRD) is a weighted average of "Perceived Development Productivity" (PRDPRD) and a variable we are calling "Projected Development Productivity" (PJDPRD). That is,

$$ASSPRD = PJDPRD*WTPJDP + PRDPRD*(1-WTPJDP) \quad (2)$$

The weighting factor (WTPJDP) moves from 1 at the beginning of the project to zero at the end of the development phase.

The conception behind this formulation is somewhat

subtle, and will, therefore, require some explanation.

As was indicated above, in the earlier phases of software development, progress tends to be measured by the rate of expenditure of resources. As a result, status reporting ends up being nothing more than an echo of the original plan. "Man-Days Perceived Still Needed for New Tasks" (MDPNNT) becomes, under such conditions, simply equal to the "Man-Days Perceived Remaining for New Tasks" (MDPRNT). That is,

$$MDPRNT = MDPNNT$$

Substituting for MDPNNT, we get

$$MDPRNT = TSKPRM / ASSPRD$$

which leads to,

$$ASSPRD = TSKPRM / MDPRNT$$

This is an interesting result. For, it suggests that as project members measure and report progress by the rate of expenditure of resources, they, by so doing, would be implicitly assuming that their productivity equals "Tasks Perceived Remaining" (TSKPRM) divided by the "Man-Days Perceived Remaining for New Tasks" (MDPRNT). Which is intersting because such an assumed value for productivity is solely a function of future projections (i.e., remaining tasks and man-days) as opposed to being a reflection of

accomplishments (i.e., completed tasks and expended resources). This implicit notion of productivity is captured in the model by the variable "Projected Development Productivity" (PJDPRD), defined, as the above equation suggests, to be equal to "Tasks Perceived Remaining" (TSKPRM) divided by "Man-Days Perceived Remaining for New Tasks" (MDPRNT).

Thus, in the early phases of software development, we would like equation (1) to reduce to,

$$MDPNNT = TSKPRM / PJDPRD \quad (3)$$

where

$$PJDPRD = TSKPRM / MDPRNT$$

which would be achieved by setting the weighting factor (WTPJDP) in equation (2) to 1, and substituting in equation (1).

As the project advances towards its final stages, though, accomplishments become relatively more visible and project members become increasingly more able to perceive how productive the workforce has actually been. As a result, what the project members assume their productivity to be, i.e., the value of "Assumed Development Productivity," ceases to be a function of future projections (i.e., remaining tasks and man-days), and instead is determined on the basis of

perceived accomplishments. This explicit notion of productivity is captured in the model by the variable "Perceived Development Productivity" (PRDPRD). Discussions with (McGowan, 3), (Nichols, 18), and (Lombardi, 23) suggest that, towards the final stages of development, the value of the team's overall productivity would be determined by dividing the value of "Cumulative Tasks Developed" (CUMTKD) by "Cumulative Development Man-Days" (CUMDMD). In other words, if 100 man-days have been expended to develop the project's 100 tasks, then "Perceived Development Productivity" would be 1 task/man-day.

Thus, in the final stages of software development, we would like equation (1) to reduce to,

$$MDPNNT = TSKPRM / PRDPRD \qquad (4)$$

where,

$$PRDPRD = CUMTKD / CUMDMD$$

which would be achieved by setting the weighting factor (WTPJDP) in equation (2) to zero, and substituting in equation (1).

To recapitulate, the value of "Man-Days Perceived Still Needed for New Tasks" (MDPNNT) is a function, as equation (1) indicates, of "Tasks Perceived Remaining" (TSKPRM) and "Assumed Development Productivity." In the early phases of

development, "Assumed Development Productivity" is implicitly determined on the basis of future projections (i.e., remaining tasks and man-days). Towards the end of development, on the other hand, "Assumed Development Productivity" gets to be explicitly determined on the basis of perceived accomplishments (i.e., completed tasks and expended resources). This is achieved through the weighted average formulation of "Assumed Development Productivity" given in equation (2), i.e., by setting the weighting factor (WTPJDP) to 1 at the beginning of the project, and to zero at the end of the development phase.

People's assumptions about their productivity, therefore, change as the project develops. The change, however, is often gradual not abrupt (McGowan, 3), (Nichols, 18), (Lombardi, 23). That is, the transition from having "Assumed Development Productivity" being determined solely on the basis of future projections early in the project, to having it being determined entirely on the basis of perceived accomplishements, towards the end of development, is a smooth, not an instantaneous, type of a transition.

This transition in people's assumption about their productivity is captured in the model through the formulation of the weighting factor (WTPJDP) of equation (2). For convenience, we are re-writing equation (2) below,

ASSPRD = PJDPRD*WTPJDP + PRDPRD*(1-WTPJDP) (2)

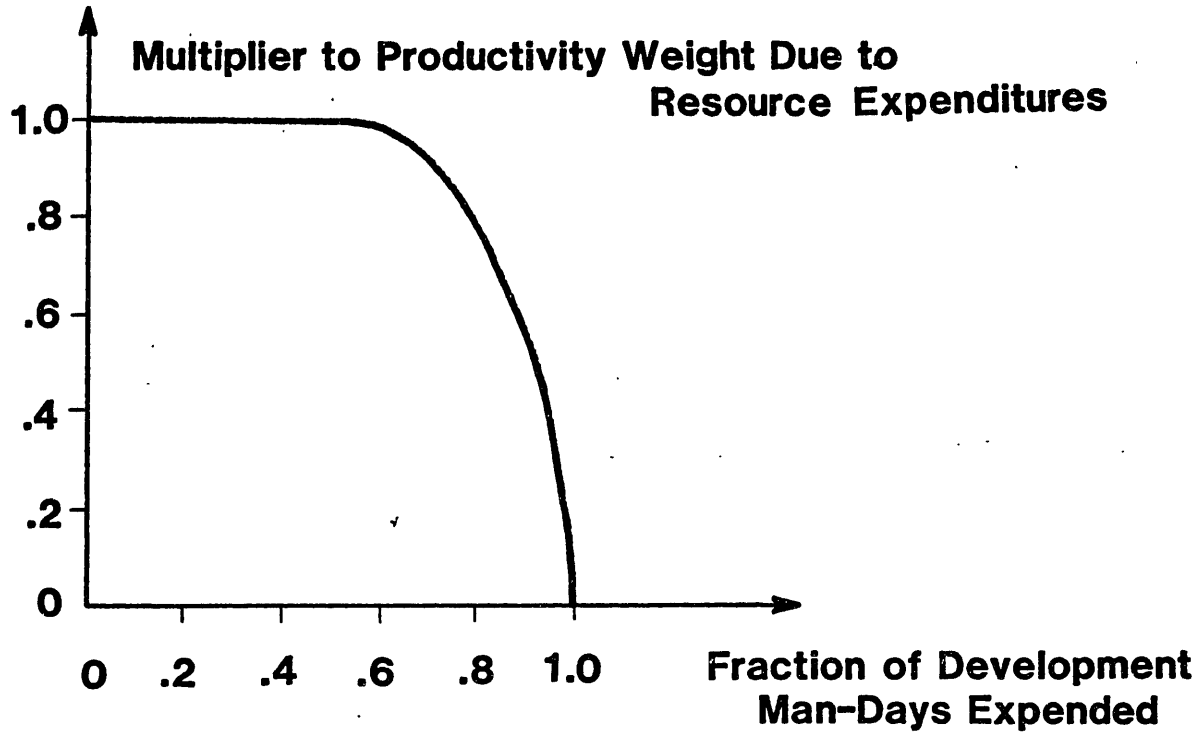
In the beginning of the project, because "Assumed Development Productivity" (ASSPRD) is equal to "Projected Development Productivity" (PJDPRD), the weighting factor WTPJDP is set equal to 1. As was explained above, under such conditions status reporting ends up being nothing more than an echo of the original project plan as "Man-Days Perceived Still Needed for New Tasks" ends up being exactly equal to "Man-Days Remaining for New Tasks." As the project develops, though, descrepancies between % of tasks accomplished (remaining) and % of resources expended (remaining) become increasingly apparent, and in addition project members become increasingly able to perceive how productive the workforce has actually been. As a result, "Assumed Development Productivity" (ASSPRD) becomes less a function of "Projected Development Productivity" (PJDPRD) and more a function of "Perceived Development Productivity" (PRDPRD). That is, the weighting factor (WTPJDP) moves from a value of 1 to a value of 0. The rate at which this learning process takes place is the product of two factors, namely, the rate of expenditure of resources and the rate of development of tasks. Remember Baber's quote (1982), "Only when the program is almost finished or when the allocated time budget is almost used up will (the programmer) be able to recognize (the descrepancy between % of tasks accomplished and % of resources expended." To accomplish this in the model, we will formulate the

weighting factor (WTPJDP) as the product of two multipliers, the "Multiplier to Productivity weight due to Resource Expenditures" and the "Multiplier to Productivity Weight due to Development." As shown in Figure III.28., both multipliers are assumed to have the same shape, moving from a value of 1 in the beginning of the project to a value of zero when all estimated development resources are expended or all tasks are developed, respectively.

Thus far we have been only discussing how "Man-Days Perceived Needed for New Tasks" is determined. As was indicated earlier, at any point in the project the amount of work that will be perceived as still remaining will, in general, be comprised of not only work needed to develop and QA new tasks, but in addition work needed to rework any detected errors and work needed to conduct the system testing activities. Thus, the "Total Man-Days Perceived to be Still Needed" to complete the project is formulated as a summation of "Man-Days Perceived Still Needed for New Tasks," "Man-Days Perceived Needed to Rework Detected Errors," and "Man-Days Perceived Still Needed for Testing."

The "Man-Days Perceived Needed to Rework Detected Errors" is formulated as the product of "Detected Errors" and "Perceived Rework Manpower Needed per Error." (The latter, as was explained in some detail in the section on "Manpower Allocation," is a SMOOTH of the "Actual Rework Manpower

(a)



(b)

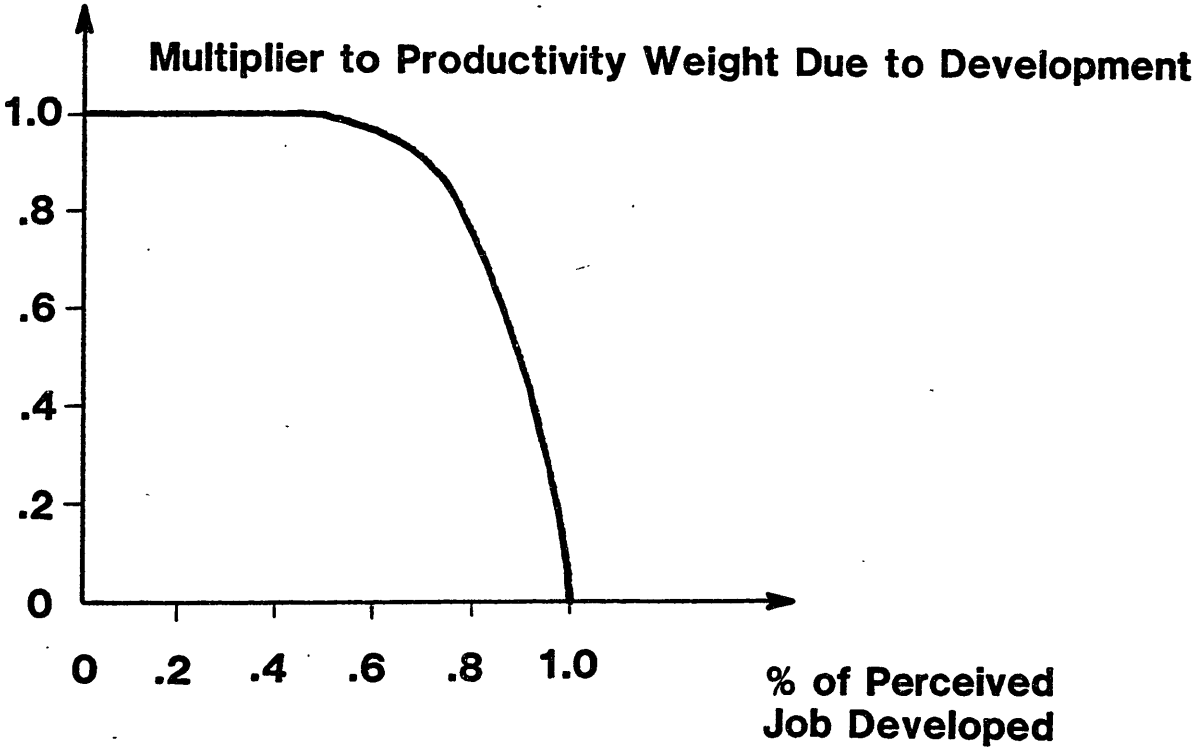


Figure III.28

Needed per Error.") For example, if at some point in the project 50 errors that have been detected through the QA activities are still uncorrected, and if it is perceived that an error requires 0.2 Man-Days, on the average, to correct, then the "Man-Days Perceived Needed to Rework (those) Detected Errors" would be 50 X 0.2 = 10 Man-Days.

The "Man-Days Perceived Still Needed for Testing," on the other hand, is determined by dividing the value of "Tasks Remaining to be Tested" by the "Perceived Testing Productivity." The "Tasks Remaining to be Tested" is simply the "Perceived Job Size in Tasks" minus "Cumulative Tasks Tested." For example, if the perceived job is 100 tasks in size, and 60 of these have already been tested, then "Tasks Remaining to be Tested" would amount to 100 - 60 = 40 tasks.

Throughout most of the development phase, and <u>before</u> the commencement of the System Testing phase, the value of "Perceived Testing Productivity" is set equal to "Planned Testing Productivity." This is the value of testing productivity that is implicit in the project's plan. For example, if for the 100 task project, the plan allocates 20 Man-Days for System Testing, then the "Planned Testing Productivity" would be 5 tasks/man-days. However, as the System Testing activity gets underway, people's perceptions of their testing productivity becomes a function of how productive the testing activity <u>actually</u> is, as opposed to

how productive it was planned to be. The "Actual Testing Productivity" is then determined by dividing the "Cumulative Tasks Tested" by "Cumulative Testing Man-Days." And, because "Full and immediate action is seldom taken on a change of incoming information (e.g., on the sudden drop in yesterday's testing productivity) ... (and because there is a) tendency to delay action until the change is insistent ..." (Forrester, 1961), "Perceived Testing Productivity" is formulated as a SMOOTH. The smooth delay is set at 50 working days.

Once "Man-Days Perceived Still Needed for New Tasks," "Man-Days Perceived Needed to Rework Detected Errors," and "Man-Days Perceived Still Needed for Testing" are all determined, they would all be summed up to determine the "Total Man-Days Perceived Still Needed" to complete the project. And once this is determined, it is then compared to the actual "Man-Days Remaining" in the project's plan. So, if 100 man-days are perceived to be still needed to complete the project, but only 50 man-days are remaining, the project would be perceived to be behind schedule. Conversely, if only 25 man-days are what is perceived to be still needed, while 50 man-days remain available in the project's plan, then the project would be perceived to be ahead of schedule.

After an assessment is made of any man-day shortages or excesses, behavior on the project can then be altered if the

need for doing so is indicated. For example, if the project is perceived to be behind (ahead of) schedule i.e., if it is experiencing a man-day shortage (excess), then project members could be motivated to work more (less) hard. The mechanisms that determine how much, if any, of any perceived man-day shortage (excess) is absorbed by the project members in the form of increased (decreased) work rate were fully explained in our discussions on software development productivity. Any shortages (excesses) that are not absorbed will be reported, and will lead to adjustments to the project's scope. (Such adjustments are then translated, in the Planning section, into adjustments to the schedule or adjustments to the workforce level, or both.)

Let us consider an example. And, again, let us consider the case of the 100 man-day project. If, after 60 man-days have been expended, the values of "Man-Days Remaining" and "Total Man-Days Perceived Still Needed" were 40 man-days and 65 man-days respectively, then the "Perceived Shortage in Man-Days" would be 25. If the project members (based on the many factors discussed in the productivity section) decide to absorb only 10 of the 25 man-days, then the "Reported Shortage in Man-Days" would be 15 man-days. If these are added to the value of "Man-Days Remaining" in the project's plan, i.e., to 40, we come up with a value of 55 man-days for the "Man-Days Reported Still Needed" to complete the project.

Any time the "Man-Days Reported Still Needed" turns out to be more (less) than the "Man-Days Remaining" in the project's plan, it would, in effect, constitute a revision of what the project's scope is perceived to be, i.e., that it is larger (smaller) than what has been planned for. For example, in the case above, reporting that 55 (rather than 40) man-days are still needed after having had 60 man-days already expended, constitutes a revision in what the project's size is perceived to be, namely, from the original estimate of 100 man-days to a revised value of 60 + 55 = 115 man-days i.e., a 15% increase. When such a "revelation" occurs in a project, project management reacts to transform those revised perceptions about the "Total Job Size in Man-Days" into actual adjustments. This adjustment process is captured, as is shown in Figure III.29., through the "Rate of Adjusting the Job's Size in Man-Days." It is the rate at which the "Total Job Size in Man-Days" is adjusted, upwards or downwards, to what is perceived as its newly revised value. The "Rate of Adjusting the Job's Size in Man-Days" is formulated as,

(GOAL - LEVEL)/ADJUSTMENT-TIME

where,

GOAL = Revised value of job size in Man-Days
= Man-Days Reported Still Needed + Cumulative Man-Days Expended

LEVEL = Total Job Size in Man-Days

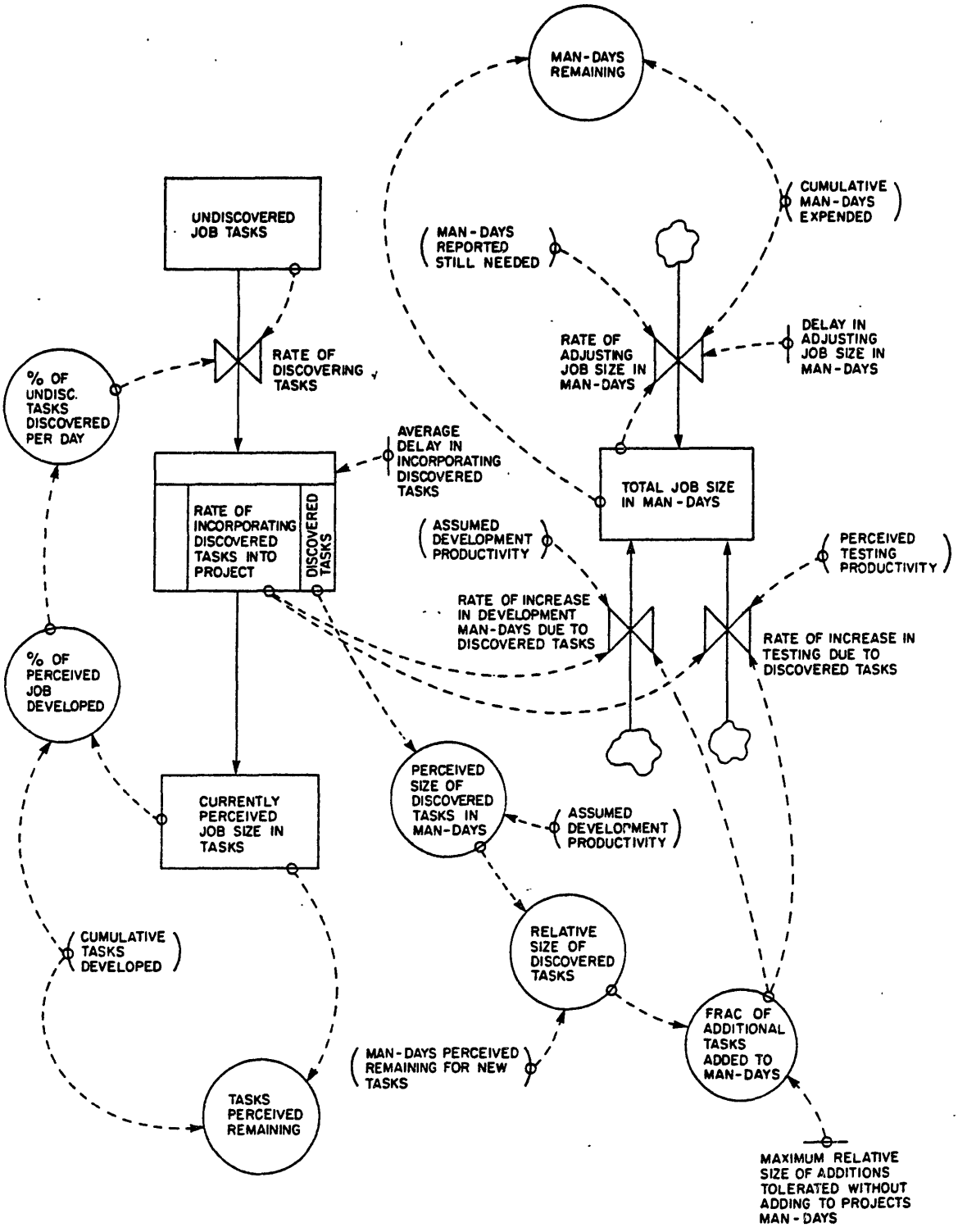


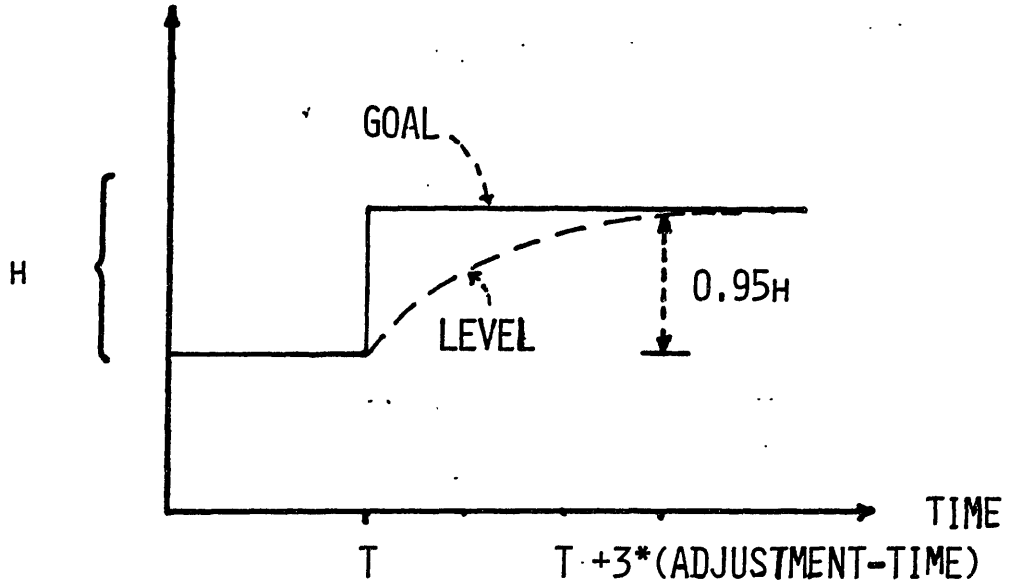
Figure III.29

ADJUSTMENT-TIME = Delay in Adjusting the Job's Size in Man-Days

Thus, the adjustment process is not an instantaneous one, instead it takes place over a time interval defined as the "Delay in Adjusting the Job's Size in Man-Days."

The above formulation of the "Rate of Adjusting the Job's Size in Man-Days" produces the behavior pattern shown in Figure III.30. In the situation portrayed in the figure, it is assumed that up until time t_1, LEVEL = GOAL. Then, at time (t_{1+}) there is a sudden permanent increase (h) in the GOAL e.g., the "Revised Value of the Job's Size" jumps from 100 man-days to 115 man-days. In response to such a step rise in the value of the GOAL, the value of LEVEL (e.g., the value of "Total Job Size in Man-Days") rises in an exponential, goal seeking pattern. And, it can be shown that, the rate at which LEVEL rises is such that it would close 63% of the gap after one "Adjustment time," and 95% of the gap after 3 "Adjustment-times."

The "Delay in Adjusting the Job's Size in Man-Days" ranged in the organizations we interviewed in from 2 days (Landolfi, 22), (Lombardi, 23) to a week (i.e., 5 working days), (Chan, 20). In the model the "Delay in Adjusting the Job's Size in Man-Days" is set to 3 working days. This value together with the ones reported by our interviewees might strike some readers as somewhat lower than what they would



RATE = (GOAL - LEVEL) / (ADJUSTMENT-TIME)

Figure III.30

have expected. But remember, this adjustment process is really the project's final, not first, reaction to some man-day shortage/excess. As we explained before, when the project is perceived to be behind (ahead) of schedule people first react by absorbing the shortage (excess). And only when this is not enough, are adjustments to the project's size undertaken. Thus, when, if ever, the the decision to also adjust the project's size is made, people in the project would have been "geared-up" for it.

Falling behind schedule is not the only reason why a project's size in man-days might be adjusted upwards. It could also happen, as Figure III.29. indicates, as a result of an upward adjustment in the project's size in tasks.

As a software project develops, project members often realize that they have under-estimated the number of tasks (e.g., modules) that comprises the software system being developed (DeMarco, 1982), (Burchett, 1982), (Daly, 1977), (Devenny, 1976). Boehm (1981) provides an explanation for this tendency to underestimate software size:

> There is a powerful tendency to focus on the highly visible mainline components of the software, and to underestimate or completely miss the unobtrusive components (e.g., help message processing, error processing, and moving data around).

In the model we define an initializing parameter called

"Tasks Underestimation Fraction." Through this parameter we can simulate any software under-sizing situation we wish to investigate. For example, if the actual size of the software product to be developed is, say, 100 tasks, then to simulate a 25% under-sizing "problem" we would simply set the "Tasks Underestimation Fraction" to 0.25. What this would do, is it would initialize the model such that the value of the "Currently Perceived Job Size in Tasks" is only (1 - 0.25) * 100 = 75 tasks. It would also initialize another level, namely, the "Undiscovered Job Tasks" to 0.25 * 100 = 25 tasks.

As the project develops, the "Undiscovered Job Tasks" are progressively discovered as "the level of knowledge we have of what the software is intended to do (increases)" (Boehm, 1981). The rate at which this happens, i.e., the number of undiscovered tasks that would be discovered per unit of time, is regulated, in the model, by the "Rate of Discovering Tasks." It is formulated as the product of the number of "Undiscovered Job Tasks" and the "Percent of Undiscovered Tasks Discovered per Day." Because the rate at which undiscovered-tasks are discovered tends to increase as the project develops (Daly, 1977) (e.g., because, as the above quote indicates, the team's level of knowledge of what the software product is intended to do increases), the "Percent of Undiscovered Tasks Discovered per Day" is formulated, not as a constant, but instead as a variable that

increases in value as the project progresses. Its formulation is depicted in the table function of Figure III.31.

As the additional tasks are discovered, they are then incorporated into the project e.g., incorporated into the project's Work Breakdown Structure, the Gantt and/or PERT charts, the Earned Value System, ... etc. This, of course, takes time. In the model this process is modeled as a third-order delay, with the "Average Delay in Incorporating Discovered Tasks" set to 10 working days (i.e., two weeks) (Landolfi, 22).

The final piece of structure we would like to discuss is the one that model's the process by which the discovery of additional tasks is translated into additions to the project's allocation of man-days. This structure occupies the lower portion of Figure III.29.

When additional tasks are discovered in a project, they do <u>not</u> necessarily always trigger an adjustment to the project's man-days estimate (Boehm, 1981). Only if the additional tasks are perceived as requiring a <u>relatively</u> "significant" amount of effort to handle, would project members "bother" to go through the trouble of formally developing cost estimates and incorporating them in the project's work plan (Chan, 20), (Lombardi, 23), (Hisamune,

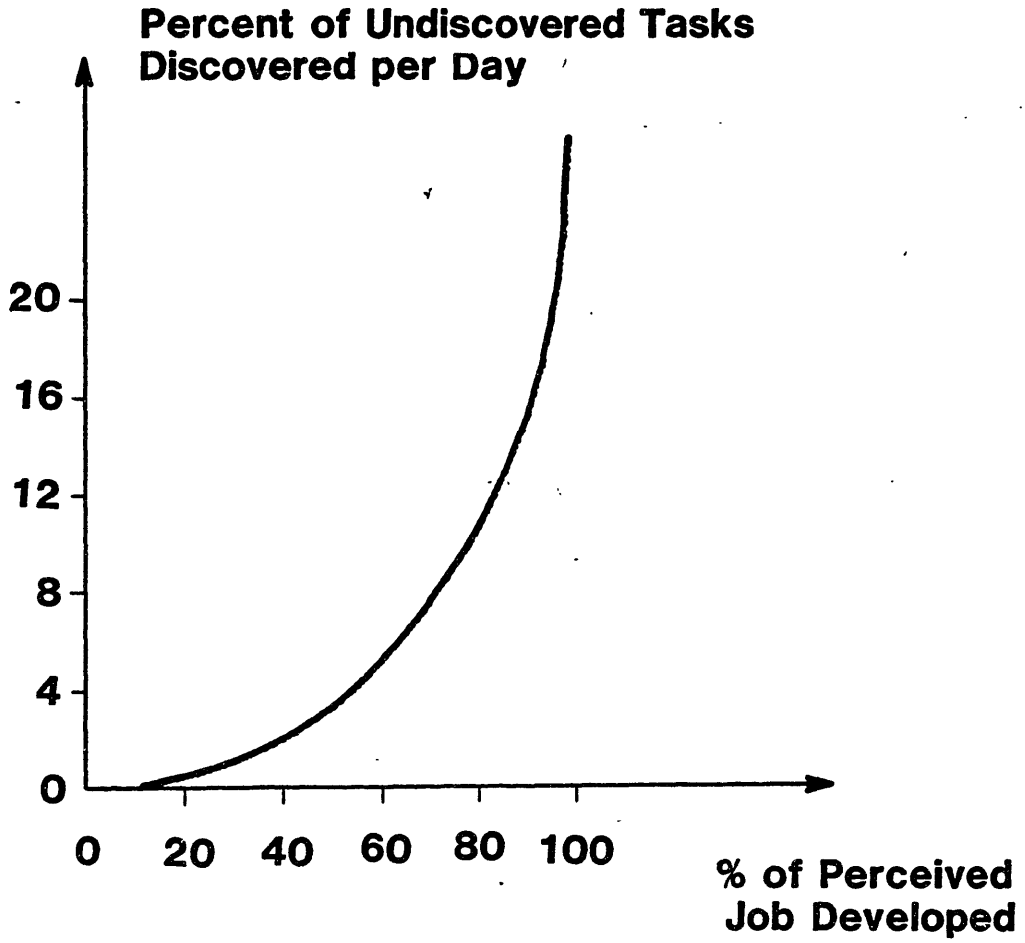


Figure III.31

26) and (Nichols, 27). As Figure III.29. indicates, the number of discovered tasks are first "mentally" sized-up by dividing them by the "Assumed Development Productivity." For example, if 10 tasks are discovered and if, at that point in the project, the value of the "Assumed Development Productivity" is 1 task/man-day, then the "Perceived Size of Discovered Tasks in Man-Days" would be 10 man-days. This absolute number by itself is not, however, enough to decide whether the new tasks do or do not deserve a "formal treatment." This determination is based, not on the perceived absolute size of the discovered tasks, but instead on what their size is perceived to be relative to the amount of effort that is perceived remaining. For example, while it would be quite possible that a 100 man-day task discovered at the beginning of 100,000 man-day project would not trigger any adjustments in the projects's man-days estimate, it would be quite unlikely for this to happen if the 100 man-day task is discovered at the end of the development phase when only 50 man-days are still remaining in the project's plan. Thus, the value of the "Perceived Size of Discovered Tasks in Man-Days" is divided by the "Man-Day Perceived Remaining for New Tasks" to determine the "Relative Size of Discovered Tasks." Once this relative size is determined, it is then compared to some threshold value, namely, the "Maximum Relative Size of Additions Tolerated Without Adding to the Project's Man-Days." If the relative size is lower than that threshold, the newly discovered tasks are totally absorbed

without triggering any adjustments to the project's man-days estimate. If, however, the relative size exceeds that threshold value, parts or all of the additional tasks are translated into additional man-days in the project's plan. This behavior is captured in the table function of Figure III.32. Based on discussions with (Hisamune, 26), and (Nichols, 27), we set the "Maximum Relative Size of Additions Tolerated Without Adding to Schedule to the Project's Man-Days" to 1%. For example, for a 1000 man-day development phase (e.g., 10 people working for 100 working days the threshold is 10 man-days).

As a result of the above decision making process, a decision could, therefore, be made to formally incorporate either part or all of those tasks discovered, at some point in the project, into the project's man-days estimate. Such an adjustment involves producing two estimates, one for the effort to develop and QA the new tasks, and the other for the system testing work. Both of these estimates are determined in basically the same manner. The former is determined by dividing the number of discovered tasks that are to be incorporated by the "Assumed Development Productivity," while the system testing effort is estimated by dividing by the "Perceived Testing Productivity."

Any such adjustments to the project's total man-days estimate, will, in turn, trigger further adjustments in

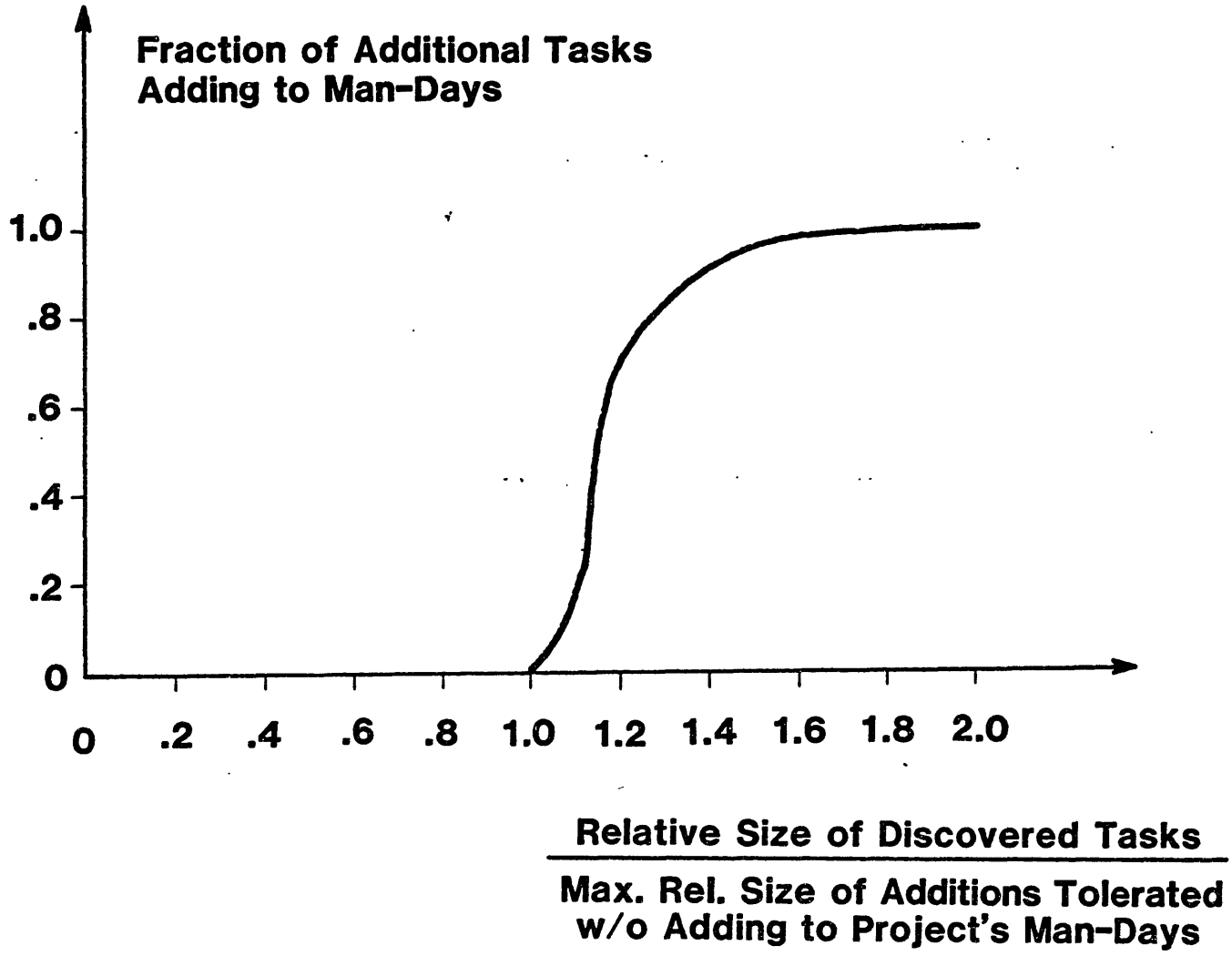


Figure III.32

either the projects schedule completion date, the workforce level, or both. These reactions are explained next in the planning section.

III.4.6. Planning:

The Planning subsystem is depicted in Figure III.33.

The "Schedule Completion Date" is formulated, not as an actual date (e.g., August 7th, 1983), but as a number of working days from the beginning of the project (e.g., 200 days). Thus, by simply subtracting the current value of "Time" (which represents the number of working days elapsed in a simulation run), we can determine the scheduled "Time Remaining." By dividing the value of "Man-Days Remaining," at any point in the project, by "Time Remaining" we can then determine the "Indicated Workforce Level." This would represent the number of full-time employees believed to be necessary and sufficient to complete the project on time i.e., on the (current) "Scheduled Completion Date." For example, if the "Scheduled Completion Date" is 100 days, and at time = 40 days the value of "Man-Days Remaining" is 600 man-days, the "Indicated Workforce Level" would be determined as follows: First, the value of "Time Remaining" would be determined to be 100 - 40 = 60 days. Dividing this into 600 man-days, we arrive at a value for the "Indicated Workforce Level" of 10 men. As we said, this value is in terms of

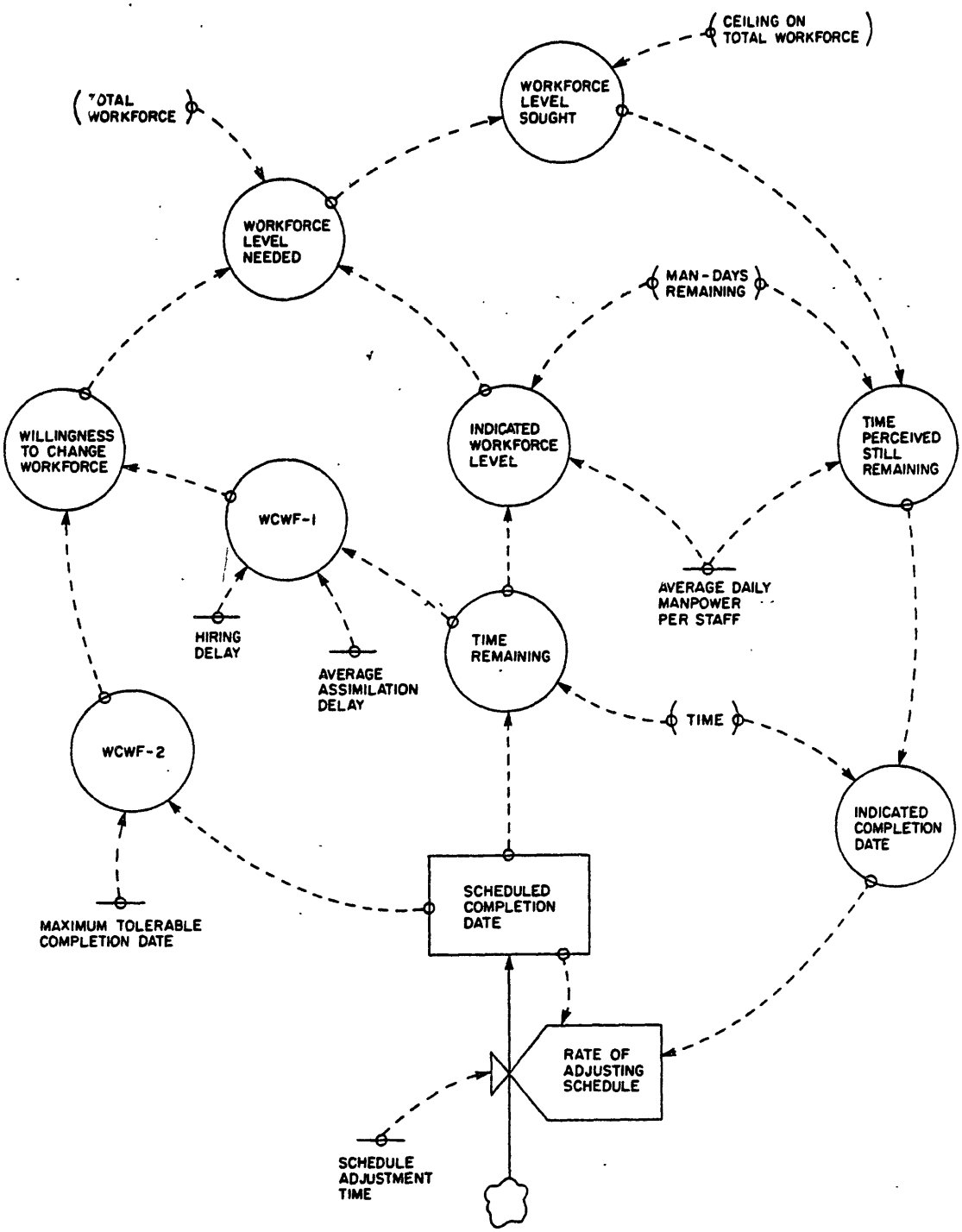


Figure III.33

full-time equivalent employees. Thus, if (actual) employees are not assigned full-time on the project, adjustments should be made. This is achieved in the model by dividing the value of the "Indicated Workforce Level" obtained above, by the value of the "Average Daily Manpower per Employee." For example, if employees assign, on the average, only 50% of their time to the project, i.e., "Average Daily Manpower per Employee" equals 0.5, then the "Indicated Workforce Level" obtained above would be adjusted to become 10 / 0.5 = 20 (actual) employees.

As was mentioned, the "Indicated Workforce Level" represents the number of full-time employees believed to be necessary and sufficient to complete the project on time i.e., on the (Current) "Scheduled Completion Date." If this number turns out to be lower than the value of the "Total Workforce" on the project, excessive employees would be simply transferred out of the project. The transfer operation was explained in detail in the "Human Resource Management Subsystem." If, on the other hand, the opposite is true, i.e., the "Indicated Workforce Level" is larger, then this would indicate a need to hire more people. However, as has also been explained in the "Human Resource Management Subsystem," hiring decisions are not determined only on the basis of scheduling considerations. In addition, consideration is also given to the stability of the workforce. That is, before hiring new project members,

management tries to contemplate the duration of need for these new members. Different firms weigh this factor to various extents. In general, however, the relative weighting between the desire for workforce stability on the one hand, and the desire to complete the project on time, on the other, changes with the stage of project completion.

The "Workforce Level Needed" is formulated as a weighted average of the (Current) "Total Workforce Level" and the "Indicated Workforce Level." It, thus, takes into account both the stable workforce level, and the number of employees that would be required to complete the project on time. Specifically, it is formulated as follows:

```
WF-Level Needed  = Indicated WF-Level * WCWF +
                   Total WF-Level * (1-WCWF)
```

(Note: The above formulation only applies when the value of the "Indicated Workforce Level" is larger than "Total Workforce," indicating a need for hiring more people. In cases where the opposite is true, i.e., "Indicated Workforce Level" is lower, then "Workforce Level Needed" would be simply set to that lower value, and any excessive employees transferred out of the project.)

The weighting factor (WCWF) is termed the "Willingness to Change Workforce Level." It is a variable that could assume values between 0 and 1, inclusive. When WCWF = 1, the

weighting considers only the "Indicated Workforce Level" i.e., management would be adjusting its workforce level to the number perceived required to finish on schedule. As WCWF moves towards 0, more and more weighting would be given to the stability of the workforce. And when WCWF equals exactly 0, the weighted number of employees desired becomes wholly dependent on the workforce stability factors.

We formulated the "Willingness to Change Workforce Level" to be comprised of two components. The first component, WCWF-1, captures the pressures that develop, as the project proceeds towards its final stages, for workforce stability. Although different firms will weigh this factor to various extents, we feel that the general form of WCWF-1 depicted in Figure III.34. (and which is based on discussions with (Lombardi, 23), (Garett, 24), and (Nichols, 25) is representitive. To understand what Figure III.34. represents, assume for the moment that "Willingness to Change Workforce Level" is only comprised of, and is therefore equal to, WCWF-1. Thus, in the early stages of the project when "Time Remaining" would generally be much larger than the sum of the "Hiring Delay" and the "Average Assimilation Delay," WCWF would be equal to 1, i.e., there would be total willingness to adjust the size of the workforce to whatever level is necessary to suit the project's scheduled completion date. As the number of days perceived remaining drops below 1.5 * (Hiring Delay + Average Assimilation Delay), though,

the figure shows increasing reluctance to increase the workforce level. For example, if the "Hiring Delay" is 40 working days and the "Average Assimilation Delay" is 80 days, then as "Time Remaining" drops below 180 days, management starts to become reluctant to hire new people, even though the time and effort perceived remaining might imply that more people are needed. This reluctance stems from the realization that most of those remaining 180 days, would be "wasted" in the hiring process and then in acquainting the new people with the mechanics of the project, in integrating them into the project team, and in training them in the necessary technical areas. When the "Time Remaining" drops below 0.3 * (Hiring Delay + Average Assimilation Delay), the table function of Figure III.34. suggests that no more additions would be made to the project's workforce i.e., the hiring rate will fall to zero. Thus, at that stage, if the project is behind schedule, project management would be coping only by pushing back the schedule completion date.

This, of course, is not always feasible or acceptable. For example, in our discussions at MITRE, we learned that in projects that involve embedded software for weapon systems, serious schedule slippages can not be tolerated. The reason is that, in such projects, software development is often on the critical path of overall system development, which, as a result, translates any serious slippages in the software schedule into very costly slippages in the overall delivery

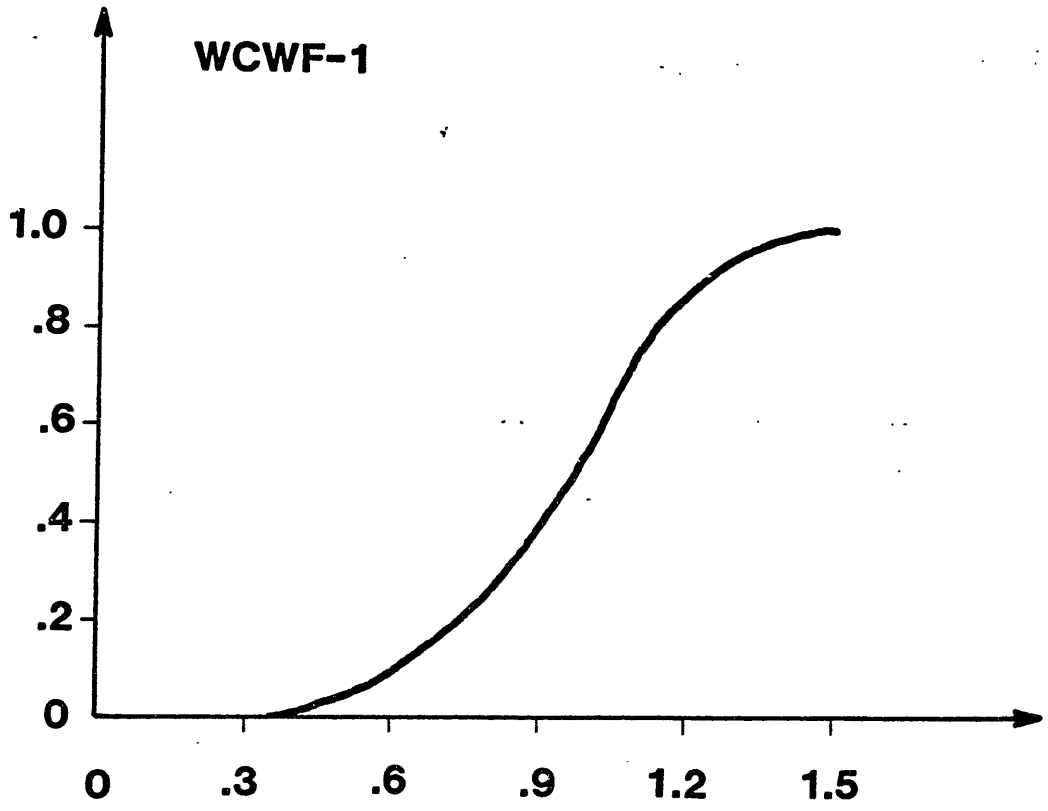


Figure III.34

schedule of the system (O'Conner, 10).

> Let's see what this meant in a recent software development for a large defense system. It was planned to have an operational lifetime of seven years and a total cost of about $1.4 billion --- or about $200 million a year worth of capability. However, a six-month delay caused a six-month delay in making the system available to the user, who thus lost about $100 million worth of needed capability --- about 50 times the direct cost of $2 million for the additional software effort (Boehm, 1973).

Because of the software industry's less than impressive track record in delivering projects on schedule, such embedded software projects are often scheduled with some "safety factor" incorporated (O'Conner, 10). For example, if some "Maximum Tolerable Completion Date" is, say, 100 days, and a 20% safety factor is used, then the project would be initially scheduled to complete in 0.80 * 100 = 80 days. If such a project starts to fall behind schedule, what would happen? We will assume the following scenario (O'Conner, 10): As long as the "Scheduled Completion Date" is comfortably below the "Maximum Tolerable Completion Date" then decisions to adjust the schedule, add more people, or do a combination of both will continue be based on the balancing of scheduling and workforce stability considerations, e.g., as captured by WCWF-1. However, as the "Scheduled Completion Date" starts approaching the "Maximum Tolerable Completion Date," pressures would develop that would override the workforce stability considerations. That is, management becomes increasingly willing to "pay any price" necessary to

avoid overshooting the "Maximum Tolerable Completion Date." And this often translates into a management that is increasingly willing to hire more people.

The development of such overriding pressures are captured through the following formulation of the "Willingness to Change Workforce Level" (WCWF),

WCWF = MAXIMUM (WCWF-1, WCWF-2)

WCWF-2, the second component of WCWF, is the table function depicted in Figure III.35. Thus, as long as "Scheduled Completion Date" is comfortably below the "Maximum Tolerable Completion Date," the value of WCWF-1 would be zero, i.e., it would have no bearing in the determination of WCWF, and consequently no bearing on the hiring decisions. When "Scheduled Completion Date" starts approaching the "Maximum Tolerable Completion Date" the value of WCWF-2 starts to gradually rise. Because such a situation would be developing towards the end of the project, the value of WCWF-1 would be probably close to zero and decreasing. Thus, as WCWF-2 exceeds the value of WCWF-1, the "Willingness to Change Workforce Level" would be totally dominated by scheduling considerations, i.e., by the desire not to overshoot the "Maximum Tolerable Completion Date."

Note that the above formulation of WCWF allows us to

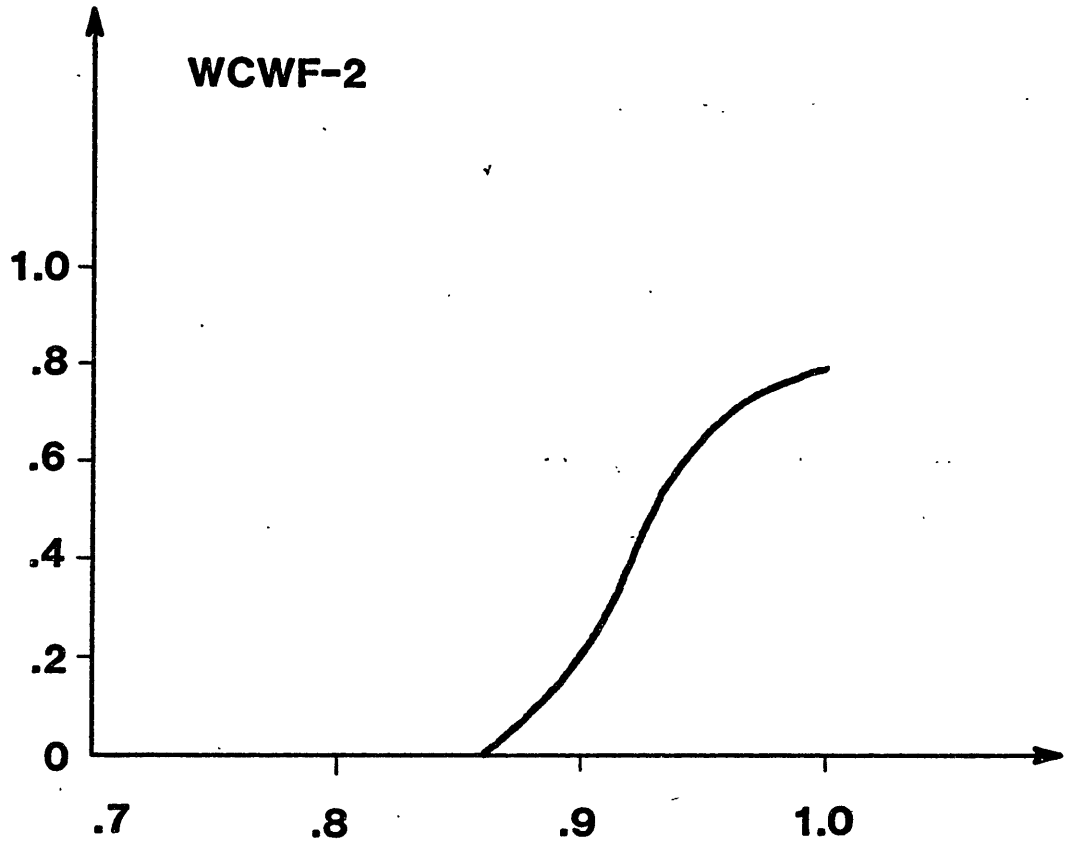


Figure III.35

easily simulate those environments in which there are no tight time commitments. In such cases we need only to set the value of "Maximum Tolerable Completion Date" to some high value. This would keep WCWF-2 always at the zero level. And, thus, WCWF becomes solely a function of WCWF-1.

One final note about the formulation of the "Willingness to Change Workforce Level." It is important to realize that the variable WCWF is an expression of a policy for managing projects. Thus, a range of functions are possible here (e.g., different forms of the table functions WCWF-1 and WCWF-2), capturing different strategies for how to balance workforce and schedule adjustments throughout the project to minimize overruns and costs. In the next chapter, we will take the opportunity to explore a range of other alternate policies besides the (representitive) one discussed here.

Once the "Workforce Level Needed" is determined, it is translated into a goal for hiring in (or transfering out) employees. This goal is termed the "Workforce Level Sought." The "Workforce Level Sought" is almost always identical to the "Workforce Level Needed." They could, however, differ. When this happens, it is usually in the early stages of the project, when the project's manpower build-up rate tends to be at its highest level. A consideration is given then, as was explained in the "Human Resource Management Subsystem," to the project's ability to absorb new people into its

organization. This factor defines, in effect, a ceiling on the number of employees sought i.e., to be hired. That is, "Workforce Level Sought" would be set to the value of "Workforce Level Needed" as long as this is less than or equal to the "Ceiling on Total Workforce." Otherwise, "Workforce Level Sought" is set to the value of the latter.

By dividing the "Man-Days Remaining" by the value of the "Workforce Level Sought" (after being adjusted if necessary to be in terms of full-time equivalent employees) we can determine the "Time Perceived Still Required." This would represent the remaining time, in working days, that is perceived to be still required to complete the project, given its current condition. Notice that by computing the "Time Perceived Still Required" in terms of the "Workforce Level Sought rather than the "Total Workforce" means that we are assuming that schedule adjustments (which would be based on this computation), are made with full awareness of the hiring decisions being made in the project. For example, if at some point as much as 1100 man-days are still remaining to complete the project, 10 full-time employees are working on it, and it has been decided to hire an additional employee (i.e., "Workforce Level Sought" is 10 + 1 = 11), then we are assuming that management would (often through a back-of-the-envelope computation) determine that the time still required is 1100 / 11 = 100 days. (Based on discussions with (Landolfi, 11), (Chan, 14), and (Lombardi,

16).)

Once the "Time Perceived Still Required" is computed, it would be added to the value of "Time" (i.e., the number of working days elapsed on the simulated project) to determine the "Indicated Completion Date." For example, if at Time = 40 days, the value of "Time Perceived Still Required" is 100 days, then the value of the "Indicated Completion Date" would be 140 days. Once this, in turn, is determined, it is used to adjust the project's formal "Scheduled Completion Date," if necessary. The "Rate of Adjusting the Schedule" has the (by now) familiar formulation,

(GOAL) - LEVEL) / ADJUSTMENT-TIME

where,

GOAL = Indicated Completion Date

LEVEL = Scheduled Completion date

ADJUSTMENT-TIME = Schedule Adjustment Time

The "Schedule Adjustment Time" is set in the model to 5 working days (i.e., one calender week) (Landolfi, 22), (Chan, 20).

III.5. Summary:

In this chapter on model development, we accomplished three tasks. First, we identified the sources of information

utilized in developing the model. As was explained in Section III.2, the model was developed on the basis of an extensive review of the literature, supplemented by 27 focused field interviews of software project managers in 5 organizations. Second, we defined the model's boundary. As was shown in Section III.3, the model focuses on the development phases of software production, extending from the beginning of the design phase of the software lifecycle, up untill the end of the system testing phase. Finally, in Section III.4, a detailed description of the model's structure was presented. The model is comprised of four sectors. At the heart of the model is the Software Production Sector, where software production activities such as coding and testing are modeled. The project management activities comprise the remaining three sectors: Planning, Human Resource Management, and Control.

THE DYNAMICS OF SOFTWARE DEVELOPMENT PROJECT MANAGEMENT: AN INTEGRATIVE SYSTEM DYNAMICS PERSPECTIVE

by

TAREK K. ABDEL-HAMID

B.Sc., CAIRO UNIVERSITY, CAIRO
(1972)

MBA, STATE UNIVERISITY OF NEW YORK, ALBANY
(1978)

Submitted to the Department of Management
in Partial Fulfillment of the
Requirements for the Degree of

DOCTOR OF PHILOSOPHY

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

January 1984

Signature of Author: Tarek K. Abdel-Hamid
Department of Management, 6 January 1984

Certified by: Stuart E. Madnick
Stuart E. Madnick, Thesis Supervisor

Accepted by:
Chairman, Department Committee on Graduate Studies

 Vol. 2

IV. A CASE-STUDY: THE NASA DE-A SOFTWARE PROJECT

In this chapter we report the results of a case-study we conducted to test the model. The objective of the case study is to examine the model's ability to reproduce the dynamic behavior patterns of a completed software project. The dynamic behavior of a set of variables pertaining to the management of the project is tracked, including: completion date estimates, man-day estimates, cost (in man-days), and workforce loading.

The case-study was conducted at the Systems Development Section of NASA's Goddard Space Flight Center (GSFC) at Greenbelt, Maryland. This organization is engaged in the development of application software that supports ground-based spacecraft attitude determination and control. The subsystems included in a typical attitude system are telemetry processing, sensor calibration, attitude computation, and maneuver planning. In the section that

follows we will provide a detailed description of one such project, namely, the DE-A project used in our case-study. This will then be followed in Section IV.2., by a discussion of model parameterization. That is, we will discuss the set of model parameters that are set to simulate the particular DE-A project environment (e.g., project size). Finally, in Section IV.3., we will simulate the DE-A project, observe its behavior, and compare it to DE-A's actual behavior patterns.

IV.1. The DE-A Project:

The basic requirements for the DE-A project were to design, implement, and test a software system that would process telemetry data and would provide definitive attitude determination as well as real-time attitude determination and control support for NASA's DE-A satellite. The DE-A satellite was designed to study the physical process of the earth's upper atmosphere, ionosphere, and magnetosphere. The overall requirements were similar to previous space mission requirements at the GSFC System Development Section (NASA, 1983).

The DE-A project was selected for the case-study by NASA. Specifically, it was selected by Frank E. McGarry, Head of the Systems Development Section of the Goddard Space Flight center, who is participating, as we are, in the NASA/MIT "Advanced Information Systems Project." The project

was selected by McGarry so as to satisfy three criteria (furnished by us): (1) to be medium in size (i.e., 16-64 KDSI); (2) recent; (3) "typical" i.e., one that would be considered as having been developed in a familiar in-house software development environment.

In the remaining part of this section we will provide a more detailed account of the nature and development history of the project. The data presented was extracted from two primary sources:

1. Interviews with Frank E. McGarry, who managed the project. Two lengthy personal interviews were conducted at the Goddard Center on August 11 and 12, 1983. These were then followed by 4 (15-minute) telephone interviews.
2. Project documentation. These included:
    * "Software Development History for Dynamics Explorer (DE) Attitude Ground Support System (AGSS)," June, 1983.
    * DE-A Resource Summary

The life cycle phases covered in this study include the design, coding, and system testing phases. Excluded from the study are the requirements definition phase and the acceptance testing phase. Both the requirements and acceptance testing phases were excluded because they both lie outside the boundary of our model. This did not pose any

complications, however. The requirements phase, it turns out, was also not included in McGarry's group's project responsibility. Requirements were, instead, the responsibility of the user organization, which for the DE-A project was the Attitude Determination and Control Section (ADCS) of the Goddard Space Flight Center. The ADCS, thus, developed the functional requirements of the system, including system input and output, algorithms, and timimg and accuracy requirements. The responsibility for the acceptance testing phase, on the other hand, was shared by both the development team, and an independent testing group. Excluding the acceptance testing phase posed no complications to our analysis simply because it was the last phase in the life cycle, hence its exclusion had no impact on any of the other life cycle phases studied.

The development and target operations machines were the IBM S/360-95 and -75. The programming language was mostly Fortran (85%). (Assembler language and assembler language macros constituted the remaining 15%.) The size of the system in Delivered Source Instructions (DSI) is 24,400 DSI. Recall the definition of a DSI:

> Delivered. This term is generally meant to exclude nondelivered support software such as test drivers. However, if there are developed with the same care as delivered software, with their own review, test plans, documentation, etc., then they should be included.
>
> Source Instructions. This term includes all program instructions created by project personnel and processed

into machine code by some combination of preprocessors, compilers, and assemblers. It excludes comment cards and unmodified utility software. It includes job control language, format statements, and data declarations. Instructions are defined as lines of code or card images. Thus, a line containing two or more source statements counts as one instruction; a five-line data declaration counts as five instructions (Boehm, 1981).

The size of the project in DSI is determined by NASA as follows (NASA, 1983):

Size in DSI = New Statements + extensively modified statements + 0.2*(Slightly Modified Statements)

Where, a "Statement" is a non-comment source instruction.

The project's actual key development dates were:

Phase	Start	End
Design	Oct. 1, 1979	May 9, 1980
Coding	May 10, 1980	March 27, 1981
Sys. Test	Nov. 15, 1980	April 24, 1981

Thus, the project was completed in 19 calendar months. In terms of cost, the project consumed 2,222 man-days of effort. (2,784 man-days were expended to complete the total project, of which 562 man-days were consumed in the acceptance testing activity.)

IV.2. Model Parameterization:

Three sets of parameters need to be set in the model, to simulate a particular project situation. These are:

Initial Project Estimates

1. Initial estimate of project size in DSI
2. Initial estimate of man-day expenditures
3. Initial staffing level
4. Initial estimate of project duration Human Resource Management

5. Average daily manpower per staff member
6. Hiring delay
7. Average employment time
8. Training overhead
9. Average assimilation delay

Software Development Environment

10. Nominal Potential Productivity
11. Error rate

We now proceed to set the DE-A project values for the above collection of model parameters.

Project Planning

1. Initial Estimate of Project Size:

As was mentioned above the actual DE-A project size was 24.4 KDSI. At the initiation of the design phase (i.e., October 1, 1979), though, the project's size was under-estimated by 45%. That is, the project was perceived to be only 24.4 * (1-.45) = 16 KDSI (NASA, 1983). (Note: Initial estimates were made in terms of source instructions

with comments. The actual size of the project in source instructions with comments was 49,500 and the initial estimate was 32,600 i.e., under-estimated by 45%.)

2. Initial Estimates of Man-Day Expenditures: In a NASA document titled Recommended Approach to Software Development (April, 1983), the following estimating guidelines are provided:

> It is important for the manager to use a model that is tuned to the specific environment and corresponds well with the resources expended for similar past projects. The Meta-Model has been developed using SEL data. However, managers must never completely rely on any formal resource estimation model. Rather, they must use the results of the model, together with historical knowledge of similar systems, to update resource and cost estimates. The new estimates are more accurate because they are based on additional information and model support.

The Meta-Model refered to above is a software estimation model developed as part of a research project of the Software Engineering Laboratory (SEL). The SEL is a research organization established in 1977 at the NASA Goddard Space Flight Center (Systems Development and Analysis Branch) in cooperation with the University of Maryland (Computer Science Department), and the Computer Sciences Corporation (Flight Systems Operation). The Meta-Model is discussed in (Bailey and Basili, 1981).

In the DE-A project, the above recommended procedure was

indeed followed ((NASA, 1983) and discussions with McGarry). That is, the Meta-Model estimates were used as guidelines, which were then adjusted on the basis of managerial experience and judgement.

For Project DE-A the initial estimates were made for the design, coding, system testing, and acceptance testing phases. The value was 1,380 man-days. Since the actual man-day expenditures (including the acceptance testing phase) were 2,784 man-days the initial estimate was 50% off the actual. Recall, though, that our model excludes the acceptance testing phase. Thus, the above 1,380 value cannot be used, and must be adjusted downwards. To do this we will make the following assumption: we will apply the man-day estimation error of 50% to the design, coding, and system testing phases of the project. For these three phases, the actual man-day expenditures were 2,222 man-days. Assuming that the effort for these three phases was under-estimated by 50% (as was the total project effort) we arrive at an initial estimate of 0.5 * 2,222 = 1,111 man-days.

This total man-day estimate is then distributed among the project's life cycle phases. In DE-A the distribution used was 85% for development (i.e., design and coding) and 15% for system testing (discussions with McGarry).

Finally, effort is also allocated to the QA activity.

The "Planned Fraction of Manpower for QA" for project DE-A is shown in Figure IV.[illegible]. (discussions with McGarry).

3. Initial Staffing Level:

The project was initialized with a staffing level of approximately 1.5 full-time equivalent employees (NASA, 1983).

4. Initial Estimate of Project Duration:

The DE-A project was initiated on October 1, 1979, and it was planned to complete (i.e., design, coding, and system testing) on January 30, 1981. (The acceptance testing phase was planned to start on January 31, 1981 and end on April 4, 1981.) That is, the project's duration (until system testing) was estimated to be 16 months, or 16 X 20 = 320 working days (NASA, 1983).

Because NASA's launch of the DE-A satellite was tied to the completion of the DE-A software, serious schedule slippage were not tolerated. Specifically, "all software was required to be accepted and frozen 90 days before launch" (NASA, 1983).

The DE-A satellite's launch date was August 3, 1981. This meant that all software was required to be accepted and frozen by May 3, 1981. And because, the acceptance testing phase was scheduled for 2 months, this meant that the

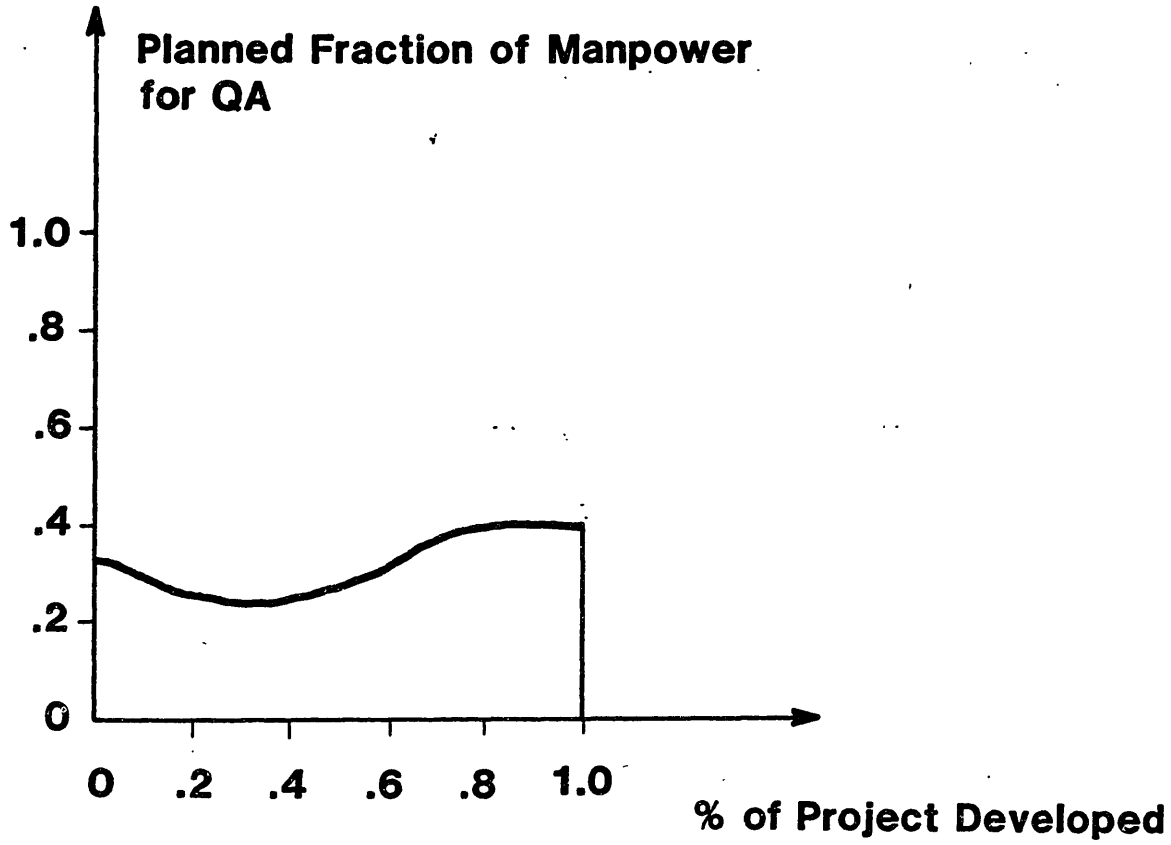

Planned Fraction of Manpower for QA
1.0
.8
.6
.4
.2
0
0
.2
.4
.6
.8
1.0
% of Project Developed

Figure IV.1

"Maximum Tolerable Completion Date" for the system testing phase was March 3, 1981. That is, the DE-A project was initially scheduled for 16 months, with the realization that it should not slip by more than 2 more months. (Note: the project ended up completing on April 24, i.e., it did overshoot the 18 month ceiling by approximately 20 calendar days. As a result the acceptance testing phase was "compressed" in duration.)

Human Resource Management

5. Average Daily Manpower per Staff:

On project DE-A the "Average Daily Manpower per Staff was set to 0.5. That is, on the average, DE-A project personnel were assigned half-time to the project (from DE-A's Resource Summary).

6. Hiring Delay:

The "Hiring Delay" was set to 30 working days i.e., 6 calendar weeks (discussion with McGarry). This is somewhat lower than the industry, average (40 days). The reason is that prompt hirings are often made from the Computer Science Corporation (CSC) (under a special arrangement between CSC and the Goddard Center).

7. Average Employment Time:

The average employment time at the Systems development Section of the GSFC is 1000 working days (i.e., 50 calendar months). This translates into a turnover rate of approximately 20% (discussions with McGarry).

8. Training Overhead:

As was explained in Chapter III, the determination of the amount of effort to commit to the training of new employees is made on the basis of managerial intuition and organizational custom. At the System Development Section, 25% of an experienced employee's time is committed per new employee (discussions with McGarry).

9. Average Assimilation Delay:

The "Average Assimilation Delay" was set for the DE-A project to 20 working days (i.e., 4 calendar weeks). This value is much lower than values reported in the literature. The reason, given by McGarry, has again to do with the special arrangement his group has with the Computer Sciences Corporation. As was said earlier, on many occasions, software professionals are recruited from CSC to work on Goddard projects. This tapped pool of software professionals is one that over the years has gained experience with the NASA project environment. And as a result, when recruited on a new project, a CSC professional is assimilated at a faster rate.

Software Development Environment

10. Nominal Potential Productivity:

Recall that this parameter captures the set of productivity determinants that distinguish different software development environments e.g., availability of software tools, computer-hardware characteristics, and product complexity. That is, the set of factors that affect productivity, but which tend to remain invariant during the life cycle of a single project.

To determine the nominal potential productivity for the DE-A project environment, we need first to determine the actual development productivity. As stated above, the total effort expended to develop the 24.4 KDSI project amounted to 2,222 man-days. Of these, 228 man-days were expended on system testing, and approximately 914 man-days on QA and rework ((NASA, 1983) and discussions with McGarry). Thus, a total of 1,080 man-days were expended on the development (i.e., design and coding) of the system. From this, we can determine the average development productivity as 24,400/1,080 = 22.59 DSI/man-day. This, however, is still not the value we are looking for. We are looking for the "Nominal Potential Productivity" and what we have is the actual productivity. Recall, potential productivity is,

> ... the maximum level of productivity that can occur when an individual or group employs its funds of

> resources to meet the task demands of a work situation. It is the level of productivity that will be attained if the individual or group makes the best possible use of its resources (that is, if there is no loss of productivity due to faulty process) (Steiner, 1966).

As was explained in detail in chapter III, actual productivity rarely equals potential productivity because of losses due to communication and motivation problems. These losses are captured in the model by the "Multiplier to Productivity due to Communication and Motivation Losses." Specifically, actual productivity is formulated in the model as the product of potential productivity and the "Multiplier to Productivity due to Communication and Motivation Losses." Thus, if we can estimate the value of this multiplier, we can then divide it into the value of actual productivity calculated above, to come up with an estimate for DE-A's "Nominal Potential Productivity."

The multiplier is itself a product of two variables, namely, the "Actual Fraction of a Man-Day on Project" and "Communication Overhead." The nominal value of the former was set in Chapter III to 0.6 (i.e., a full-time employee allocates, on the average, 60% of his or her time to productive work on the project). The "Communication Overhead," on the hand, was shown to be a function of team size. In DE-A, the size of the team size was approximately 10 people during most of the development period. From Figure III.15., we can then determine that the loss due to

"Communication Overhead" will be 60%. Thus, the value of the "Multiplier to Productivity due to Communication and Motivation Losses" becomes: 0.6 * (1-.06) = 0.564. By dividing this into the value of actual productivity (22.59 DSI/Man-Day) calculated above, we come up with the estimate for the "Nominal Potential Productivity," namely, 22.59/0.564 = 40 DSI Man-Day.

11. Error Rate:

In Chapter III, we explained that the formulation of the table function "Nominal Number of Errors Committed per Task" serves two purposes. First, its shape over the project's life reflects the relative generation rates of different error types (e.g., design versus coding errors) throughout the life of the project. These assumptions, as all others in the model, are expected to apply to all project situations modeled. Hence, this shape would remain the same, even when modeling different project situations. The second purpose of the formulation, namely, its absolute value, reflects the different error generation characteristics of different project environments i.e., the software product's characterisitics (e.g., reliability requirements), as well as those of the organization in which it is developed (e.g., quality of personnel). This, obviously, would generally change when modeling different project environments.

In the DE-A project, the actual number of errors committed was somewhere between 495 and 510 errors. (The exact figure is not known, because of "errors" in counting errors.) On the basis of this data, we formulated the "Nominal Number of Errors Committed per Task" for the DE-A project environment as shown in Figure IV.2. The shape of the curve is exactly similar to that of the base-case (shown in Figure III.17.), however, for DE-A the absolute values are slightly lower, ranging from 24 errors/KDSI at the beginning of design to 12 errors/KDSI towards the end of coding, with an average value for the project of 18 errors/KDSI.

Notice that an average nominal error rate of 18 error/KDSI would generate 18 X 24.4 = 439 errors only ... and not 495-510. This is because this error rate is the nominal rate. As was explained in Chapter III, the nominal error rate is defined to be that generated by the average experienced-type employee. Such a rate is therefore a lower bound, attained only when the workforce is solely comprised of experienced personnel. When this is not the case, i.e., when the workforce contains new hirees as well, the error rate would be adjusted upwards through the the "Multiplier to Error Generation due to Workforce Mix." By setting the nominal average error rate to 18, we are, therefore, assuming that 15% more errors (i.e., above the nominal level) will be produced) because of new hirees on the DE-A project (18 X 24.4 X 1.15 = 505 errors).

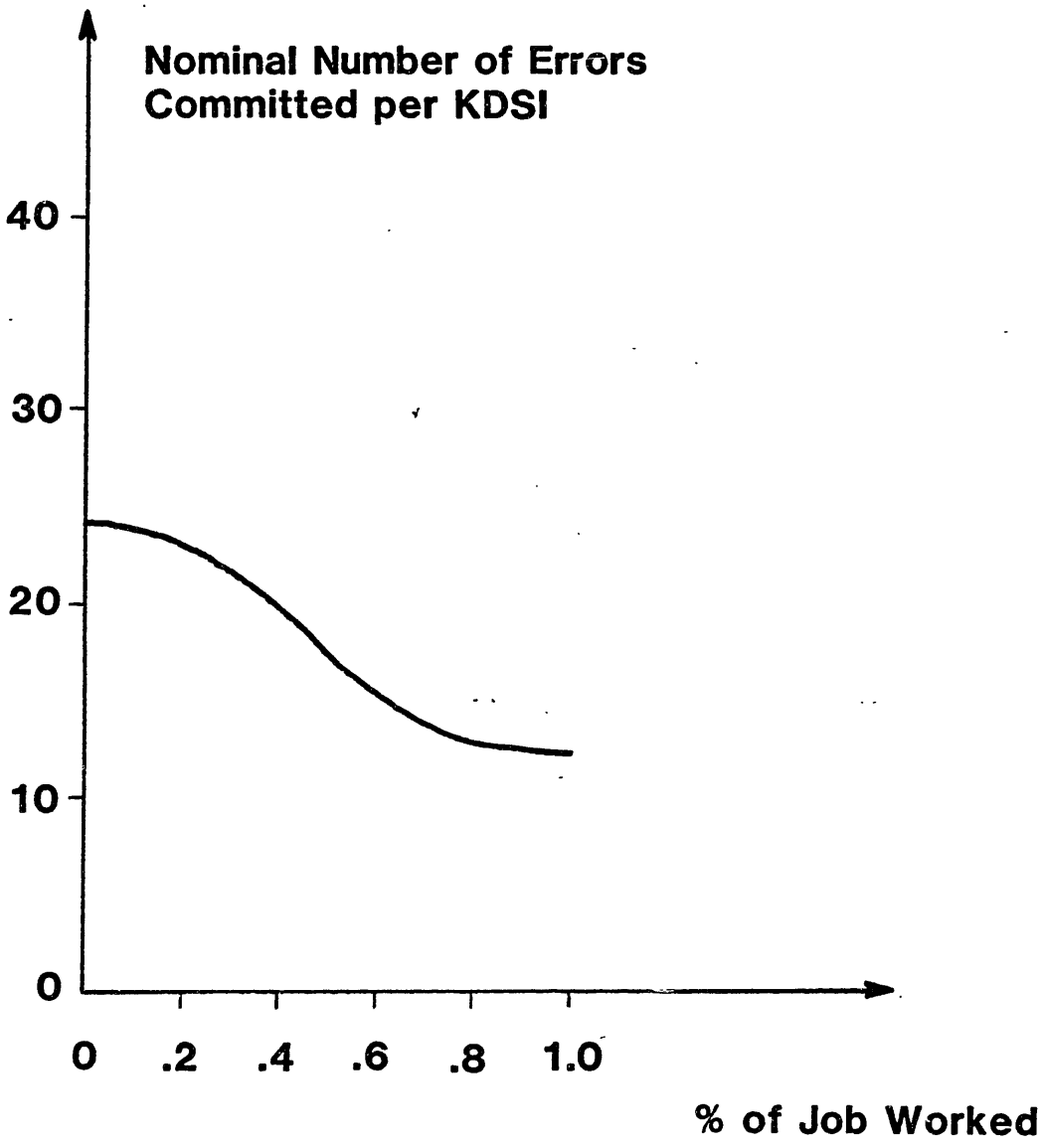

Nominal Number of Errors
Committed per KDSI
40
30
20
10
0
0
.2
.4
.6
.8
1.0
% of Job Worked

Figure IV.2

Summary

The value of the above set of model parameters is summarized in Table IV.1. The parameters are in the same order abcve, and are refered to by their DYNAMO names.

It is important to notice that the parameterization process of this section did not involve any of the model's policy formulations. By policy we mean the criteria for decision making. The set of parameters we have set merely defines the particular environment within which the policies are exercised. For example, by setting parameters such as "Hiring Delay" and "Turnover," we do not alter in any way the rationale that determines how hiring/firing decisions will be modulated through-out the project's lifecycle. Thus, while it can be determined from the set of parameter values of Table IV.1 that, for example, the DE-A project is initialized with a workforce level of 1.5 full-time-equivalent employees, one can not, on the other hand, ascertain the project's workforce loading pattern. The dynamic behavior of management systems tends to be largely a function of the interaction of the collection of policies that govern such systems (Forrester, 1979). For example, we will see in the next section how the workforce loading pattern of the DE-A software project is a function of not only the policies

	Parameter Name		Value
1.	UNDEST	(DIMENSIONLESS)	35.0
2.1	TOTMD1	(MAN-DAYS)	1,111.0
2.2	DEVPRT	(DIMENSIONLESS)	0.85
2.3	TPFMQA	(%)	.325/.29/.275/.255/.25/.275/.325/.375/.4/.4/0
3.	INUDST	(DIMENSIONLESS)	0.4
4.1	TDEV1	(DAYS)	320.0
4.2	MXSCDX	(DIMENSIONLESS)	1.16
5.	ADMPPS	(DIMENSIONLESS)	0.5
6.	HIREDY	(DAYS)	30.0
7.	AVEMPT	(DAYS)	1,000.0
8.	TRPNHR	(DIMENSIONLESS)	0.25
9.	ASIMDY	(DAYS)	20.0
10.	DSIPTK	(DSI/TASK)	40.0
11.	TNERPT	(ERRORS/KDSI)	24/22.9/20.75/15.25/13.1/12

TABLE IV.I

governing the management of the human resource, but also of the interaction between these policies and other policies such as those of project scheduling.

IV.3. Actual and Simulated Project Behavior:

Once the model was parameterized, it was run to simulate the DE-A project. In this section we discuss the model's output and compare it to DE-A's actual behavior. We will examine the dynamic behavior of the following four project variables: (1) estimated completion date; (2) estimated project man-day expenditures; (3) cumulative man-day expenditures; and (4) workforce level.

Figure IV.3. depicts how DE-A's estimated completion date and estimated total man-day espenditures changed overtime. The actual project values are shown as circles with a dot inside. The time axis is in terms of working days (a calendar month is 20 working days).

Notice that the model accurately portrays management's inclination not to adjust the project's scheduled completion date during most of the development phase of the project. Adjustments, in the earlier phases of the project, are instead made in the project's workforce level. This behavior pattern arises, according to DeMarco (1982), because of political reasons:

Portions of the text
on the following page(s)
are not legible in the
original.

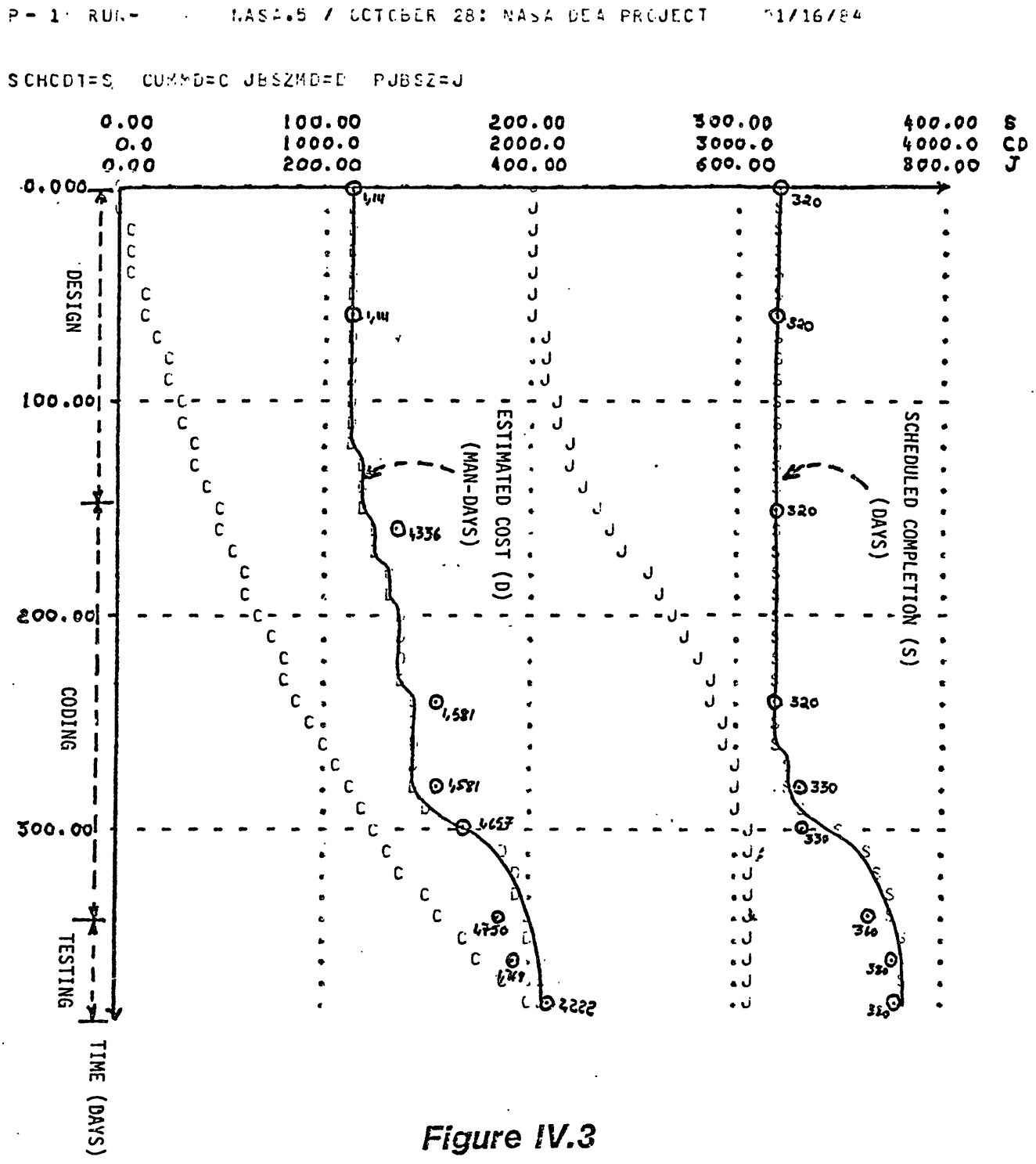

P- 1 RUN- NASA.5 / OCTOBER 28: NASA DEA PROJECT 01/16/84
SCHEDTES COMMDEC JBSZMDED PJBSZEJ
0.00 100.00 200.00 300.00 400.00 S
0.0 1000.0 2000.0 3000.0 4000.0 CD
0.00 200.00 400.00 600.00 800.00 J
0.000
100.00
200.00
300.00
DESIGN
CODING
TESTING
TIME (DAYS)
ESTIMATED COST (D)
(MAN-DAYS)
SCHEDULED COMPLETION (S)
(DAYS)
1,114
1,114
4336
1,581
4581
4657
4750
4849
2222
320
320
320
320
330
330
340
350
360

Figure IV.3

> Once an original estimate is made, its' all too tempting to pass up subsequent opportunities to estimate by simple sticking with your previous numbers. This often happens even when you know your old estimtes 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.

Notice also that adjustments in the project's man-days budget start to be made towards the end of the design phase. These adjustments are triggered as "undiscovered Job Tasks" start to be discovered (discussion with McGarry). Recall that at the project's initiation, the project was incorrectly perceived to be (only) 16 KDSI in size. The actual adjustments that were made in DE-A are, however, somewhat larger than those estimated by the model. This indicates that the visibility in the DE-A project is somewhat higher than that assumed in the model. That is, DE-A management detected more of the discrepancies between the project's actual scope, and it detected them faster. Indeed, in a post-project evaluation, the project was rated as "above average" in the area of project visibility (NASA, 1983). This was attributed to the utilization of a number of project management tools, including: librarians that maintain a central repository of the project's records, configuration analysis tools (CATs), and Unit Development Folders (UDF).

However, while the visibility in the DE-A project is

somewhat better than the industry's norm (as captured by our model), it is still, by no means, total. As a result, significant adjustments in both the project's man-days and the schedule continue to be made until the final stages of development. An outcome that the model successfully reproduces.

Notice that the model's values for the project's final man-day expenditures (2,092) is slightly lower than the actual (2,222). The primary reason for this, is that the model, while it successfully reproducess the project's manpower loading pattern (as we shall see later), it slightly under-estimates the values of the manpower level. Lower manpower levels mean lower communication and training overheads, which means a slight over-estimation of productivity.

Also, the model's project duration (387.5 days) is slightly longer than DE-A's actual (380 days). The reason for this, is again, the fact that the DE-A management behaved slightly more aggressively (than is assumed in the model) in acquiring manpower, especially during the final stages of the project. In DE-A, the workforce level at the end of the system testing phase was approximately 16 full-time equivalent people, while the model's value was 14.8. With more people at hand in the actual project, a smaller schedule overshoot was achieved.

We turn next to Figures IV.4. and IV.5., which depict the simulated and actual manpower loading patterns, respectively. For the reader's convenience, we also plotted a number of actual values alongside the simulation output.

The model accurately replicates the actual DE-A pattern. What is quite encouraging about this result is the fact that the model successfully reproduced such an "atypical" workforce loading pattern. The "typical" software project workforce pattern discussed in the literature is a concave-type curve that rises, peaks, and then drops back to lower levels as the project proceeds towards the end of the system testing phase (e.g., see (DeMarco, 1982), (Boehm, 1981), and (Albrecht, 1979).)

The reason why the workforce level shoots upwards towards the end of the project has to do with NASA's tight scheduling constraints. As explained above, because NASA's launch of the DE-A satellite was tied to the completion of the DE-A software, serious schedule slippages were not tolerated. Specifically, "all software was required to be accepted and frozen 90 days before launch" (NASA, 1983). This, in effect, defined a "Maximum Tolerable Completion Date" for the project. For the DE-A project that date was March 3, 1981 ... or day 380 from the start. As this date was approached, pressures develop that override the workforce stability considerations. That is, project management

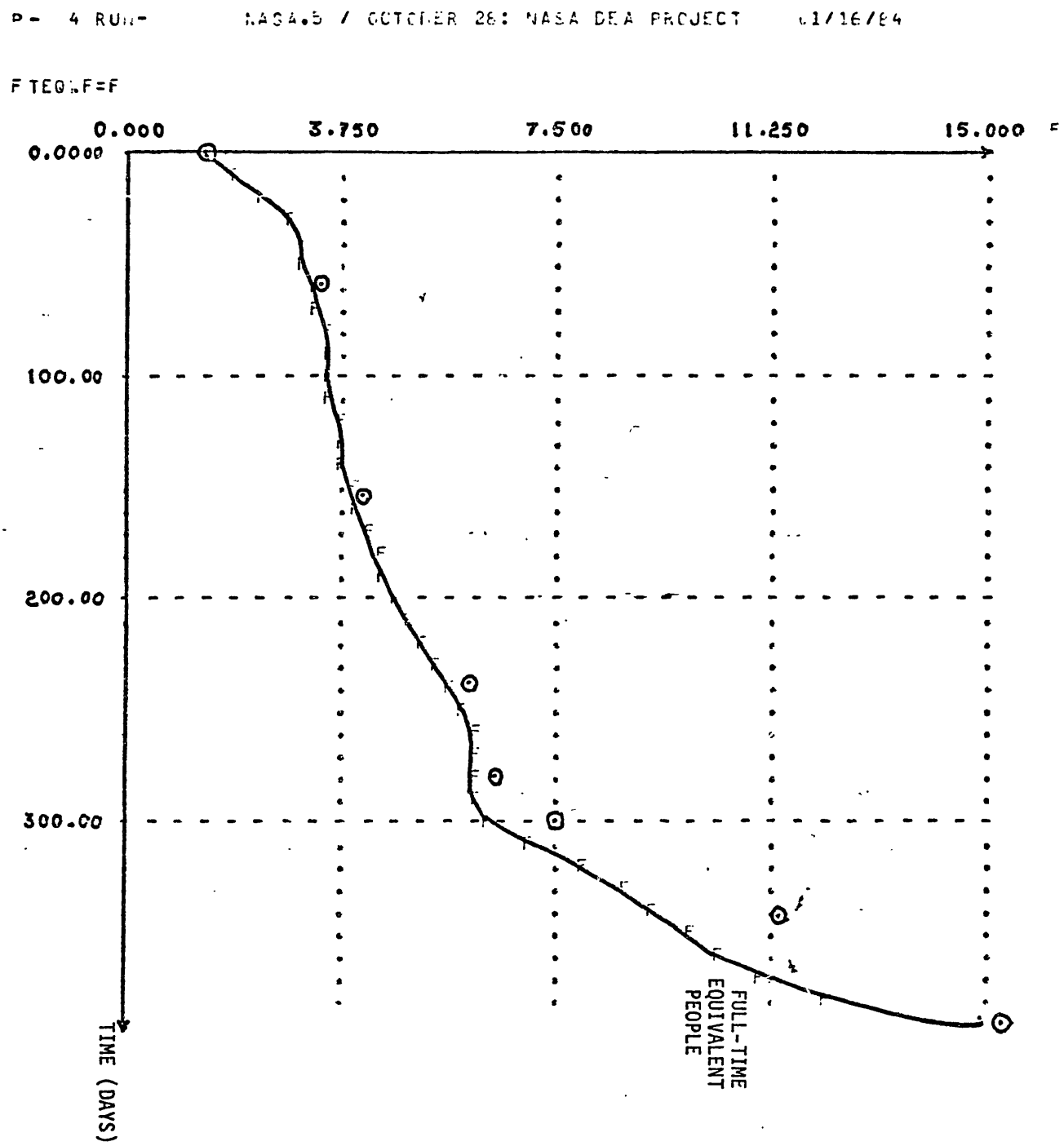

P- 4 RUN- NASA.5 / OCTOBER 28: NASA DEA PROJECT 01/16/84
FTEQ.F=F
0.000
3.750
7.500
11.250
15.000
0.0000
100.00
200.00
300.00
TIME (DAYS)
FULL-TIME EQUIVALENT PEOPLE

Figure IV.4

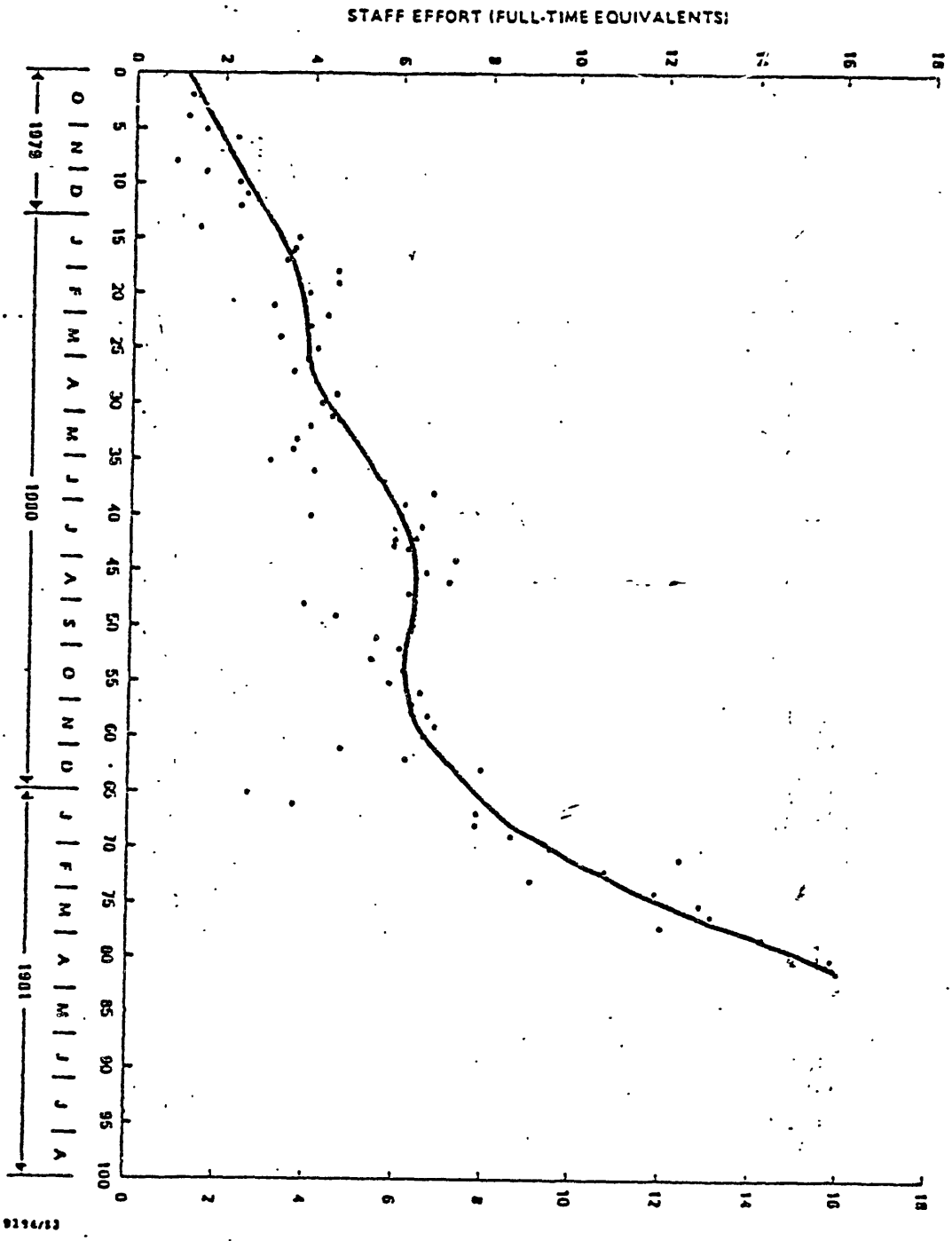

STAFF EFFORT (FULL-TIME EQUIVALENTS)
1979
1980
1981
O N D J F M A M J J A S O N D J F M A M J J A
0 5 10 15 20 25 30 35 40 45 50 55 60 65 70 75 80 85 90 95 100
0 2 4 6 8 10 12 14 16 18
0 2 4 6 8 10 12 14 16 18

Figure IV.5

becomes increasingly willing to "pay any price" necessary to avoid overshooting the "Maximum Tolerable Completion Date." And this translates, as the figures indicate, into a management that is increasingly willing to add more people.

Finally, in Figure IV.6. we plot the model's cumulative man-day expenditures, together with actual project results. Again, the model captures the exponentially increasing pattern. The actual figures are slightly higher, however, because, as we explained earlier, the model slightly under-estimates the workforce level ... especially, towards the second half of the project i.e., the DE-A management's "Willingness to Change Workforce" does not decrease as the project proceeds towards its final stages, nearly as much, as is assumed in the model.

IV.4. Conclusion:

The objective of this case-study was to test the model's ability to reproduce the dynamic behavior patterns of a completed software project, namely, the NASA DE-A software project. To do this we first parameterized the model. The process involved setting model parameters that capture the particular DE-A project environment. The parameter values were obtained from two sources, namely, interviews at NASA and project documentation. The 14 model parameters that were set, (e.g., "Hiring Delay," "Turnover Rate," ... etc.), it

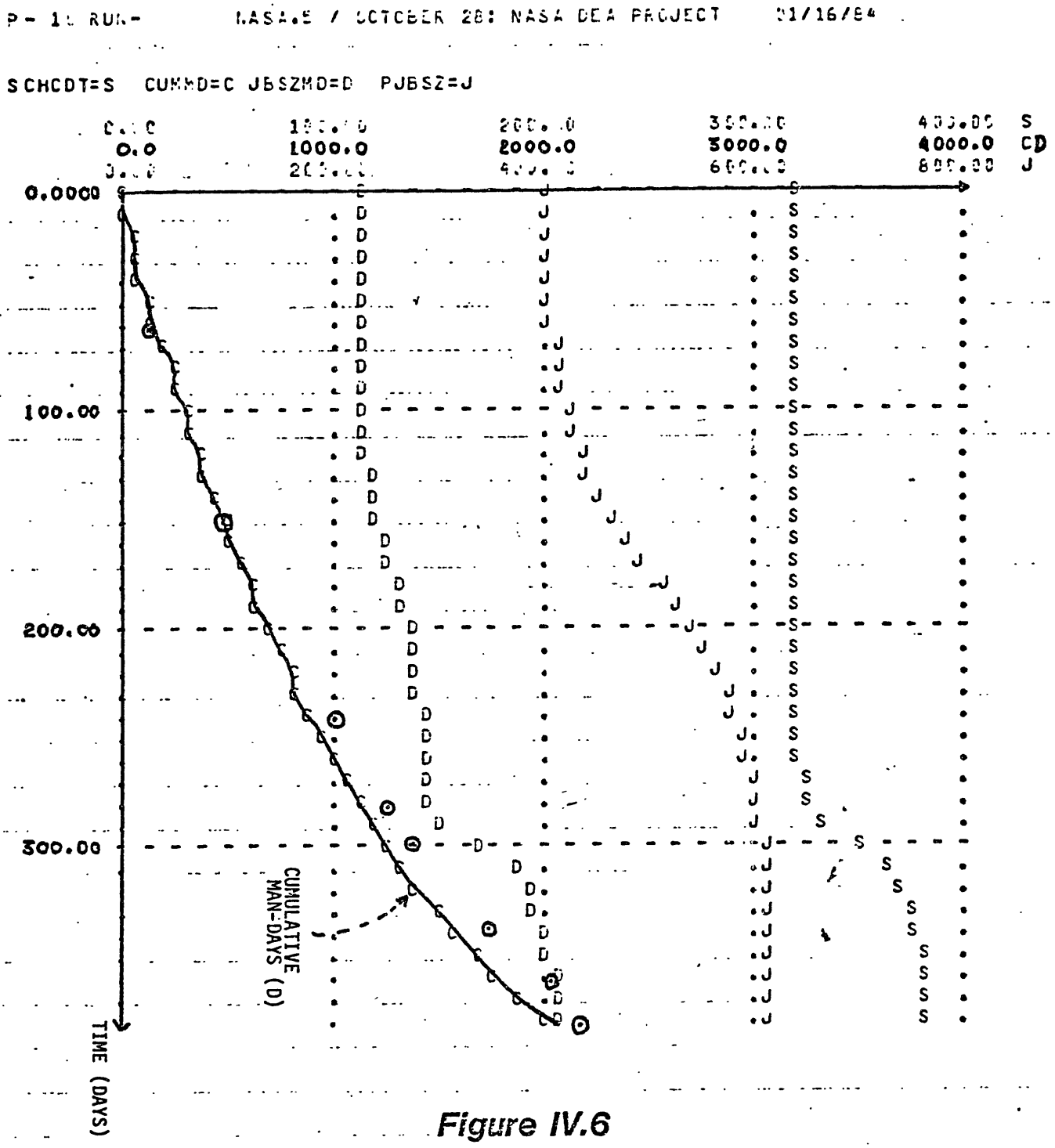

P- 1: RUN-
NASA.E / OCTOBER 28: NASA DEA PROJECT
01/16/84
SCHCDT=S CUMMD=C JBSZMD=D PJBSZ=J
0.0
1000.0
2000.0
3000.0
4000.0
S
CD
J
0.0000
100.00
200.00
300.00
CUMULATIVE MAN-DAYS (D)
TIME (DAYS)

Figure IV.6

is important to note, do not involve any changes in the formulation of the model's policy structures. The paramter set merely defines the (DE-A) environment within which the policies are exercised. This is significant, since the dynamic behavior patterns generated are largely a result of the interaction of the model's (unchanged) policy structures.

Four DE-A project variables were examined, namely, completion date estimates, man-day estimates, cost in man-days, and workforce loading. While the model was quite accurate in reproducing the project's patterns of dynamic behavior, it slightly under-estimated the absolute value of DE-A's workforce level. That is, DE-A's management was somewhat more aggressive in its manpower acquisition policy than is assumed in the model. This underestimate caused the model to slightly underestimate the project's cost in man-days (by 6%), and slightly overestimate the project's duration (by 2%).

One of the advantages of system dynamics modeling is that it not only allows us to generate the dynamic implications of a given set of policies, but it in addition allows us to go a step further and explore the implications of new and different sets of managerial policies and procedures. In the next chapter, we will take this further step, as we explore an array of managerial policies pertaining to the management of software projects. To set

the stage for such an analysis, let us explore some of the "what-if" questions that DE-A's management, having completed the project, might be interested in answering:

1. What if a different estimation tool was used? In DE-A, estimation by NASA's Meta-Model was used as a guideline, that was then adjusted on the basis of management's experience and judgement. Like NASA, a number of other software development organizations have developed other quantitative software estimation tools e.g., TRW's COCOMO model. How can the applicability of such new tools to the NASA environment be evaluated? To what extent are such models portable to the NASA environment? If not, why not? And how can the portability of new estimation models be improved?

2. What if more/less quality assurance (QA) effort was expended? In DE-A, 30-40 % of the development effort was allocated to QA activities ... a level that is significantly higher than the industry average. Is this an "optimal" allocation? How can we determine what an "optimal" allocation is? And what project and organizational factors affect such a determination?

3. What if more people were <u>not</u> added at the final stages of the project? Brooks' Law suggests that adding more people to a late project makes it later. When

would the DE-A project have completed had management resisted adding more people at DE-A's final stages?

These are some of the issues we turn to next in Chapter V.

V. MODEL BEHAVIOR:
AN ANALYSIS OF THE DYNAMICS OF SOFTWARE DEVELOPMENT

V.1. Introduction:

A system dynamics model is a laboratory tool. It allows repeated experimentation with the system, testing assumptions or altering management policies. The purpose is to gain an understanding of, and make predictions about, the dynamic implications of managerial actions, policies, and procedures.

> The most important advantage of a simulation model ... is its ability to 'play out' the dynamic consequences of a given set of assumptions in a way the human mind can do neither well nor consistently; a useful model produces scenarios which are both realistic and explainable in the policymaker's own terminology. In addition, a simulation model provides an experimental arena for discovering the sources of real-life problems and evaluating alternative policy options in relatively little time and with little cost. (Quoted from (Homer, 1983) who references (Forrester,1979) and (Forrester, 1979b).)

Using the system dynamics model as an experimentation vehicle should be particularly welcomed by the software engineering community. Several authors have "complained"

about the lack of tested "ideas" in the software engineering field (Thayer, 1979), (Weinwurm, 1970). For example Weiss (1979) commented:

> ... in software engineering it is remarkably easy to propose hypotheses and remarkably difficult to test them. Accordingly, it is useful to seek methods for testing software engineering hypotheses.

Unfortunately, controlled experiments in the area of software development tend to be costly and time consuming (Myers, 1978). Furthermore, " ... the isolation of the effect and the evaluation of impact of any given practice within a large, complex and dynamic project environment can be exceedingly difficult " (Glass, 1982).

In addition to permitting less-costly and less-time consuming experimentation, simulation models make "perfectly" controlled experiments possible, which, as the following quotation shows, addresses the difficulty expressed by Glass above:

> The effects of different assumptions and environmental factors can be tested. In the model system, unlike real systems, the effect of changing one factor can be observed while all other factors are held unchanged. Such experimentation will yield new insights into the characteristics of the system that the model represents. By using a model of a complex system, more can be learned about internal interactions than would ever be possible through manipulation of the real system. Internally, the model provides complete control of the system organizational structure, its policies, and its sensitivities to various events. Externally, a wider

range of circumstances can be generated than are apt to be observable in real life (Forrester, 1961).

In this chapter we will use our integrative system dynamics model of software project management to predict/study the dynamic implications of an array of managerial actions, policies, and procedures pertaining to the development of software. Four areas will be studied: (1) Scheduling; (2) Controlling; (3) Quality Assurance; and (4) Staffing. To set the stage for this discussion, we will first characterize, in the next section, the software project (which we will simply call EXAMPLE) to be used in our analysis.

V.2. The "EXAMPLE" Software Project:

The objective of this section is to set up the environment within which to conduct our experimentation and analysis of the dynamics of software development. To do this we will first characterize the "EXAMPLE" software project, which will serve as the prototype project for the experiments. We will then run the model to simulate the "EXAMPLE" project, and observe its dynamic behavior. The behavior of a number of significant project variables (e.g., workforce level, schedule completion time, errors, productivity, ... etc.) will be analyzed and explained. And we will demonstrate that the model's behavior patterns do replicate those reported in the literature. Once this is

done, we will then move on to Sections V.3. through V.6., to study the dynamic implications of an array of managerial actions, policies, and procedures pertaining to the software development environment.

We will define the "EXAMPLE" software project to be 64,000 DSI in size. DSI stands for "Delivered Source Instructions." These are defined as follows (Boehm, 1981):

> Delivered. This term is generally meant to exclude nondelivered support software such as test drivers. However, if these are developed with the same care as delivered software, with their own review, test plans, documentation, etc., then they should be included.
>
> Source Instructions. This term includes all program instructions created by project personnel and processed into machine code by some combination of preprocessors, compilers, and assemblers. It excludes comment cards and unmodified utility software. It includes job control language, format statements, and data declarations. Instructions are defined as lines of code or card images. Thus, a line containing two or more source statements counts as one instruction; a five-line data declaration counts as five instructions.

Recall that in Chapter III, productivity was defined, not in terms of DSI/Man-Day, but in terms of Tasks/Man-Day. And it was explained then, that the notion of a "Task" is tied to that of "Nominal Potential Productivity." Specifically, we indicated that "Task" is a unit for sizing up a software project, that it is defined in terms of a number of DSI, and that its value, for a particular simulation, would be set to the numerical value of "Nominal Potential Productivity," when the latter is expressed in

terms of DSI/Man-Days. For example, if "Nominal Potential Productivity," for a particular project situation, is, say, 50 DSI/Man-Day, then the value of "Task" would be set, for that particular project situation, to 50 DSI. This would then allow us to maintain the value of "Nominal Potential Productivity" to 1 Task/Man-Day, for all project situations.

Let us provide an example to further clarify the concepts of "Nominal Potential Productivity" and "Task." Assume two different software development organizations, ORG-1 and ORG-2, have just completed the development (i.e., design and coding) of a software project. The two projects, PROJ-1 and PROJ-2, are 8000 DSI in size. Now, let us assume that in ORG-1 the development effort consumed a total of 400 man-days to design and code the 8000 DSI PROJ-1, while in PROJ-2 the development effort was 200 man-days. If, for purposes of simplification, we disregard the communication and motivation losses in both organizations i.e., assume that actual productivity = potential productivity, we could then conclude that the potential productivity in ORG-1 is half that of ORG-2. (This productivity differential can be due to a number of differences between the two organizations, such as differences in the availability of software tools, personnel capability, computer-hardware characteristics, ... etc.) This productivity differential would be realized in the model as follows: The "Nominal Potential Productivity" parameter would be defined in both runs of the model at the

same value, namely, 1 Task/Man-Day, but in the PROJ-1 run we would define a Task to be 20 DSI, while in the PROJ-2 run a "Task" would be set at 40 DSI. That is, the 8000 DSI project PROJ-1 will be defined in the first run as a 400 Task project, while the 8000 DSI project PROJ-2 would be defined as a 200 Task project.

To determine the value of "Task" in the EXAMPLE project we need to do the following: First, select some project environment; second, determine the value of "Nominal Potential Productivity" in terms of DSI/Man-Day for that environment; and finally set the value of "Task" to the numerical value of "Nominal Potential Productivity."

There aren't many project environments that are adequately characterized in the literature. One exception is Barry Boehm's excellent book titled Software Engineering Economics, which provides a wealth of data on the software production environment at TRW. To maintain consistency, the EXAMPLE project will be characterized totally on the basis of this TRW data. In particular, we will draw upon Boehm's data on the set of projects he described as "the most common type of software project: the small-to-medium size (project) developed in a familiar, in-house, organic software development environment" (Boehm, 1981).

For a 64,000 DSI project, Boehm's data indicate that

overall project productivity would be approximately 338.4 DSI/Man-Month. This value is arrived at by dividing the project's size in DSI by the total effort expended e.g., to develop, QA, rework, and test the software. Boehm's data also indicates that system testing would consume approximately 22% of the total effort, while the effort expended on QA activities would be in the range of 15 - 20% of the total effort. No explicit estimates are given, however, for the effort to rework errors during development. If we assume that this rework effort will be approximately 10% of total effort, then QA, rework, and testing activities would together constitute approximately 50% of the project's man-months. (Note: Boehm's data covers the design, coding, and system testing stages of software production, as does our model.) This means that the amount of effort expended on developing the product (e.g., designing and coding it) is half the total man-days expended on the project. Which in turn means that the development productivity would be 2 * 338.4 = 676.8 DSI/Man-Month. To translate this into DSI/Man-Day we divide by 20, and get 33.84 DSI/Man-Day. This, still, is not the value we are looking for. We are looking for the "Nominal Potential Productivity" and what we have is an estimate for the actual productivity. Recall, potential productivity is,

> ... the maximum level of productivity that can occur when an individual or group employs its funds of resources to meet the task demands of a work situation. It is the level of productivity that will be attained if

> the individual or group makes the best possible use of its resources (that is, if there is no loss of productivity due to faulty process) (Steiner, 1966).

As was explained in detail in Chapter III, actual productivity rarely equals potential productivity because of losses due to communication and motivation problems. These losses are captured in the model by the "Multiplier to Productivity due to Communication and Motivation Losses." Specifically, actual productivity is formulated in the model as the product of potential productivity and the "Multiplier to Productivity due to Communication and Motivation Losses." Thus, if we can estimate the value of this multiplier, we can then divide it into the value of actual productivity calculated above, to come up with an estimate for EXAMPLE's "Nominal Potential Productivity."

The multiplier is itself a product of two variables, namely, the "Actual Fraction of a Man-Day on Project" and "Communication Overhead." The nominal value of the former was set in Chapter III to 0.6 (i.e., a full-time employee allocates, on the average, 60% of his or her time to productive work on the project). The "Communication Overhead," on the other hand, was shown to be a function of team size. Again, referring to Boehm's results, we find his estimate for the "average staffing level" for the 64,000 DSI project, to be approximately 10 people. From Figure III.15. we can then determine that the loss due to "Communication

Overhead" will be 6%. Thus, the value of the "Multiplier to Productivity due to Communication and Motivation Losses" becomes: 0.6 * (1-.06) = 0.564. By dividing this into the value of actual productivity (33.84 DSI/Man-Day) calculated above, we come up with the estimate for the "Nominal Potential Productivity," namely, 33.84 / 0.564 = 60 DSI Man-Day.

We said there were three steps to determine the value of "Task" in the EXAMPLE project. The third and final step, is to set the value of "Task" to the numerical value of "Nominal Potential Productivity" when the latter is expressed in terms of DSI/Man-Day. Thus, for the project EXAMPLE, Task is defined to be 60 DSI. (Which, therefore, allows us to maintain "Nominal Potential Productivity" as being 1 Task/Man-Day.)

Thus far we have first defined the real size of the project EXAMPLE to be 64,000 DSI and second, by defining what constitutes a Task we have also (implicitly) defined the project's environment. When any project is initialized, managerial decisions are made on how much manpower and time to allocate to the project. Such decisions are obviously important determinants of how the project will develop. For the project EXAMPLE we must do the same, i.e., initialize its manpower and schedule allocation variables.

As was stated earlier, in order to maintain consistency in our characterization of the project EXAMPLE, we will characterize it totally on the basis of Boehm's TRW data. In calculating EXAMPLE's development effort, schedule, and staffing level, we will, therefore, use Boehm's COCOMO model. COCOMO stands for the COnstructive COst MOdel, and is a software project estimation model that has been developed and is being used by TRW. COCOMO exists in a hierarchy of increasingly detailed forms. In our analysis we will use the version called "Basic COCOMO," and which, according to Boehm (1981) is "the version applicable to the large majority of software projects: small-to-medium size (projects) developed in a familiar in-house software development environment."

The development period covered by COCOMO estimates begins at the beginning of the product design phase (successful completion of a software requirements review) and ends up at the end of the system testing phase, as does our model. The primary input to COCOMO is the perceived size of the project in KDSI (i.e., thousand delivered source instructions). Notice that it is the perceived not the real size of the project that is input to COCOMO to derive the estimates, since at the beginning of development (when the estimates are made) the real size of the project is often not known.

> As with other computer-based models, (COCOMO) is a 'garbage in-garbage out' device: if you put poor sizing

> (data) in one side, you will receive poor cost estimates out the other side (Boehm, 1981).

Boehm further suggests (we assume on the basis of his TRW experience) that "The software undersizing problem is our most critical road block to accurate software cost estimation." This is substantiated by the experiences of several other authors (DeMarco, 1982), (Burchett, 1982), (Daly, 1977), (Devenny, 1976). A major cause for this undersizing problem is,

> ... (the) powerful tendency to focus on the highly visible mainline components of the software, and to underestimate or completely miss the unobtrusive components (e.g., help message processing, error processing, and moving data around) (Boehm,1981).

But how much undersizing? There is, obviously, a wide range of "reasonable possibilities." For the project "EXAMPLE" we will assume that management (at the beginning of design) underestimates the project's size by a factor of 1.5. This value was, again, chosen to conform to Boehm's estimates (Boehm, 1981). That is, a project of size N (KDSI) would be incorrectly perceived as being only 0.67N (KDSI) in size. In terms of our EXAMPLE software project, this means that the project would be perceived (at the beginning of the simulation run) as being only 0.67 * 64 = 42.88 KDSI in size.

In other words, we will assume that as the project EXAMPLE is initialized, project management's perception of

the project's size will (incorrectly) be 42.88 KDSI. This value then becomes the input that management uses in COCOMO's effort and schedule estimation equations.

The COCOMO equation for the number of man-days (MD) to develop and test the project is:

$$MD = 2.4 * 19 * (KDSI)^{1.05}$$

Substituting for the EXAMPLE project we get,

$$MD = 2.4 * 19 * (42.88)^{1.05}$$

$$= 2,359 \text{ man-days}$$

This represents the total man-days to develop and test the software product. For planning purposes, this effort is then distributed among the project's life cycle phases. In our model there are two explicit phases, namely, development (which includes design and coding) and testing. So, how much would management allocate to development versus testing in our EXAMPLE project? Boehm provides a number of phase distribution guidelines. He notes (1981):

> The phase distribution varies as a function of size of the product. Larger software projects require relatively more time and effort to perform integration and test activities ...

For a 42 KDSI project (which is what EXAMPLE is perceived as being) a development to testing distribution of 80 to 20% is suggested (Boehm, 1981). That is, we will

initialize project EXAMPLE with the following allocation of man-days:

MD for Development = 0.8 * 2,359
= 1887 man-days

MD for Testing = 0.2 * 2,359
= 471 man-days

In addition to estimating the project's man-day requirements, management also estimates the project's development time and the staffing level.

The COCOMO equation for the development time (TDEV) is:

$$TDEV = 47.5 * (MD/19)^{0.38} \text{ days}$$

Substituting for the value of man-days (MD), we get

$$TDEV = 47.5 * (2,359/19)^{0.38}$$
$$= 296 \text{ days}$$

Finally, the average staffing level (ASL) is determined by dividing the estimated value of the total man-days (MD), by the estimated value of the development time (TDEV). Thus, for project EXAMPLE we get

ASL = MD / TDEV
= 2,359 / 296
= 8 full-time-equivalent software personnel

We will assume that, on project EXAMPLE, project members will be working full-time on the project. That is, the

model's parameter "Average Daily Manpower per Staff" would be set to 1 man-day. Thus, the average staffing level calculated above would be 8 actual software personnel. Not all 8 personnel will be on-board, however, at the beginning of the project. Most software projects start with a smaller core of designers, and as the project develops, the workforce slowly builds up to higher levels. For project EXAMPLE, we will assume that the project starts with a workforce level equal to half the "Average Staffing Level," i.e., with 0.5 * 8 = 4 software personnel on board. (Again, this initial staffing level is based on the results reported in (Boehm, 1981).)

With the above accomplished, our model initialization procedure is complete. Next, we run the model to simulate project EXAMPLE, and observe its behavior. The remaining part of this section will be devoted to a discussion of the model's results. The following will be discussed:

* Project progress
* Manpower distribution
* Work intensity

Project Progress:

Six key measures of progress are depicted in Figure V.1, namely, cumulative tasks developed (i.e., designed and

coded), cumulative tasks tested, cumulative man-days, the perceived job size in tasks, the perceived job size in man-days, and the scheduled completion date in days. And in Table V.1., the project's "Vital Statistics" are shown.

As was mentioned above, at EXAMPLE's initiation, its size is underestimated by a factor of 1.5. That is, instead of being perceived as being a 64,000 DSI project, it would be perceived as being only 42,880 DSI. In terms of "Tasks" (where a "Task" is 60 DSI), the project would be perceived at its initiation as being only 714.6 tasks in size, rather than 1,067 tasks ... its true size. As we have already mentioned, the undersizing problem is largely due to the tendency to underestimate the size of the unobtrusive components of the software system e.g., help message processing, error processing, support software, ... etc. As the project develops, such "Undiscovered Job Tasks" are progressively discovered as the " ... level of knowledge we have of what the software is intended to do (increases)" (Boehm, 1981). Notice, though, that the rate at which the "Undiscovered Job Tasks" are discovered remains low for a significant portion of the development phase, before it starts to accelerate rapidly. (Such behavior was also observed in the NASA case study.) The early phase of development constitutes the architectural design phase of the project. In the architectural design phase, the emphasis is on determining the overall structure of the system,

Portions of the text
on the following page(s)
are not legible in the
original.

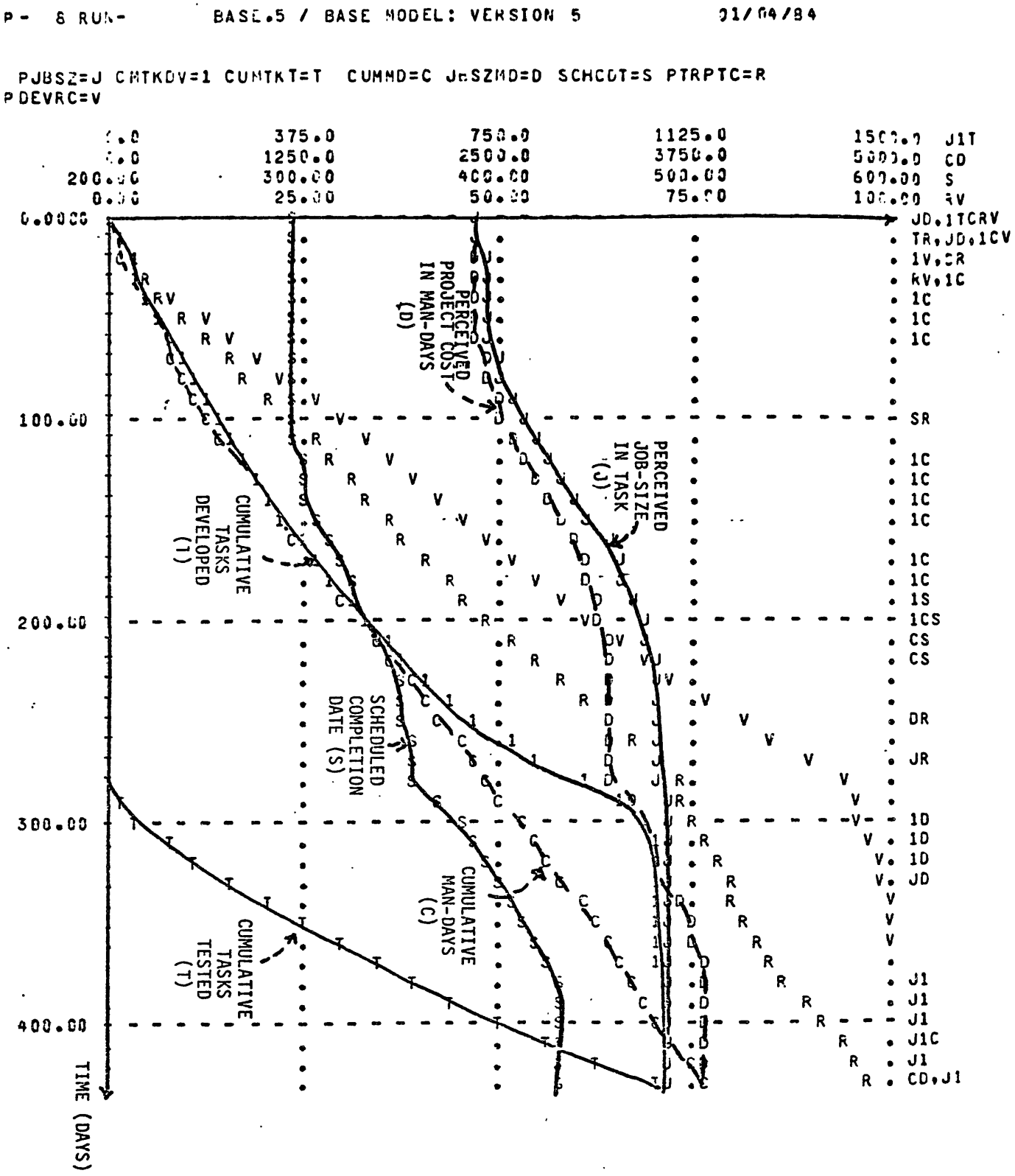

P - 6 RUN- BASE.5 / BASE MODEL: VERSION 5 01/04/84
PJBSZ=J CMTKDV=1 CUMTKT=T CUMMD=C JbSZMD=D SCHCOT=S PTRPTC=R
PDEVRC=V
375.0
750.0
1125.0
1500.0
1250.0
2500.0
3750.0
5000.0
300.00
400.00
500.00
600.00
25.00
50.00
75.00
100.00
JD,1TCRV
TR,JD,1CV
1V,CR
RV,1C
SR
CS
DR
JR
1D
JD
J1
J1C
CD,J1
100.00
200.00
300.00
400.00
TIME (DAYS)
CUMULATIVE TASKS DEVELOPED (T)
PERCEIVED PROJECT COST IN MAN-DAYS (D)
PERCEIVED JOB-SIZE IN TASK (J)
SCHEDULED COMPLETION DATE (S)
CUMULATIVE MAN-DAYS (C)
CUMULATIVE TASKS TESTED (T)

Figure V.1

1. Project Size = 64,000 DSI

2. Man-Days

Total	=	3,795	man-days
Development	=	2,681	"
Coding + Design	=	1,782	"
QA	=	380	"
Rework	=	519	"
Testing	=	1,114	"

3. Completion Time = 430 working-days

4. Errors

Total Error Generated = 1,494 → 23 Error/KDSI

Total Error Caught

During Development = 728 → 49% of Error Generated

TABLE V.I

decomposing the system into its major components, and specifing the interfaces between the components (Gagliardi, 1980). At that level, implementation details such as help message processing or error processing would (still) not be visible. And thus the rate of discovering such "Undiscovered (Unobtrusive) Job Tasks" remains low. The rate, however, starts to accelerate rapidly as the project work moves into the detailed design phase, where the emphasis is on the selection, evaluation, and design of the implementation algorithms (Gagliardi, 1980).

As the additional tasks are discovered i.e., as project members start realizing that the project's scope is larger than what has been expected, adjustments are made in the project's plan to accomodate the additional work load. As Figure V.1 indicates, both the "Job's Size in Man-Days" and the "Scheduled Completion Date" are adjusted upwards. There are, however, two interesting observations about these adjustments. Firstly, the adjustments prove to be inadequate to fully accomodate the additional work load, and secondly, the first adjustment to the schedule lags considerably behind the first adjustment to the man-days.

The additions to the project's man-days and schedule that are triggered explicitly by the discovery of new tasks level off at approximately day 200 when almost all the "Undiscovered Job Tasks" have been discovered. As shown in

Figure V.1., at approximately day 200 the value of perceived job size levels off at 1,067 tasks ... the true size of the project. At that point, the project's size in man-days plateaus at a value of 3,200 man-days. However, notice that while the perceived job size remains unchanged after day 200, further significant additions are made to the project's man-days and its schedule. These further adjustments are not triggered by the discovery of further "additional tasks" (since none are discovered after day 250). Their direct cause (as will be explained in more detail later) is the realization at approximately day 300 that the project is behind schedule i.e., that the "Total Man-Days Reported Still Needed" to complete the project is more than "Man-Days Remaining" in the project's plan. (Such a shortage in man-days can, of course, arise even if the project's size had not been underestimated. For example, it would arise if management overestimates its staff's productivity, and as a result does not allocate enough man-days to the project.) In this case, however, the man-day shortage problem is largely the indirect result of the project's undersizing problem. What happens is that when additional tasks are discovered in a software project (as they do up until day 200 in EXAMPLE), the additions that are made in the project's man-days to accomodate those additional tasks are often not quite sufficient. The reason being that some of the discovered tasks are absorbed by the project members without any formal adjustments to the project's plans. Only if the additional

tasks are perceived as requiring a relatively significant amount of effort to handle, would project members "bother" to go through the trouble of formally developing cost estimates and incorporating them in the project's work plan.

Thus, by day 200 when almost all the "Undiscovered Job Tasks" have been discovered i.e., when the value of the perceived job size attains the job's true size of 1,067 tasks, the value to which the "Job's Size in Man-Days" would be raised, namely, 3,200 man-days, would not be high enough to accomodate all the additional tasks. An interesting comparison to make, and one which would provide us with some feel of how much higher the man-day level should have been raised, is to compare the 3,200 man-day value (which is supposedly enough to develop a 1,067 task product) to the number of man-days that would be allocated to a project perceived <u>from the start</u> as being 1,067 tasks (i.e., 64 KDSI) in size. To do this we use COCOMO's man-days (MD) equation,

$$\begin{aligned} MD &= 2.4 * 19 * (KDSI)^{1.05} \\ &= 2.4 * 19 * (64)^{1.05} \\ &= 3{,}593 \text{ man-days} \end{aligned}$$

Thus, increasing the number of man-days allocated to the project from 2,359 (at the beginning of EXAMPLE) to only 3,200 (by day 200, when almost all the undiscovered tasks are discovered), falls approximately 400 man-days short of the above 3,593 man-days benchmark, a significant deficit in the project's man-days budget.

When and how is this man-days deficit handled? It is handled when it becomes visible. This usually happens (as was explained in Chapter III) towards the later stages of development when the development work is almost finished and/or when the allocated man-days budget is almost used up. Once visible, the man-days deficit would be handled by overworking (e.g., the staff members work overtime hours), and/or adjusting the project's man-days budget upwards. Both of these responses take place in project EXAMPLE, and will be discussed in some detail later. (Notice, though, that the latter response i.e., adjusting the man-day budget, is evident in Figure V.1., as the "Job's Size in Man-Days" makes a significant upward adjustment at around day 300.)

The second interesting observation about Figure V.1. concerns the adjustments made in the schedule completion date. Notice that the first adjustment to the schedule lags considerably behind the first adjustment to the man-days. (Such behavior was also observed in the NASA case study.) Specifically, the first adjustment to the "Job's Size in Man-Days" is made around day 80, whereas the first adjustment to the schedule is made 60 days later, around day 140. Why?

When the "Job's Size in Man-Days" is adjusted upwards, it is done by adjusting the men, the days, or both. That is, it is done by adjusting the project's workforce level, the project's schedule completion date or both. As was explained

in Chapter III. the decision on which alternative to choose is really an expression of management's policy on how to balance workforce and schedule adjustments throughout the project. (A number of different policies will be explored later in this chapter.) In general, though, the decision is a function of the project's stage of completion. In the earlier stages of the project, project managers are generally willing to make any necessary adjustments to the workforce level to maintain the project on its scheduled course. However, as the project proceeds, management becomes increasingly reluctant to add new people to the project, as consideration is increasingly given to the stability of the workforce. As this happens, any additions to the project's man-days get absorbed, not only through adjustments to the project's workforce level, but, in addition, they get absorbed in part by adjustments to the schedule. This shift away from workforce adjustments to schedule adjustments continues as the project progresses.

With this in mind we can now refer back to Figure V.1., and explain why the first adjustment to project EXAMPLE's schedule completion date lags considerably behind the first adjustment to the man-days level. Notice that the first adjustment to the project's man-days is made at day 80. At that relatively early point, the additional man-days are absorbed totally by adding more people to the project, rather than by changing the schedule. This can be clearly seen in

Figure V.2. The figure depicts EXAMPLE's manpower level for the project's full life cycle. And it also shows curve (*), which depicts what the manpower level of EXAMPLE would have been like (for the first 150 days) if none of the "Undiscovered Job Tasks" were ever discovered. Notice that the two curves coincide up until day 100 i.e., approximately 40 days after "Undiscovered Job Tasks" are first discovered in EXAMPLE. This 40 day delay constitutes the "Hiring Delay" in project EXAMPLE.

As explained above, as the project proceeds management becomes increasingly reluctant to add new people to the project. As this happens, any additions to the project's man-days get absorbed, not only through adjustments to the project's workforce level, but, in addition, they get absorbed in part by adjustments to the schedule. Thus, as EXAMPLE's man-days level continues to escalate (as a result of the continued discovery of new tasks), the point is reached (at around day 140) when the project's schedule starts absorbing part of the newly added man-days load. Notice that the rate at which the schedule is adjusted upwards remains low at first, as most of the emphasis is still on adjusting the workforce level. However, as the project proceeds, the emphasis shifts away from workforce adjustments, and towards schedule adjustments. The result of this shift is clearly reflected in the much faster rate at which the project's schedule completion date is adjusted

upwards during the second set of adjustments in the project's man-days i.e., that start at around day 300. Notice also that during this second adjustment process, adjustments in the project's scheduled completion date do not lag behind the adjustments in man-days i.e., that both start around day 300.

Manpower Distribution:

In this section we will discuss, not one, but two manpower distributions. The first and foremost is, of course, the manpower distribution of project EXAMPLE. This is depicted in Figure V.2. The shape of EXAMPLE's manpower distribution curve shown in the figure conforms well with manpower distributions reported in the literature (e.g., see (DeMarco, 1982), (Boehm, 1981), and (Basili and Zelkowitz, 1979)). For example, Figure V.3. represents the manpower distribution at IBM's DP Services organization reported in (Albrecht, 1979).

The second workforce distribution we would like to comment on is the one we encountered in the NASA case study. For the reader's convenience, the simulated and the actual NASA workforce distributions are included below in Figures V.4. and V.5. What is interesting about the NASA workforce distribution is its non-conformance to the "typical" workforce patterns discussed in the literature. And it is quite encouraging that the model has proved capable of

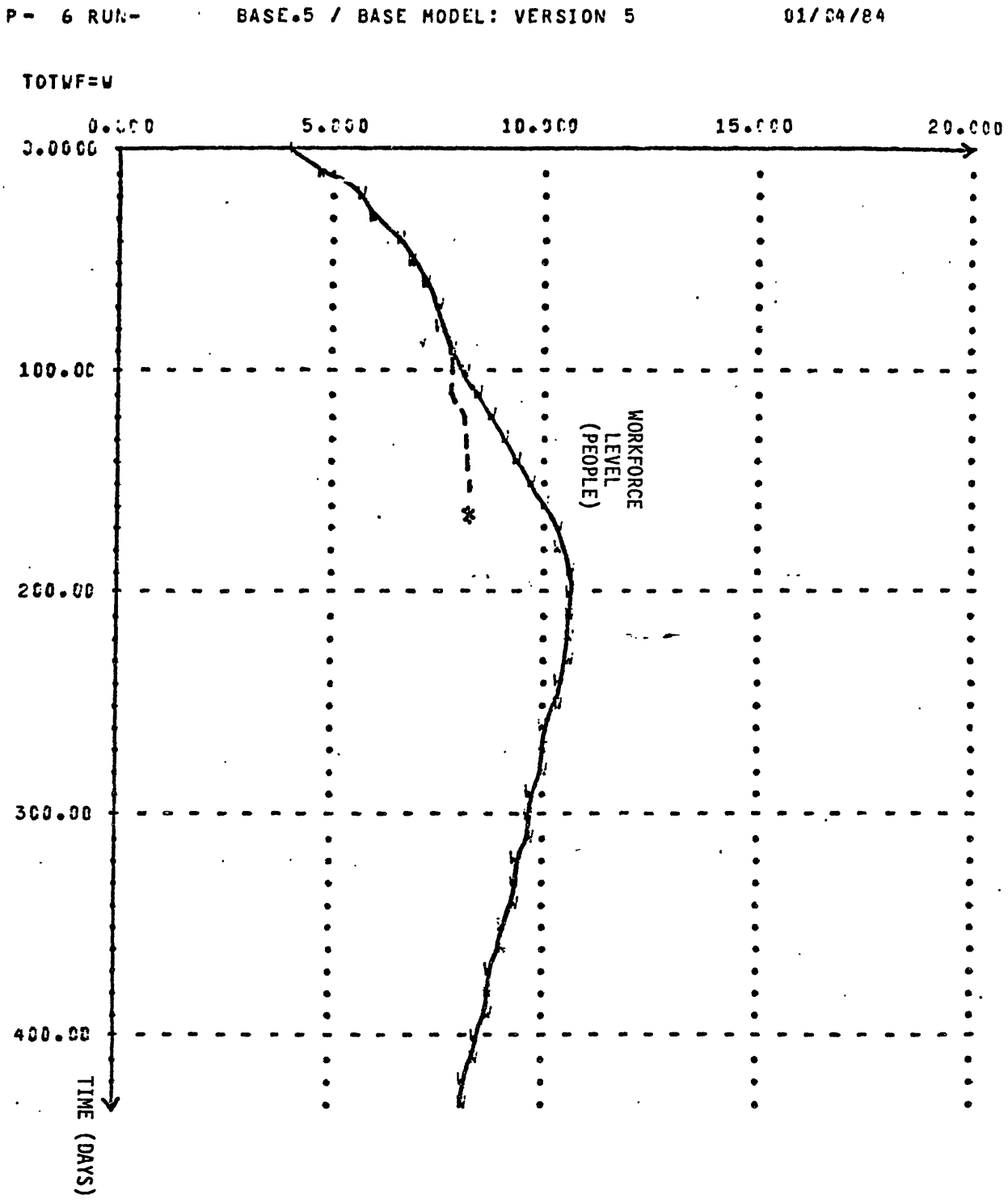

P= 6 RUN= BASE.5 / BASE MODEL: VERSION 5 01/04/84
TOTWF=W
0.000
5.000
10.000
15.000
20.000
0.00000
100.00
200.00
300.00
400.00
WORKFORCE LEVEL (PEOPLE)
TIME (DAYS)

Figure V.2

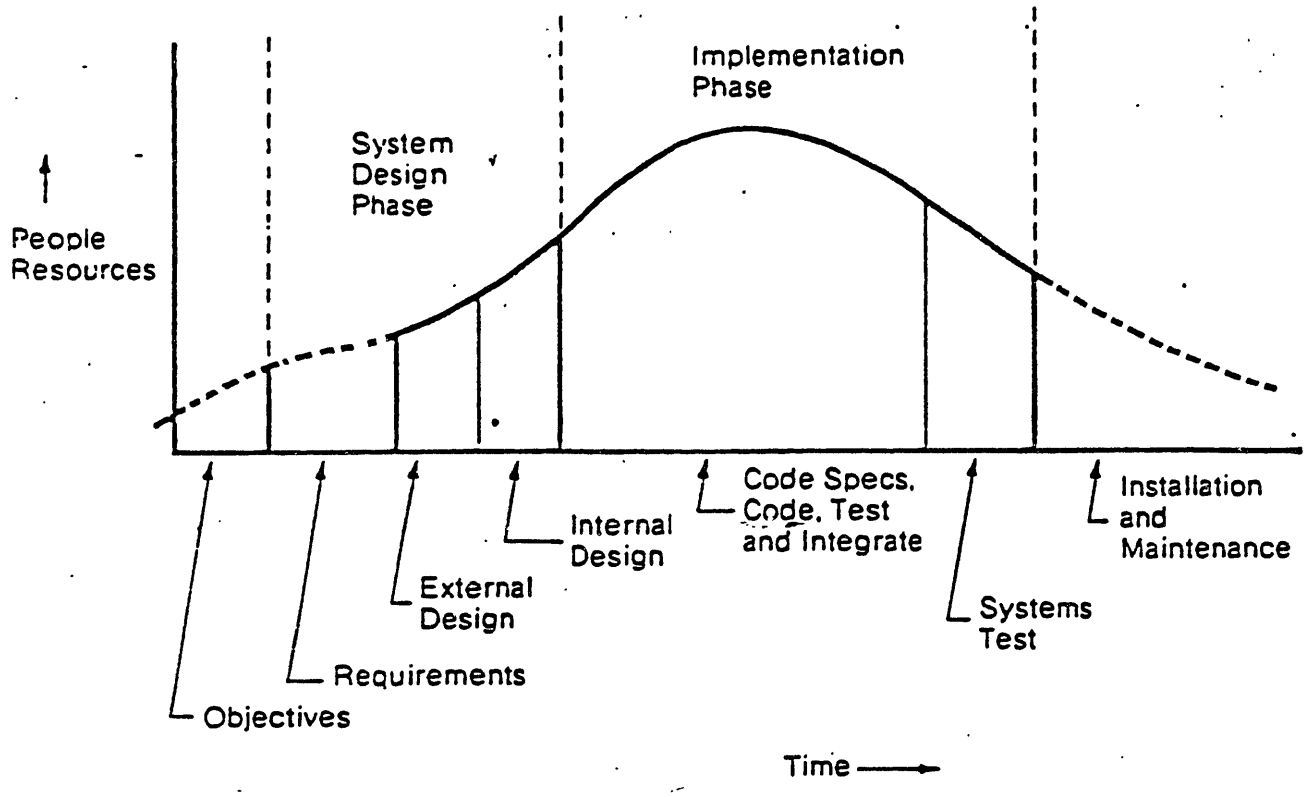

People
Resources
System
Design
Phase
Implementation
Phase
Objectives
Requirements
External
Design
Internal
Design
Code Specs.
Code, Test
and Integrate
Systems
Test
Installation
and
Maintenance
Time

Figure V.3

reproducing both types of distributions. As was explained in Chapter IV, the reason why the workforce level in the NASA project shoots upwards towards the end of the project has to do with NASA's tight scheduling constraints. Because software is embedded in a large and expensive space system, serious schedule slippages (e.g., [illegible] would jeopardize the launch date) can not be tolerated. Because of this, when software projects are planned, they are not only provided with a "Scheduled Completion Date," but, in addition, a "Maximum Tolerable Completion Date" is specified. As long as the "Scheduled Completion Date" is below the "Maximum Tolerable Completion Date" then decision to adjust the schedule, add more people, or do a combination of both will continue to be based on the balancing of scheduling and workforce stability considerations. However, as the "Scheduled Completion Date" starts approaching the "Maximum Tolerable Completion Date," as it does in the NASA project, pressures develop that override the workforce stability considerations. That is, project management becomes increasingly willing to "pay any price" necessary to avoid overshooting the "Maximum Tolerable Completion Date." And this translates, as the results indicate, into a management that is increasingly willing to hire more people.

Work Intensity:

In Chapter III it was explained that the "typical" 8-hr

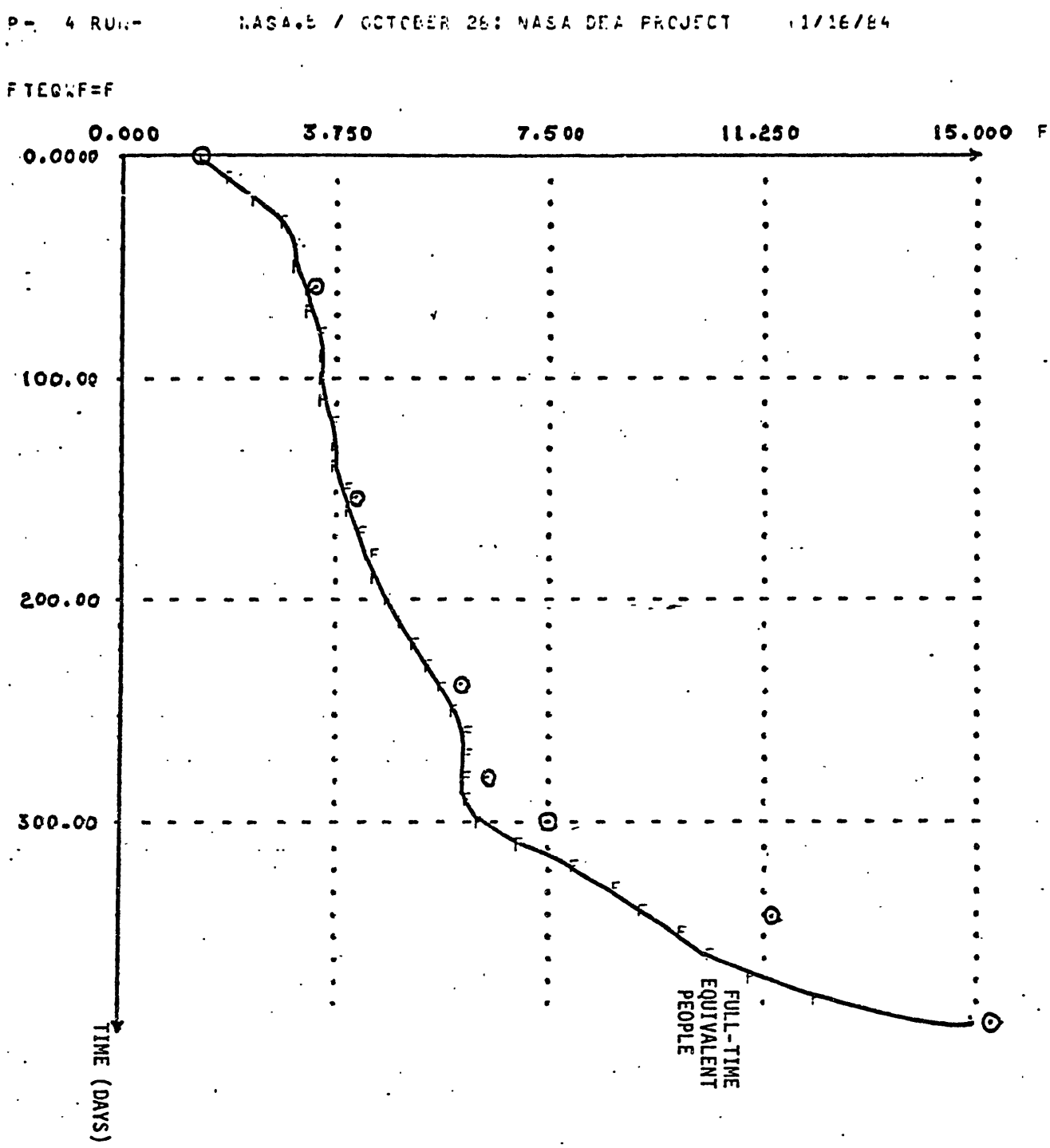

P= 4 RUN= NASA.5 / OCTOBER 26: NASA DEA PROJECT 11/16/84
FTEQ=F=F
0.000
3.750
7.500
11.250
15.000
F
0.0000
100.00
200.00
300.00
TIME (DAYS)
FULL-TIME EQUIVALENT PEOPLE

Figure V.4

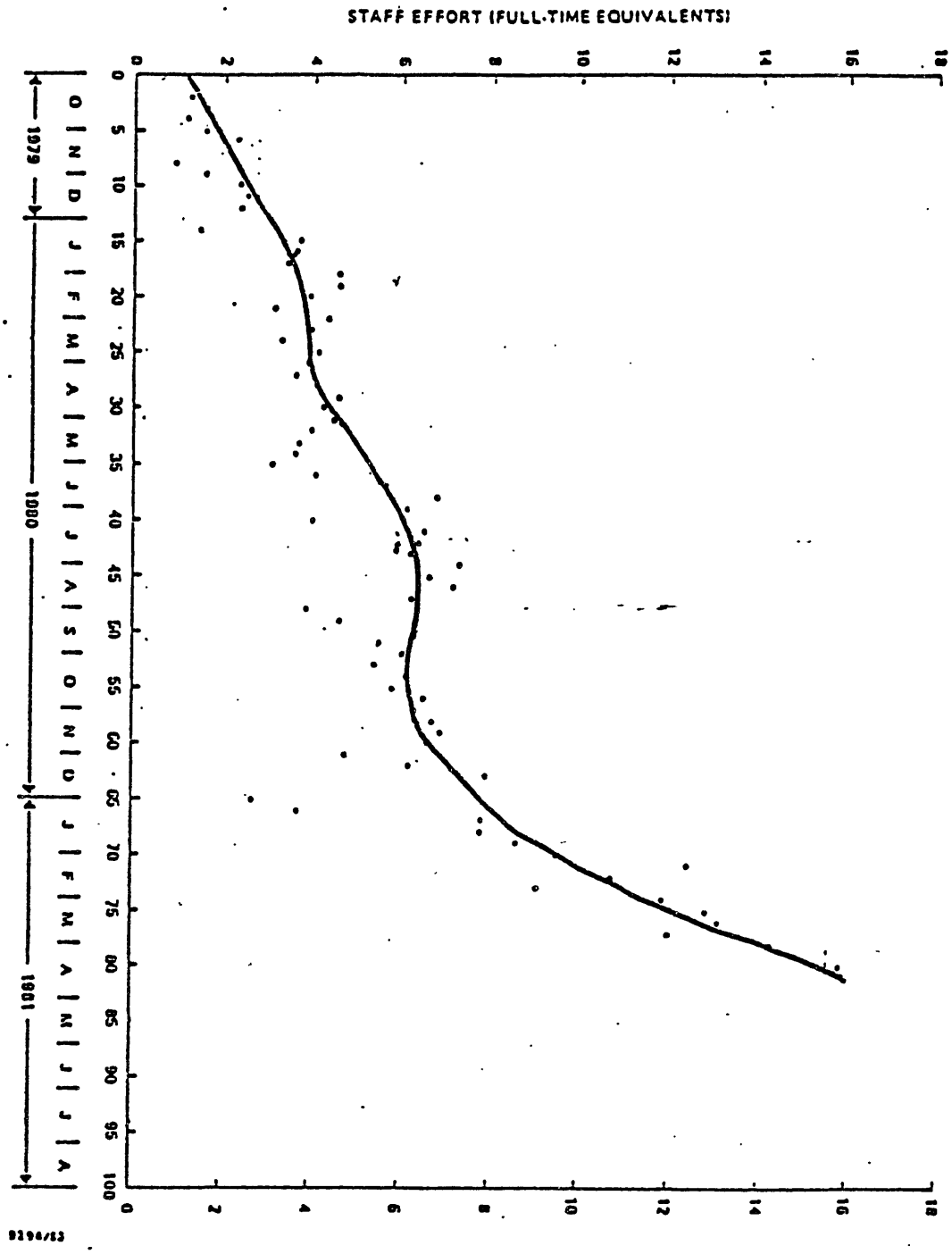

STAFF EFFORT (FULL-TIME EQUIVALENTS)
1979
1980
1981
O N D J F M A M J J A S O N D J F M A M J J A S O N D J F M A M J J A
0 5 10 15 20 25 30 35 40 45 50 55 60 65 70 75 80 85 90 95 100
0 2 4 6 8 10 12 14 16 18
0 2 4 6 8 10 12 14 16 18
8194/13

Figure V.5

day of a full-time staff member on a software project is not entirely devoted to productive project-related work. Time is often lost on personal matters, coffee-breaks, non-project communication, and other miscellaneous non-project related activities. These slack components comprise about 40% of the software person's time on the job.

The loss in productivity due to these slack components does not, of course, remain constant at the 40% level throughout the life of the project. The motivational effects of schedule pressures can push the "Actual Fraction of a Man-Day on Project" to both higher (under positive schedule pressure) as well as lower (under negative schedule pressure) values.

For example, positive schedule pressures arise whenever the project is perceived to be behind schedule. That is, whenever the total effort still needed to complete the project is perceived to be greater than the total effort actually remaining. Such a difference represents a perceived shortage in man-days on the project. When confronted with such a situation, software developers tend to work harder, i.e., allocate more man-hours to the project, in an attempt to compensate for the perceived shortage and bring the project back on schedule. This would be achieved by first compressing the slack time, and then, if needed, by working overtime. This then decreases the man-hour lost per-day,

while increasing the "Actual Fraction of a Man-Day on Project."

The dynamic behavior of the "Actual Fraction of a Man-Day on Project" for project EXAMPLE is depicted in Figure V.6. Notice that the two "spikes" in overwork occur, in both occasions, as an explicit project milestone is approached. The first spike occurs towards the end of the development phase (which includes both design and coding), and the second spike occurs towards the end of the only other explicit milestone in our model, the end of the system testing phase. This behavior pattern was observed by Boehm, and which he labelled as the "Deadline Effect" phenomenon. Figure V.7. shows his measured data on two projects with three major milestones: a plans and requirements review (PRR); a product design review (PDR), and an acceptance test (Boehm, 1981). It is clear that the Deadline Effect held strongly for both projects, generally producing a doubling of effort as each milestone is approached.

With a simulation model (such as ours), one need not guess at the cause of, say, a spike in a particular variable. Simulation experiments isolating and combining the effects of suspected factors can precisely pinpoint the mechanism(s) responsible. In the remaining part of this section, we will make use of this capability to trace out the set of actions and reactions that lead to the behavior pattern of the

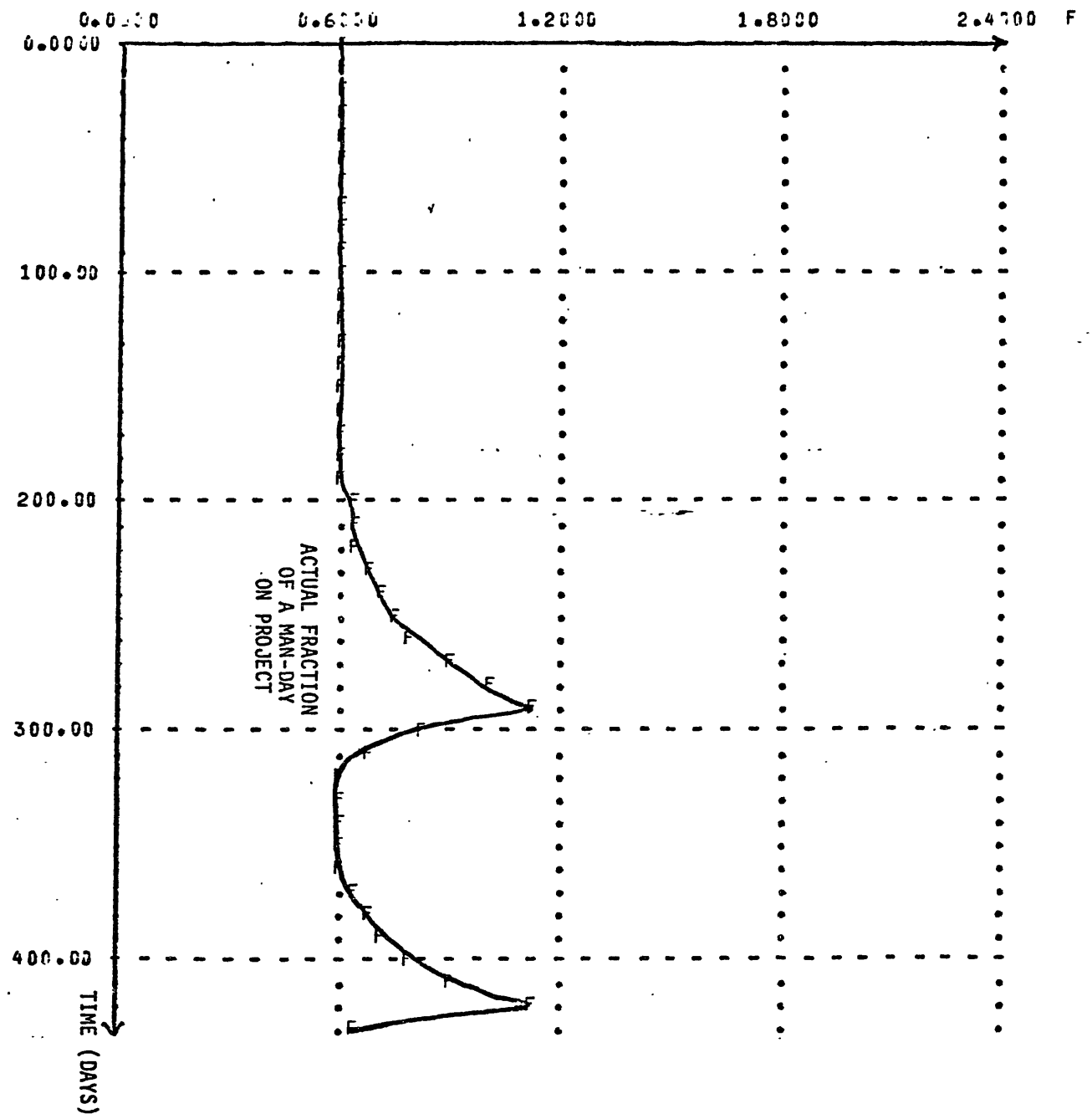

P- 9 RUN- BASE.5 / BASE MODEL: VERSION 5 01/04/84
AFMDPJ=F
0.0000
0.6000
1.2000
1.8000
2.4000 F
0.0000
100.00
200.00
300.00
400.00
ACTUAL FRACTION OF A MAN-DAY ON PROJECT
TIME (DAYS)

Figure V.6

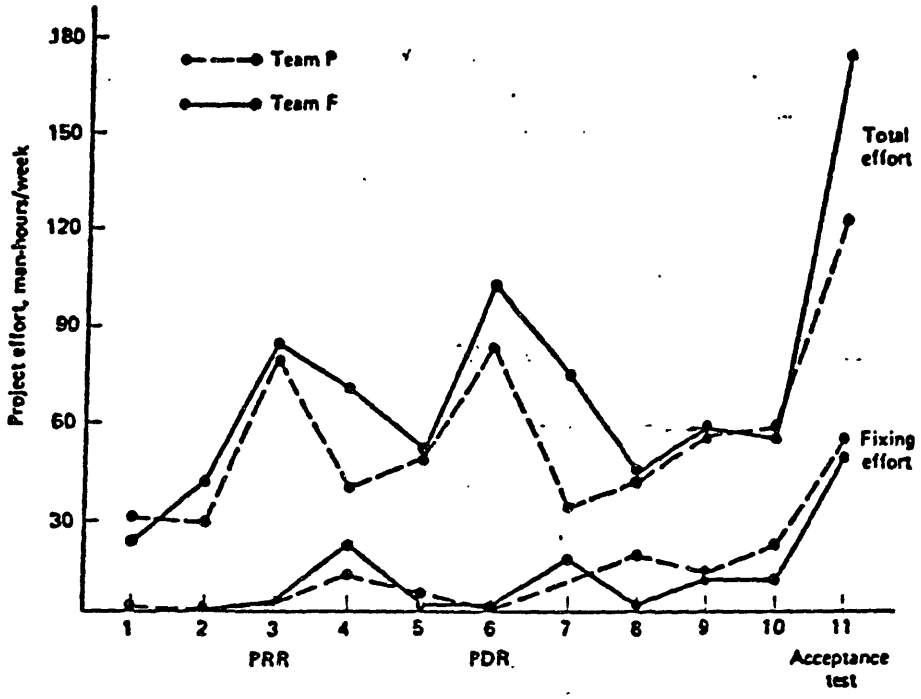

180
150
120
90
60
30
Project effort, man-hours/week
Team P
Team F
Total effort
Fixing effort
1
2
3
4
5
6
7
8
9
10
11
PRR
PDR
Acceptance test

Figure V.7

"Actual Fraction of a Man-Day on Project" shown in Figure V.6.

As was mentioned under "Project Progress," when project EXAMPLE is started, it is incorrectly perceived as being only 714.6 tasks in size, rather than 1,067 tasks ... its true size. As the project develops, and those "Undiscovered Job Tasks" are progressively discovered, adjustments are then made in the project's man-days budget to accomodate the additional work load. However, as has been explained in detail, these additional man-day allocations turn out to be less that what is actually required. This, therefore, creates a man-day shortage in the project. Unfortunately, though, such a man-day shortage is not immediately visible. In fact, it only becomes visible towards the end of the development phase, when the development work is almost finished and the allocated man-days budget is almost used up.

The "Perceived Shortage in Man-Days" is depicted in Figure V.8. The shortage in man-days is first perceived quite late in the development phase, at around day 180. As project members perceive the shortage, they react by working harder i.e., allocating more man-hours to the project, in an attempt to compensate for the perceived shortage and to bring the project back on schedule. Working harder translates in the model into the higher values of the "Actual Fraction of a Man-Day on Project" as shown in Figure V.8.

P- 11 RUN- BASE.5 / BASE MODEL: VERSION 5 01/04/84

AFMDPJ=F EXHLEV=X OVWDTH=V MDHDL=H PMDSHR=P SHRRPT=1 SCHPR=S
JBSZMD=D CUMMD=C SHRRPT=1

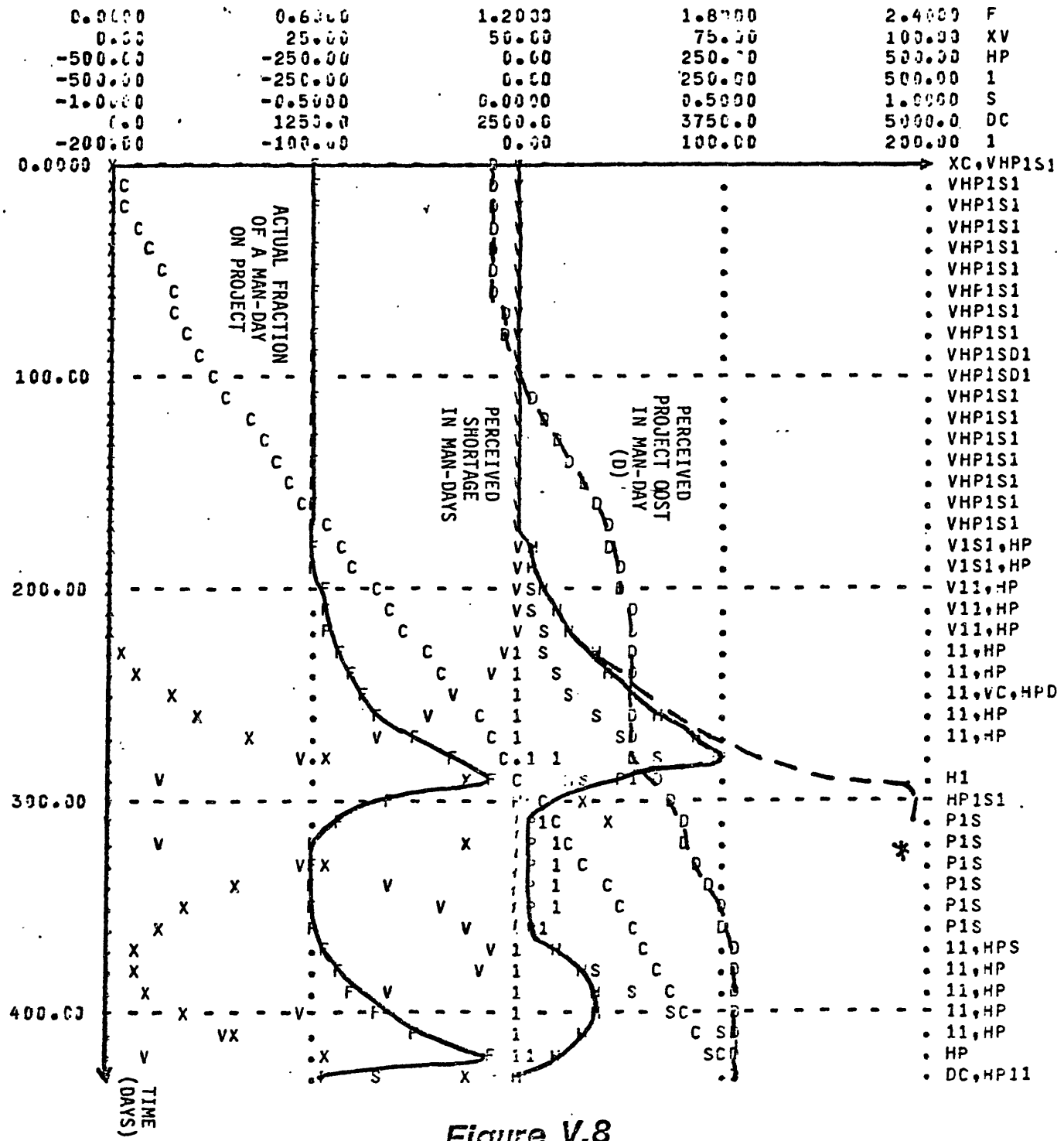


Figure V.8

Notice that even though the project members are working harder, the shortage in man-days keeps on rising. The reason this happens is that as the development phase continues to approach its final stages, the degree of visibility increases rapidly, exposing even larger man-day shortages. Thus, by working harder, project members are in effect only cutting into, not eliminating the man-day shortage, whose real magnitude is becoming progressively apparent. To appreciate the significance of the workforce's contributions, we also plotted curve (*) which depicts what the level of the "Perceived Shortage in Man-Days" would have been, had the project members maintained their normal (lower) work rate.

Project members would not, however, be willing to maintain an above-normal work rate indefinately. Once people start working harder, i.e., at a rate above their normal rate, their "Overwork Duration Threshold," which (as was explained in Chapter III) at any point in time would represent the maximum remaining duration for which they would be willing to continue working harder, would decrease. This happens because people enjoy their slack time (e.g., coffee breaks, social communications, personal business, ... etc.), and they would not tolerate prolonged deprivation of such "breathers." Thus a compressed slack-time exhausts them in the sense that it cuts into their tolerance level for continued hard work since that would mean a continued "deprivation" of their slack time. As the "Overwork Duration

Threshold" decreases, the maximum amount of man-days of backlogged work that the project members would be willing to handle (in addition to their planned work) also decreases. If this "Maximum Shortage in Man-Days to be Handled" happens to drop below the value of the "Perceived Shortage in Man-Days," only the maximum value would be handled through overwork, while arrangements with project management would be made to adjust the project's man-days budget so as to handle what exceeds the "Maximum Shortage in Man-Days to be Handled."

In project EXAMPLE, this is exactly what happens. That is, the persistence of the man-day shortage eventually overwhelms the workforce's intensified efforts, and around day 300 (i.e., at the end of the development phase) arrangements with project management are made to handle those remaining shortages through adjustments to both the project's man-days budget and its schedule.

The same sequence of events recur towards the end of the system testing phase. As testing progresses the system's error proneness becomes relatively more visible, and the project members become increasingly more able to perceive how productive (in testing) the workforce has actually been. As this happens, any shortages in man-days (for the testing phase) become more apparent. As Figure V.8. indicates, in project EXAMPLE such shortages are indeed perceived, and at

an accelerating rate, starting at day 370. It is interesting to note, though, that no such shortages were experienced in the NASA project, which, at first sight, seems counter-intuitive since in the NASA project only 15% of the project's man-days were allocated to the systems testing phase, whereas here, in project EXAMPLE, 20% were allocated. The answer lies in NASA's exceptionally high expenditures on Quality Assurance activities, which, as a fraction of the development effort, is almost double that of EXAMPLE. As a result, in the NASA project a larger fraction of the errors is detected early on during the development phase (when errors are relatively less costly to detect and correct), which of course dramatically reduces the workload of the system testing activity.

Returning to project EXAMPLE and Figure V.8., as the man-day shortage is detected, the workforce reacts again by working harder i.e., compressing their slack time and increasing the "Actual Fraction of a Man-Day on Project." At this stage, though, the magnitude of the shortages is not as high as those experienced towards the end of the development phase. Recall that at the end of development, the man-day shortage had to be handled, not only through a surge in productivity, but also through additions to the project's man-days budget. Here, however, the shortage in man-days is sufficiently low to be handled solely through this final surge in productivity.

Conclusion:

The objective of this section was to define the experimental setting within which to conduct our experimentation and analysis of the dynamics of software development. We first characterized the "EXAMPLE" software project, which will serve as the prototype project for the experiments. We then ran the model to simulate EXAMPLE and observed its behavior. The behavior of a number of project variables were presented and explained. And we also demonstrated that the model's behavior pattern does replicate those reported in the literature. With this done, we are now ready to move on to the next three sections, where we use the model as a laboratory tool to study the dynamic implications of an array of managerial actions, policies, and procedures in the four areas of (1) scheduling, (2) controlling, (3) quality assurance, and (4) staffing.

V.3. Software Cost and Schedule Estimation:

Over the years, estimation of software project development time and cost has been an intuitive process. Experience and analogy have been used as a basis to develop estimates for any given project (Oliver, 1982), (McKeen, 1981), (Gehring, 1976). More recently, a number of quantitative software estimation models have been developed.

They range from highly theoretical ones, such as Putman's model (1978), to empirical ones, such as the Walston and Felix model (1977), and Boehm's COCOMO model (Boehm, 1981). An empirical model uses data from previous projects to evaluate the current project and derives the basic formulae from analysis of the particular data base available. A theoretical model, on the other hand, uses formulae based upon global assumptions, such as the rate at which people solve problems, the number of problems available for solutions at a given point in time, ... etc.

Still, software cost schedule estimation continues to be a major difficulty associated with the management of software development (Devenny, 1976), (Distaso, 1980), (Mills, 1976), (Pooch and Gehring, 1980), (Yourdon, 1982), (Zelkowitz et al, 1979), (Zmud, 1980). "Even today, almost no model can estimate the true cost of software with any degree of accuracy" (Mohanty, 1981). Farquhar (1970), articulates the significance of the issue:

> Unable to estimate accurately, the manager can know with certainty neither what resources to commit to an effort nor, in retrospect, how well these resources were used. The lack of a firm foundation for these two judgements can reduce programming management to a random process in that positive control is next to impossible. This situation often results in the budget overruns and schedule slippages that are all too common today.

A number of reasons for the difficulty have been suggested in the literature, including:

1. Software development is a process, that is not yet fully understood by "estimators" (Myers, 1972), (Oliver, 1982), (Gehring and Pooch, 1980), (Synnott and Gruber, 1981), (Pietrasanta, 1968).

2. The phases and functions which comprise the software development process are influenced by a large number of ill defined variables (Gehring and Pooch, 1980), (Devenny, 1976), (Aron, 1976), (Distaso, 1980), (Pressman, 1982), (Oliver, 1982).

3. Most of the activities within the process are still primarily human rather than mechanical, and therefore prone to all the subjective factors which affect human performance (Gehring and Pooch, 1980), (Pressman, 1982), (Oliver, 1982).

Identifying the causes of a difficulty or a problem is an important first step towards resolving the difficulty or problem. The next step is to then identify a strategy for handling those identified hurdles. For the software estimation problem, a strategy that has been frequently quoted in the literature was articulated by Pietrasanta more than a decade ago:

> The serious student of estimating must first be willing to probe deeply into the fascinating and complex system

> development process, to uncover the phases and functions of the process, to highlight the subtle interrelationships of the program system being developed and the project organization doing the developing ... relationships is precisely what is required if estimates are ever to be improved. Only then can we do meaningful quantitative research and scientific analysis of resource requirements (Pietrasanta, 1968).

Having "probed deeply into the fascinating and complex system development process," and captured within our integrative system dynamics model (we hope) those "influence variables of software development and their causal relationships," we will embark, in this section, on a quantitative analysis of software cost and schedule estimation. We will conduct three separate experiments. In one, we will focus on the most widely used estimation technique, namely, estimation by analogy. We will examine the long-term implications of using such a method. And we will demonstrate how the feedback concept is a useful tool to study those long-term dynamic issues. The second feature of our modeling approach, namely, its integrative perspective, will prove useful in a second experiment, in which we focus on the much heralded quantitative estimation tools. We will identify a number of managerial and organizational variables that the current models fail to "acknowledge," but which significantly influence the cost of software development. Finally, in the third experiment, we turn our attention from the techniques of software estimation, to address a more basic issue. It is the issue of estimation accuracy.

The above three experiments are discussed next, in reverse order.

V.3.1. On the Accuracy of Software Estimation:

In this section we will show firstly, why software cost estimators should reject the notion that a (new) software estimation tool can be adequately judged on the basis of how accurately it matches historical project results; and secondly, why a more accurate estimate is not necessarily a "better" estimate.

Consider the following situation: A 64 KDSI software project which has been estimated at its initiation, using an estimation method "A," to be 2,359 man-days, ends up actually consuming, at its completion, 3,795 man-days. The project's characteristics (e.g., its size, complexity, ... etc.) are then fed into another estimation method "B" (e.g., that is being considered by management for adoption) and its results compared to the project's actual performance. And let us assume that method "B" produces a 5,900 man-day estimate. If we define "% of relative absolute error" in estimating man-days (MD) as,

$$\% \text{ Error} = 100 * ABS[MD_{ACT} - MD_{EST}] / MD_{ACT}$$

Then, for estimation method "A,"

$$\%\ Error_A = 100 * ABS[3,795-2,359] / 3,795$$
$$= 38\%$$

And for method "B,"

$$\%\ Error_B = 100 * ABS[3,795-5,900] / 3,795$$
$$= 55\%$$

Question: Can one conclude from this that estimation method "B" would have provided a less accurate estimate of the project's man-days, had it been used instead of method "A"?

The answer is NO. And the reason why we cannot make such a conclusion is that we cannot, and should not, assume, that had the project been initiated with B's 5,900 man-day estimate, instead of A's 2,359 man-day estimate, that it would have still ended up actually consuming exactly 3,795 man-days. In fact the project could end-up consuming much more or much less than 3,795 man-days. And before such a determination can be made, no "accurate" assessment of the relative accuracy of the two methods can be made.

The point we are trying to make is this: a different estimate creates a different project.

This phenomenon is somewhat analogous to the "General Heisenberg" principle in experimentation. The principle is stated as follows: "When experimenting with the system about which we are trying to obtain knowledge, we create a new

system" (Koolhass, 1982). Koolhas gives a fine example of this: "A man who inquires through the door of the bedroom where his friend is sick, How are you? whereupon his friend replies Fine, and the effort kills him."

In an analogous manner, by imposing different estimates on a software project we would, in a real sense, be creating different projects. In the remainder of this discussion we will explain how.

Research findings clearly indicate that the decisions that people make in project situations, and the actions they choose to take are significantly influenced by the pressures and perceptions produced by the project's schedule (Roberts, 1981b), (Hart, 1982), (Shooman, 1983), (Gagliardi, 1981), and (Brooks, 1978). In our model, we capture such schedule influences. The most significant of which are depicted in the causal loop diagram of Figure V.9.

Schedules have a direct influence on the hiring and firing decisions throughout the life of a software project. As was shown earlier in this chapter, in TRW's COCOMO model, the project's staff size is simply determined by dividing the man-days estimate (MD) by the development time estimate (TDEV). Thus, for example, a tight time schedule (i.e., a low TDEV value) means a larger workforce. We also saw how scheduling can dramatically change the manpower loading

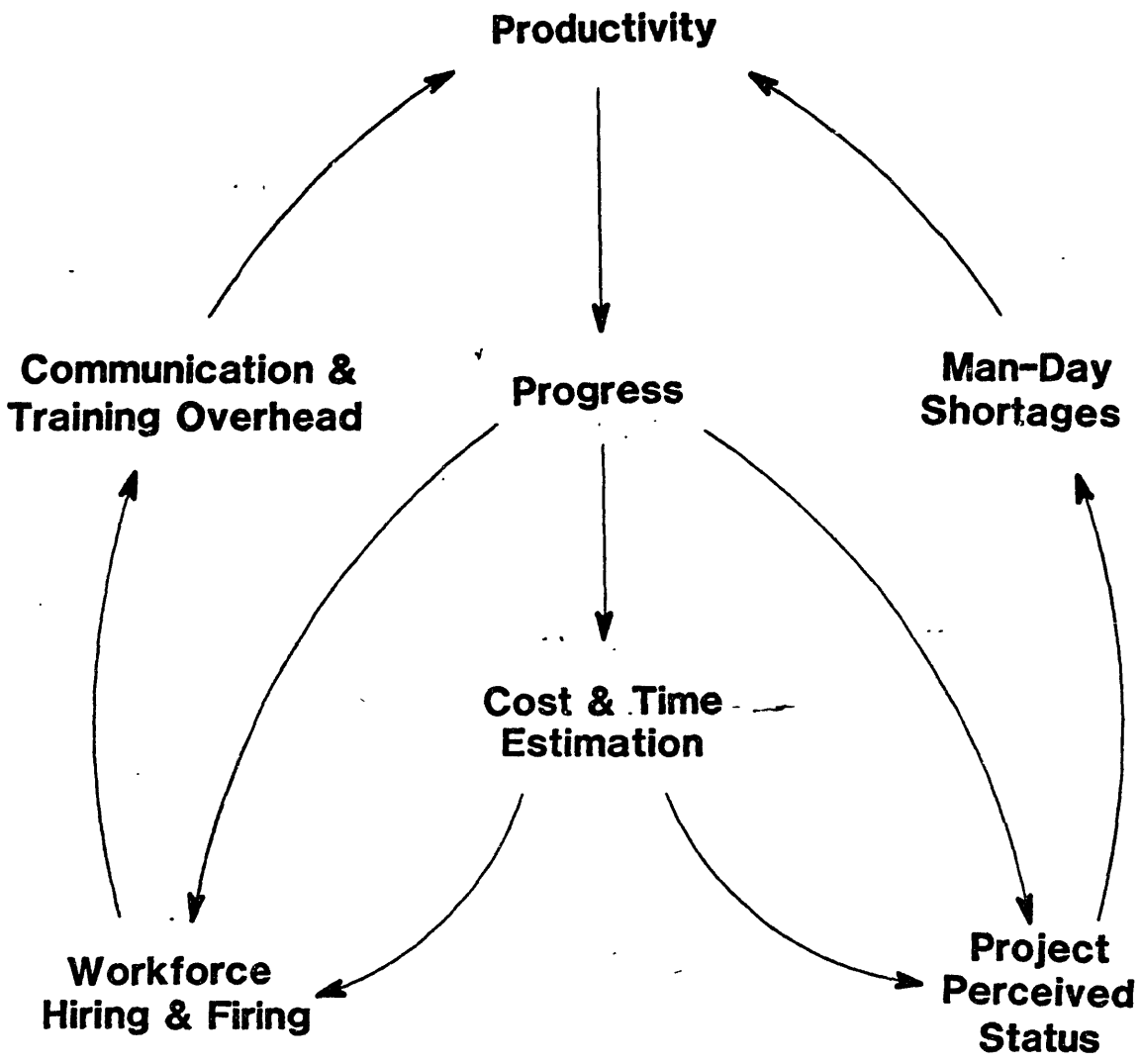

Productivity
Communication &
Training Overhead
Progress
Man-Day
Shortages
Cost & Time
Estimation
Workforce
Hiring & Firing
Project
Perceived
Status

Figure V.9

throughout the life of a project. For example, we saw how the workforce level shoots upwards towards the end of the NASA project (but not in project EXAMPLE), because of NASA's strict constraints on the extent to which the project's schedule is allowed to slip.

Through its effects on the workforce level, a project's schedule also affects productivity. This happens because a higher workforce level, for example, means more communication and training overhead, which in turn affects productivity negatively.

As shown in Figure V.9. (and as we explained in detail in Chapter III), productivity is also influenced by the presence of any man-day shortages. For example, if the project is perceived to be behind schedule i.e., when the total effort still needed to complete the project is perceived to be greater than the total effort actually remaining in the project's plan, software developers tend to work harder i.e., allocate more man-hours to the project, in an attempt to compensate for the perceived shortage and to bring the project back on schedule. Such man-day shortages are, obviously, more prone to occur when the project is initially underestimated. Conversely, if project management initially over-estimates the project, man-day "excesses" could arise. And when the project is perceived to be ahead of schedule i.e., when the total man-days remaining in the

project's plan exceed what the project members perceive is needed to complete the project, "Parkinson's Law indicates that people will use the extra time for ... personal activities, catching up on the mail, etc." (Boehm, 1981). Which, of course, means that they become less productive.

Having identified how software project estimation can influence project behavior, are we now in a position to return back to the example we posited at the beginning of this section, and answer the still unanswered question, namely, whether estimation method "A" is truely more accurate than method "B"?

Identifying the feedback relationships through which software estimation influences project behavior is one thing, and discerning the dynamic implications of such interactions on the total system is another. Paraphrasing Richardson and Pugh (1981),

> The behavior of systems of interconnected feedback loops often confounds intuition and analysis, even though the dynamic implications of isolated loops may be reasonably obvious.

One option that might be suggested, is to conduct a controlled experiment, whereby the 64 KDSI software project is conducted twice under exactly the same conditions, except that in one case it would be initiated with a 2,359 man-day estimate (i.e., on the basis of method "A"), and in the

second case with a 5,900 man-day estimate (i.e., on the basis of method "B"). While theoretically possible, such an option is almost infeasible from a practical point of view because of its high cost, both in terms of money and time.

Simulation experimentation provides an, obviously, more attractive alternative. In addition to permitting less-costly and less-time-consuming experimentation, simulation experimentaion makes "perfectly" controlled experiments possible (Forrester, 1961).

However, rather than conduct a limited experiment simply to investigate methods "A" and "B," above, we will instead conduct a broader experiment that answers a broader set of issues that were raised in one of the organizations we interviewed in.

In the particular organization, project managers were rewarded on how close their projects met their initially estimated man-days budget. The estimation procedure that they informally used was as follows:

1. Use Basic COCOMO to estimate the number of man-days (MD). That is, use

   $MD = 2.4 * 19 * (KDSI)^{1.05}$ man-days

2. Multiply this estimate by a Safety Factor. The safety factor ranged from 25% to 50%.

3. Use the new value of man-days (MD') to calculate the development time (TDEV) using COCOMO. That is, use

$$TDEV = 47.5 * (MD'/19)^{0.38} \text{ days}$$

It is important to note, before we proceed with our experiment, that this "Safety Factor Philosophy" is not, in any way, unique to this one organization. For example, in a study of the software cost estimation process at the Electronic System Division of the Air Force Systems Command, Devenny (1976) found that most program managers budget additional funds for software as a "management reserve." He also found that these management reserves ranged in size (as a percentage of the estimated software cost) from 5% to 50% with a mean of 18%. And as was the case in the organization we interviewed in, the policy was an informal one: " ... frequently the reserve was created by the program office with funds not placed on any particular contract. Most of the respondents indicated that the reserve was not identified as such to prevent its loss during a budget cut" (Devenny, 1976).

To test the efficacy of such an informal policy we will run a number of simulations of our prototype project, namely, project EXAMPLE, with different values for the Safety Factor. We will experiment with values ranging from 0 (i.e., the base run) to 100%. For example, for a Safety Factor of 50%, the project would be initialized with the following estimates:

1. First, calculate MD,

   $MD = 2.4 * 19 * (42.88)^{1.05} = 2,359$ man-days

2. Second, calculate MD'

   MD' =MD * (1+Safety-Factor/100)
   =MD * 1.5 = 3,538.5 man-days

3. Finally, calculate TDEV

   $TDEV = 47.5 * (MD'/19)^{0.38} = 346$ days

The results of the experiment are exhibited in Figures V.10. through V.13.

In Figure V.10., the % of the relative error in estimating man-days 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. In particular, in the 25-50% range (which is what was used in the organization) the estimation error drops from being approximately 40% in the base run, to values in the upper twenties. In fact, Figure V.10. suggests that by using a Safety Factor in the 25-50% range, the project manager might not be going far enough, since a 100% Safety Factor, for example, would drop the estimation error down to a "more rewarding" 12%.

The rational, or the justification, for using a Safety Factor (as provided by our interviewees) is based on the following set of assumptions:

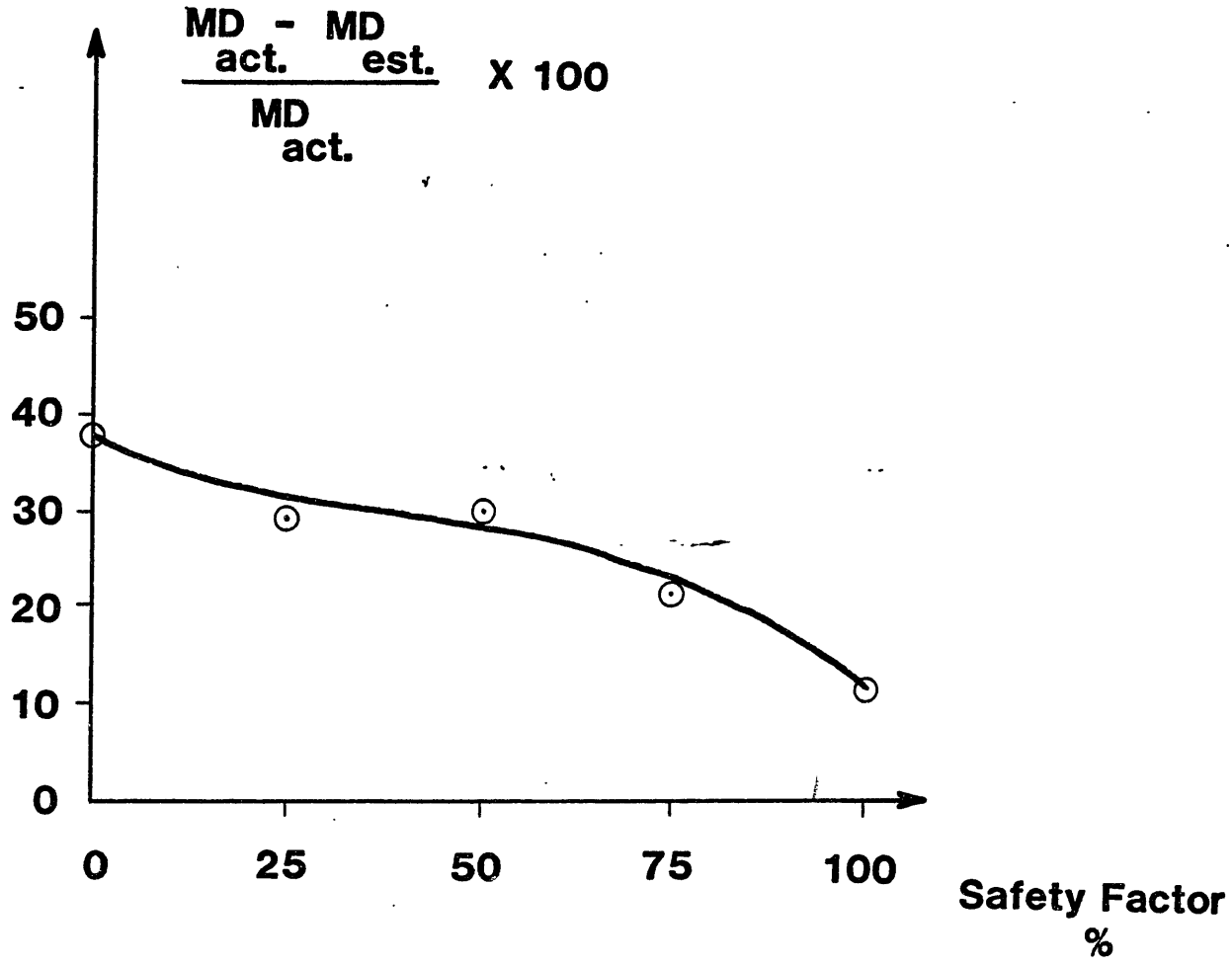

MD - MD
act. est.
MD
act.
X 100
50
40
30
20
10
0
0
25
50
75
100
Safety Factor
%

Figure V.10

1. Past experiences indicate a strong bias on the part of software developers to underestimate the scope of a software project.
2. "(One) might think that a bias would be the easiest kind of estimating problem to rectify, since it involves an error that is always in the same direction ... (But biases) are, almost by definition, invisible ... the same psychological mechanism (e.g., optimism of software developers), that creates the bias works to conceal it" (DeMarco, 1982).
3. To rectify this bias on the part of software developers (e.g., systems analysts and designers), project management must use a Safety Factor. When the project manager " ... adds a contingency factor (25%? 50? 100?) he is, in effect, saying that: 'much more is going to happen that I don't know about, so I'll estimate the rest as a percentage of that which I do know something about'" (Pietrasanta, 1968).

In other words, the assumption is that the Safety Factor is simply a mechanism to bring the initial man-days estimate closer to the project's true size in man-days ... as shown in Figure V.11.

Notice that such an assumption cannot be contested solely on the basis of Figure V.10. which provides only part

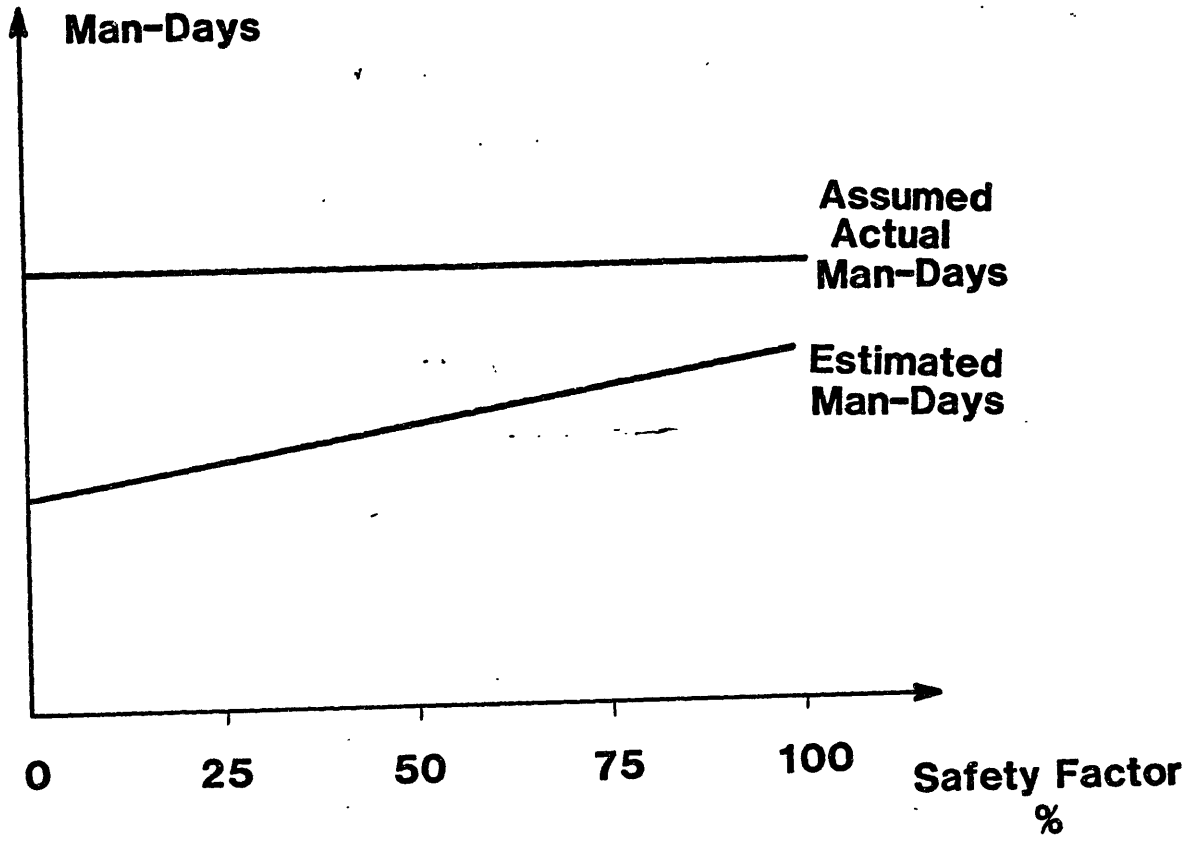

Man-Days
Assumed
Actual
Man-Days
Estimated
Man-Days
0
25
50
75
100
Safety Factor
%

Figure V.11

of the story. A more complete picture is provided by Figure V.12. In the figure we plot the actual man-days that were consumed by the project EXAMPLE, when different Safety Factors are applied to its initial estimate. The assumption of Figure V.11 is obviously invalidated. As higher Safety Factors are used, leading to more and more generous initial man-day allocations, the actual amount of man-days consumed, does not remain at some inherently-defined value. For example, in the base run, project EXAMPLE would be initiated with a man-day estimate of 2,359 man-days and would end up consuming 3,795 man-days. When a Safety Factor of 50% is used, i.e., leading to a 3,538 man-day initial estimate, EXAMPLE ends up consuming, not 3,795 man-days, but 5,080 man-days. To reiterate a point made earlier:

A different estimate creates a different project.

The reason why this happens (as was explained earlier) is that the project's initial estimates create pressures and perceptions that affect how people behave on the project. In particular, an overestimate of the project's man-days can lead to a larger buildup of the project's workforce, leading to higher communication and training overheads, which in turn affect productivity negatively. In addition, when a project is overestimated, it often leads to an expansion of the project members' slack time activities (e.g., non-project communication, personal activities, ... etc.), leading to

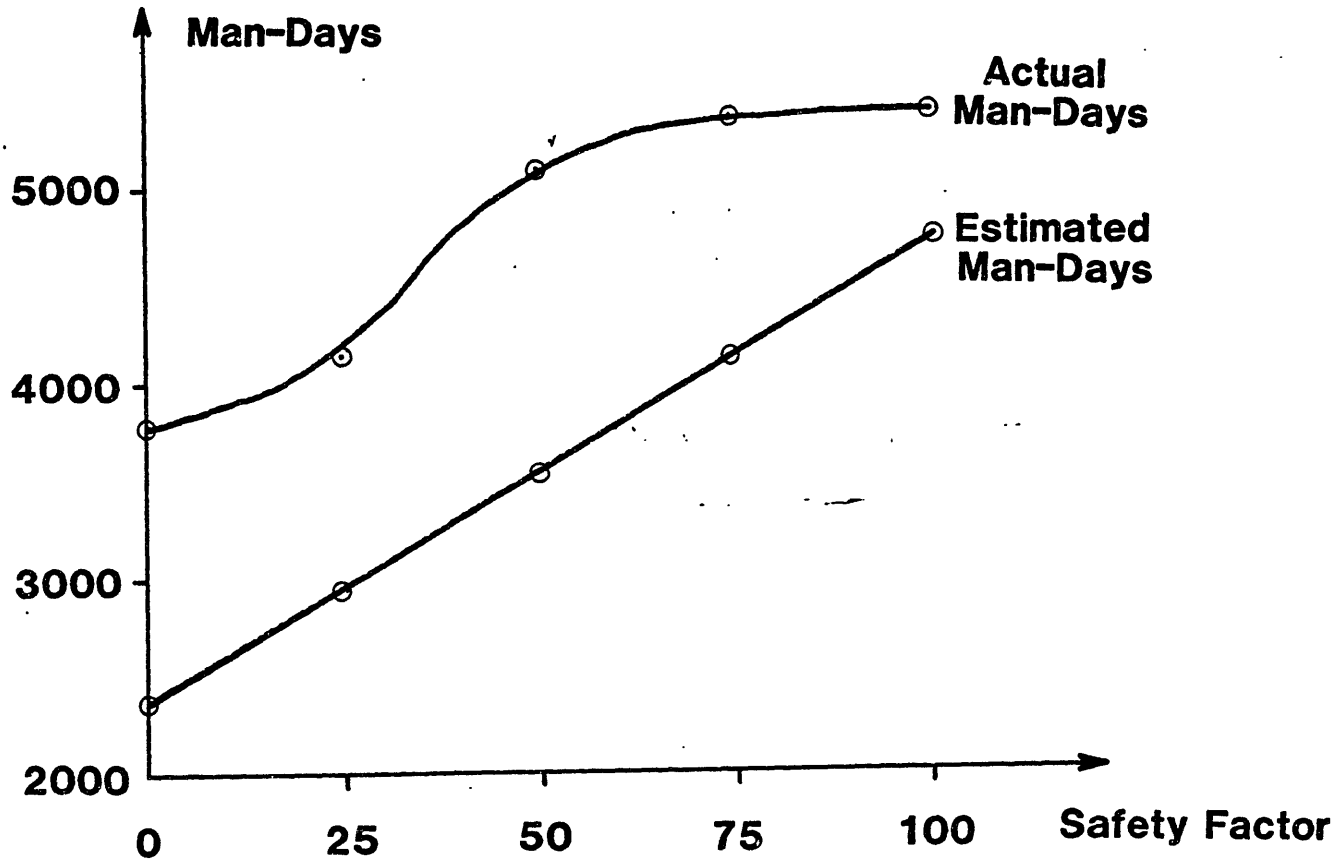

Man-Days
Actual
Man-Days
Estimated
Man-Days
5000
4000
3000
2000
0
25
50
75
100
Safety Factor

Figure V.12

further reductions in productivity.

Figure V.13. is a plot of "Gross Productivity," which is defined as the project size in DSI (i.e., 64,000 DSI) divided by the actual number of man-days expended, for the different Safety Factor situations. Gross Productivity drops from a value of 16.8 DSI/Man-Day in the base run, to as low as 12 when a 100% Safety Factor is used. Notice that the drop in productivity is initially significant, and then levels off for higher Safety Factors. The reason for this is that when the Safety Factor increases from 0 (i.e., in the base run) to say a relatively small value (e.g., 25%) most of the man-day excesses that result are absorbed by the employees. This happens in two ways, less overworking and more slack time. Recall that in project EXAMPLE's base run, man-day backlogs occurred towards the end of both the development phase and the system testing phase leading to periods of overwork. When a small Safety Factor is used, however, such backlogs will decrease, leading to less overwork durations. As the Safety Factor is increased further, man-day excesses, rather than backlogs will result. When these excesses are "reasonable" they tend to be largely absorbed in the form of reasonably expanded slack activities. However, as was explained in detail in Chapter III, there is a limit on how much "fat" employees would be willing, or allowed, to absorb. Beyond these limits, man-day excesses would be translated into cuts in the project's workforce,

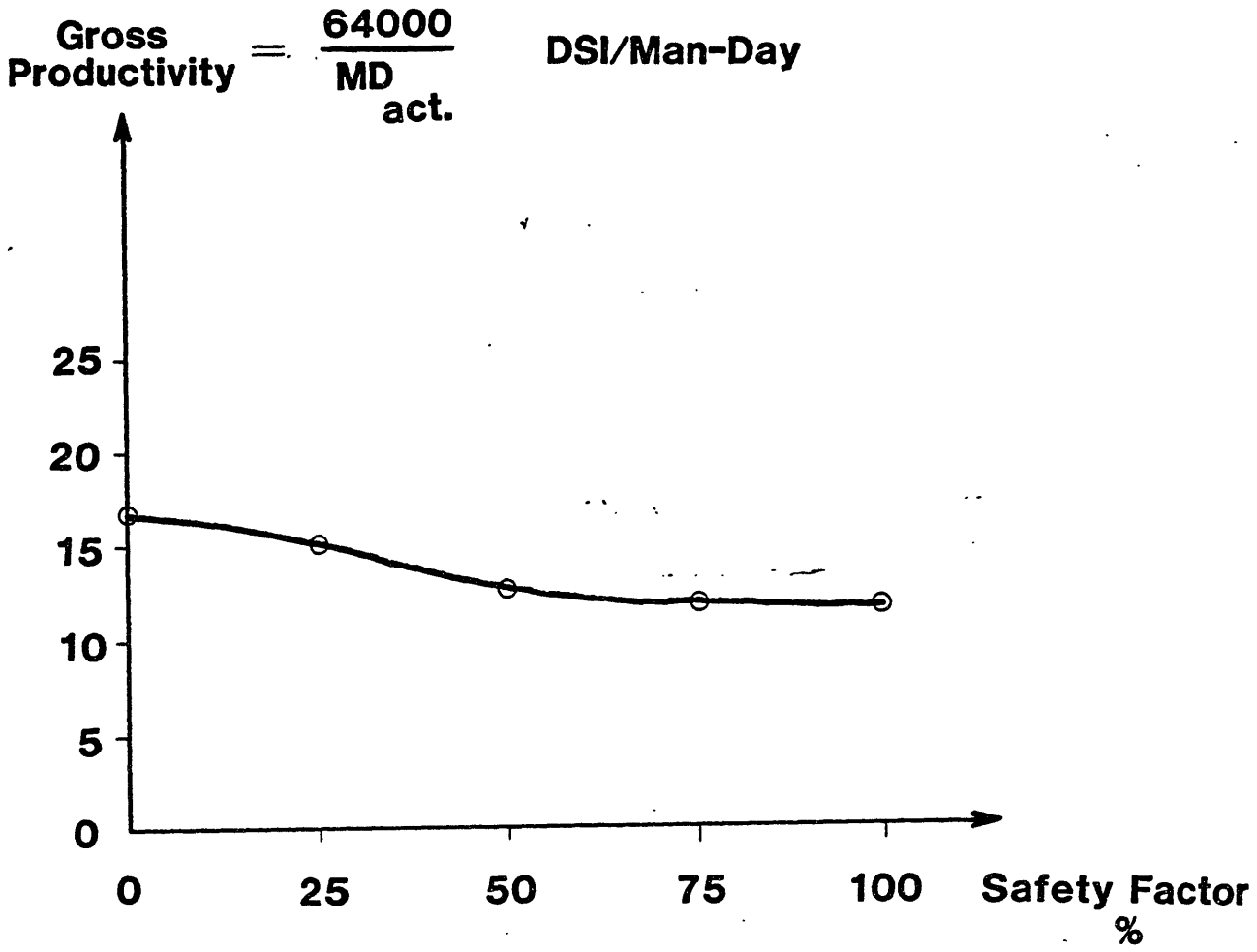

Gross Productivity = 64000 / MD act. DSI/Man-Day
25
20
15
10
5
0
0
25
50
75
100
Safety Factor %

Figure V.13

schedule, or both. Thus, as the Safety Factor increases to larger and larger values, losses in productivity due to the expansion of the slack time activities decreases, leading to lower and lower drops in Gross Productivity.

We are now in a position to answer the question posited at the beginning of this discussion. The situation concerned a 64 KDSI project which is in fact our own project EXAMPLE, and a comparison of two estimation methods. Method "A" produces a 2,359 man-day estimate. It is, in other words, the estimate used in the base run. Since, project EXAMPLE ended up actually consuming 3,795 man-days, the % of relative absolute error in estimating man-days is 38%. We then questioned whether a new estimation methods "B," which produces a 5,900 man-day estimate for project example (i.e., an estimate that is 55% higher than EXAMPLE's actual man-day expenditures of 3,795), would have provided a less accurate estimate of the project's man-days, had it been used instead of method A.

Notice that method "B's" estimate of 5,900 man-days is 150% higher than "A's" 2,359 estimate i.e., method "B" is equivalent to a "Safety Factor Policy" in which the Safety Factor is set to 150%. To check the behavior of project EXAMPLE had it been estimated using Method "B," we re-ran the model with an initialized value of the man-days estimate (MD) equal to 5,900. The results of the run, together with those

of the base case are tabulated below:

	Method "B"	Method "A" (Base Run)
MD_{EST}	5,900	2,359
MD_{ACT}	5,412	3,795
% Error	9 %	38 %

The results are quite interesting. Method "B" turns out to be, in fact, a more accurate estimator. However, the improved accuracy is attained at a high cost. The project turns out consuming 43% more man-days!

In terms of the real life organization we interviewed in, the message is the same. The "Safety Factor Policy" does achieve its intended objective, namely, produces relatively more accurate estimates. However, the organization is paying dearly for this. As Figure V.12. indicates, a Safety Factor in the 25-50 % range results in a 15-35% increase in the project's cost in terms of man-days.

To conclude this section, we restate the two basic insights we gained:

1. A different estimate creates a different project. The important implication that follows from this is that both the project manager as well as the student of

software estimation should reject the notion that a new software estimation model can be adequately judged on the basis of how accurately it can estimate historical projects. Because of the significant influence that a schedule has on the behavior of a software project, the only real test of an estimation method is to try it.

2. A more accurate estimate is not necessarily a better estimate. An estimation method should not be judged only on how accurate it is, but in addition it should be judged on how costly the projects it "creates" are.

V.3.2. On the Portability of the Quantitative Software Estimation Models:

There has been a fair amount of work towards developing different kinds of quantitative software estimation models. These models vary in what they provide (e.g., total cost, manning schedule) and what factors they use to calculate their estimates. They also vary with regard to the type of formula and parameters they incorporate. In almost all cases, the model is based either directly or indirectly on past historical data (Shooman, 1983). Sometimes the collected data are translated into tables or graphs indicating the productivity (instructions per man-day, man-month, or man-year). Another approach is to formulate a parametric model, a mathematical function of several

variables, suggested by previous experimentation and engineering judgement. Statistical techniques are then applied to the data in order to reduce the number of model variables (analysis of variance and correlation) and to compute the constants in the equation (parameter estimation).

However, "Even today, almost no model can estimate the true cost of software with any degree of accuracy" (Mohanty, 1981). For example, the "Basic COCOMO" estimates come within a factor of 1.3 of the actual development figures for the projects in the COCOMO data base only 28% of the time, and with a factor of 2 only 60% of the time" (Boehm, 1981). The 1965 SDC model had a standard deviation which was larger than the mean estimate (Nelson, 1966). The analysis of the IBM-FSD model in (Walston and Felix, 1977) reported a standard deviation of a factor of 1.71 (mean of 274 instructions/man-month; range about the mean of 160-470 instructions/man-month).

Furthermore, the portability of such models from the companies in which they were developed to another, has proven to be poor (Benbasat and Vessey, 1980), (Boehm, 1981), (Mohanty, 1981).

The thesis of this section is that both the accuracy as well as the portability of software estimation models can be significantly improved by taking into consideration not only

the technical aspects of the software development environment, as is the case with the current models, but, in addition, by accounting for the managerial and organizational characteristics of the environment. Specifically, we will identify a number of managerial and organizational variables that the current models fail to "acknowledge," but which significantly influence the cost of software development.

To set the stage for our analysis, we will first report on an interesting experiment by Mohanty (1981), which cleverly demonstrates the above two weaknesses in the current models.

Mohanty's objective was to examine the extent to which the available quantitative software estimation models produce the same cost estimate for a given project. The following models were included in the exercise: (1) The Farr and Zagorski Model; (2) The Kustanowitz Model; (3) The Wolverton Model; (4) The Walston-Felix Model; (5) The Aerospace Model; (6) The Aaron Model; (7) The GRC Model; (8) The Naval Air Development Center Model; (9) The Doty Model; (10) The SDC Model; (11) The Schneider Model; and (12) The Price-S Model.

In order to fully specify his hypothetical software project for the experiment, it was necessary to first identify the full set of factors that are collectively

incorporated in the 12 models. Once this set is identified, the software project could then be defined in terms of this set of parameters. Forty-nine factors were identified. They involved system size, data base, system complexity, type of program, documentation, environment (e.g., requirements definition, security, and computer access), and an "other" category that includes such items as miles traveled, reliability, and growth requirements. However, none of the cost models described uses all the factors. Cost models developed before 1974, for example, emphasized productivity without considering the quality of the finished product. Newer cost models do consider quality; however, they do not include it explicitly.

As we said, a hypothetical software project was then defined in terms of the identified set of parameters. The size of the project was chosen to be 36,000 machine-language executable instructions. The resulting 12 cost estimates for the project are exhibited in Figure V.14. (Note: the estimates cover the design, coding, and testing phases only ... as does our model.) As the figure indicates, the estimated cost varies from a low of $362,500 (the Farr and Zagorski Model) to a high of 2,766,667 (the Kustanowitz Model) for the same software project.

Since the size of the project and cost per instruction were the same for the different models, the variations in

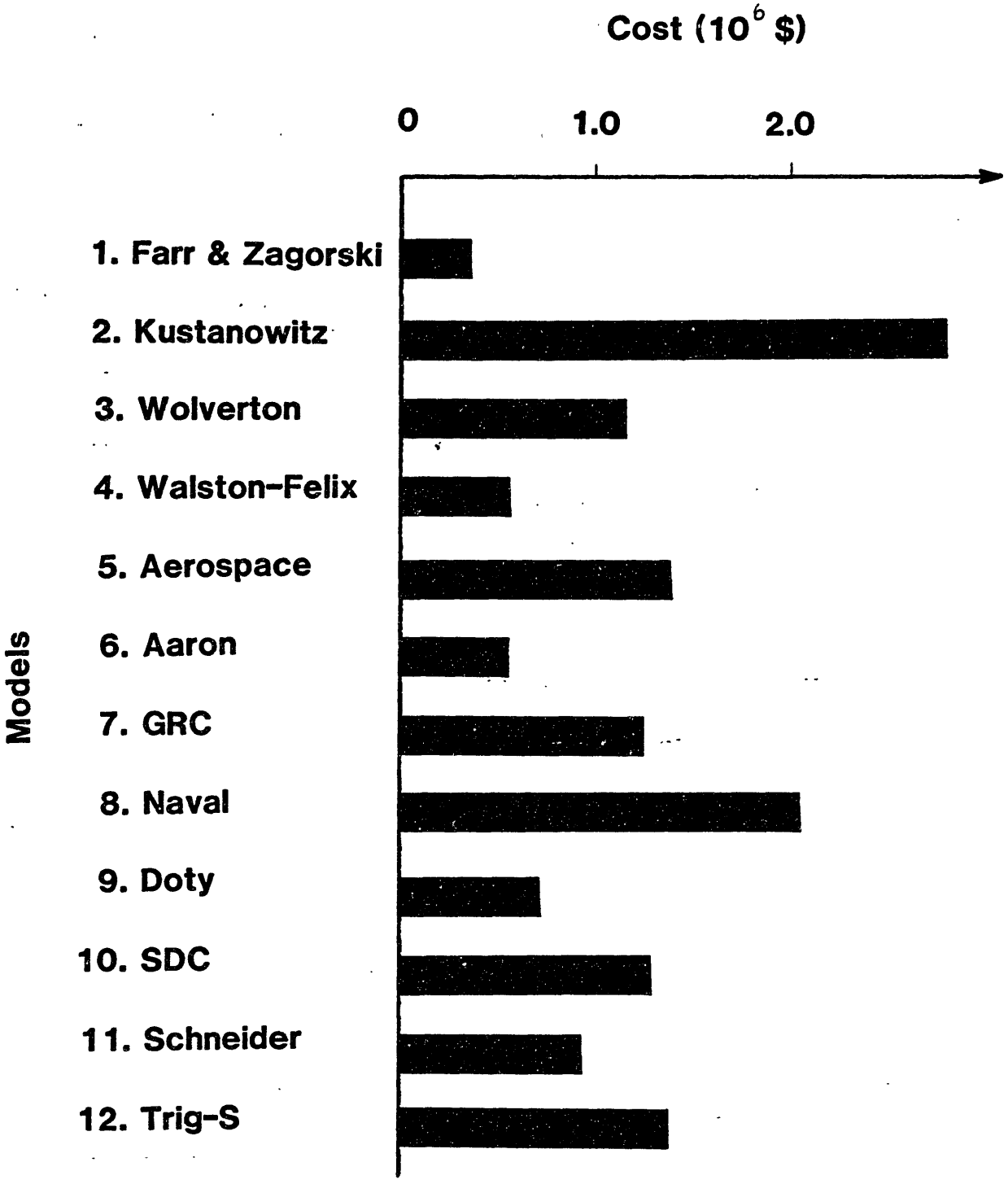

Cost (10^6 $)
0
1.0
2.0
Models
1. Farr & Zagorski
2. Kustanowitz
3. Wolverton
4. Walston-Felix
5. Aerospace
6. Aaron
7. GRC
8. Naval
9. Doty
10. SDC
11. Schneider
12. Trig-S

Figure V.14

cost are obviously caused by other factors. Two sources of variation were suggested by Mohanty. The first related to the quality of the final product. For example, when the costs of highly reliable software are collected into a cost data base, a model that uses this data base will estimate the cost of a reliable product. On the other hand, if the data base reflects software products with low reliability, any model based on it would invariably estimate the cost of a less reliable product. Since the cost data bases used in developing the cost models are different, embodying software with different qualities, one source of variation in estimated cost is the quality of the final product.

The second source of variation suggested by Mohanty is environmental:

> ... That is, each model was developed for a cost data base collected in a given company environment. This data base thus embodies the specific nature of the organizational problems, work patterns, and management approaches and practices. Where this data base is regressed to derive coefficients for use in a given model, the model reflects that company's environment only (Mohanty, 1981).

This contention, on the significance of the managerial and organizational environment, is supported by others in the literature [(Tausworthe, 1977), (Bartol and Martin, 1982), (Pietrasanta, 1968), and (Clap, 1976)]. A few researchers have even suggested some managerial/organizational factors which they feel need to be accounted for in software cost

estimation. For example, Tausworthe (1977) discusses the importance of accounting for manpower turnover, while Clap (1976) argues for the consideration of managerial policy on both the acquisition of manpower and the distribution of effort among the software development activities.

In the remaining part of this section we will discuss the results of a simulation experiment we conducted to quantify the impact of four managerial variables on the cost of software development. Two of the variables address manpower-acquisition and staffing policy issues, while the other two concern issues of effort distribution among the software development activities. The four variables were selected with two criteria in mind. The two criteria were proposed in (Boehm and Wolverton, 1980), and they are: (1) objectivity and (2) prospectiveness. According to Boehm and Wolverton, software cost estimation models should only include objective variables to avoid allocating the software cost variance to poorly calibrated subjective factors (e.g., complexity). That is, the inclusion of only objective variables makes it harder to manipulate the model to obtain any result that one wants. Secondly, a software cost estimation model should avoid the use of variables whose values cannot be determined until the project is complete.

First we will examine the impact of each of the four variables individually. This will then be followed by an

experiment to evaluate the impact of the four variables combined. The result is quite significant: project EXAMPLE's cost varies by a factor of two.

Manpower-Acquisition and Staffing Variables:

As mentioned above, two model variables that address manpower-acquisition type policy issues, will be examined. The two model variables are: (1) "Average Daily Manpower per Staff," and (2) the "Willingness to Change Workforce."

Our interviews at GM and Digital, revealed a difference in the two organizations' software project staffing policies. At GM, project members were assigned full-time to a single project (Hisamune, 15), whereas at Digital, it was more common to assign software people to more than one project (usually two) (Landolfi, 11), (Lombardi, 16). The practice of these two types of policy for staffing software projects has been also reported in the literature, e.g., in (Knutson, 1980). In the model, this staffing issue is captured, as was explained in Chapter III, by the variable "Average Daily Manpower per Staff." For example, when project members are assigned full-time to the project, the value of the "Average Daily Manpower per Staff" would be set to 1 i.e., each project member contributes 1 man-day every (working) day to the project. On the other hand, if project members assign, on the average, only 50% of their time to the project (e.g.,

as is the case with the Digital groups we studied), the value of the "Average Daily Manpower per Staff" would be set to 0.5.

To examine the impact of these two different staffing policies on project cost, we ran project EXAMPLE twice, in the first run the value of the "Average Daily Manpower per Staff" was set to 1, and in the second it was set to 0.5. And we compared EXAMPLE's cost under the two policies. The measure of project cost we will use is simply the value of the total number of man-days expended to complete the project. The results were as follows:

Average Daily Manpower per Staff	Man-Days
1.0	3,795
0.5	4,641

In other words, the policy of allocating project members half-time (on the average) to the project results in a cost that is 22% higher. And the reason for this increase in cost is two-fold. First, there is a loss in productivity due to the increased communication overhead. This factor accounts for approximately 90% of the increase in the project's cost. As was explained earlier, the average staffing level of a project (in terms of full-time equivalent employees) is determined by dividing the estimated value of the projects development time, i.e.,

Staffing Level = MD / TDEV full-time equiv. employees

If the "Average Daily Manpower per Staff" is less than 1, adjustments would then be made to determine the actual number of employees needed. For example, if MD = 1000 man-days and TDEV = 200 days, the average staffing level in terms of full-time equivalent employees would be 5. And if employees will be assigned only half-time, on the average, to the project, then the actual staffing level would be 10 employees. Having 10 people involved in developing the system rather than 5 increases the communication overhead in the project, and, therefore, decreases the group's overall productivity. As was explained in detail in Chapter III, the productivity loss takes two forms. First, more time is lost on human communication, e.g., to resolve questions about design, testing, ... etc. (Tausworthe, 1977). Secondly, the amount of work itself usually increases e.g., in the form of more documentation, more modules and interfaces, ... etc. (Gagliardi, 1981), (Conway, 1968).

The second reason why the cost increases is because of an increase in the training overhead. This second factor accounts for the remaining 10% of the increase in the project's cost. Again, as was explained in detail in Chapter III, when new project members are recruited (from within the organization or from the outside), they pass through a

project orientation period (Brooks, 1974) e.g., to learn the project's ground rules, the goals of the effort, the plan of the work, and all the details of the system (GRC, 1977), (Thayer and Lehman, 1977). This training of newcomers, is usually carried out by the "old timers" (Tanniru, et al., 1981), (GRC, 1977), (Winrow, 1982), (Corbato and Clingen, 1978). This training overhead is, of course, costly, because "while (the oldtimer) is helping the new employee learn the job, his own productivity on his other work is reduced" (Canning, 1977). This training overhead is a function of the number of newcomers, not of the number of equivalent full-time newcomers (Brooks, 1975). For example, in (Gordon and Lamb, 1977) when project members were assigned half-time on the project, the team size was doubled, and as a result the training overhead also doubled. When the "Average Daily Manpower per Staff" is, therefore, less than 1, a larger training overhead will be incurred, because as was shown above, it would mean a larger workforce buildup in terms of actual employees.

The second manpower-acquisition variable we examined is the "Willingness to Change Workforce." In Chapter III we made the following note about the "Willingness to Change Workforce:"

> It is important to realize that the variable 'Willingness to Change Workforce' is an expression of a policy for managing projects. Thus, a range of functions are possible here, capturing different

> strategies for how to balance workforce and schedule adjustments throughout the project to minimize overruns and costs.

Our objective now is to examine the sensitivity of the project's cost to this policy variable. In particular, we will examine two different policies that lie at different sides of the base case policy i.e., the one explained in Chapter III.

The first manpower acquisition policy, we'll call it policy (A), can be defined as follows: At the initiation of the project estimates are made for the project's total effort in man-days (MD), and its development time (TDEV). Based on this, the project's desired staffing level is determined i.e., by dividing MD by TDEV. People are hired, complementing the core of project members on hand at the initiation of the project, until the desired staffing level is reached. Once reached, the workforce is maintained at that level. That is, new people would be hired only to replace either those who quit or are transferred out.

Such a policy was reported by Devenny (1976), in his study of software cost estimation at the Electronic Systems Division of the Air Force Systems Command. He observed:

> The data indicate that none of the ten contractors ever significantly altered the size of the original software team. The contractor will normally keep the initially formed team working until the software is eventually

completed.

In terms of project EXAMPLE, this policy will be implemented as follows: Estimates for the total effort in man-days, the development time, and the staffing level will be calculated exactly as we did before in Section V.2. These values turn out, respectively, to be 2,359 man-days, 296 days, and 8 people. We will also continue to assume that at the project's initiation only half the desired number of people (i.e., 4) will be actually on board. To achieve the desired staffing level of 8 people, 4 more people will then be recruited into the project. Once, that desired level is achieved, it is maintained until the end of the project. That is, new people would be hired only to replace those who either quit or are transferred out.

The result of this policy, together with that of the base run, are tabulated below:

Manpower Acquisition Policy	Man-Days	Duration
Base Case	3,795	430
A	3,559	488

As the figures indicate, Policy (A) leads to a 6% drop in cost (i.e., below the base case). Notice, however, that this is achieved at the cost of a larger schedule slip. Under Policy (A), project EXAMPLE takes 13.5% more time to

complete (i.e., over the base case). Whether this tradeoff is made explicitly and willingly by the Electronic Systems Division contractors is not clear. However, by foregoing the flexibility of adjusting the workforce level to account for any initial errors in estimating the scope of the project, the policy leaves little room to handle any initial under-estimate but to translate them into a software schedule slip. (Remember, project EXAMPLE's size is initially underestimated by 33%, i.e., it is initially perceived as being 42.88 KDSI in size, rather than being 64 KDSI, its true size.) In the base case, on the other hand, when the project's "Undiscovered Job Tasks" are progressively discovered i.e., as project management comes to realize that the project's scope is larger than what has been expected, adjustments are made (as we explained in detail in Section V.2.) not only to the schedule, but to the workforce level as well.

The point here is not to decide which policy is better, since this can only be evaluated on the basis of what an organization's objectives are, but merely to point out that the different policies do impact what the project's cost will end up being, and should, therefore, be explicitly considered when project cost estimates are made.

Under the second manpower acquisition policy we will examine, call it policy (B), project management is not only

willing to adjust the workforce level to account for any initial underestimation error, but it is willing to continue making such adjustments further into the project's life cycle (that is, further than in the base case).

In the base case (and based on discussions with (Lombardi, 23), (Garett, 24) and (Nichols, 25)), the "Willingness to Change Workforce" was formulated in terms of the sum of the "Hiring Delay" and the "Average Assimilation Delay." Specifically, in the early stages of the project when "Time Remaining" would generally be much larger than the sum of the "Hiring Delay" and the "Average Assimilation Delay" management would be willing to adjust the workforce level to meet the project's scheduled completion date. As the number of days perceived remaining drops below 1.5 * (Hiring Delay + Average Assimilation Delay), though, management starts becoming reluctant, and increasingly so, to increase the workforce level. For example if the "Hiring Delay" is 40 working days and the "Average Assimilation Delay" is 80 days, then as "Time Remaining" drops below 180 days, management, in the base case, starts becoming reluctant to hire new people, even though the time and effort perceived remaining might imply that more people are needed. The reluctance stems from the realization that most of those remaining 180 days, would be "wasted" in the hiring process and then in acquainting the new people with the mechanics of the project, in integrating them into the project team, and

in training them in the necessary technical areas. And when the "Time Remaining" drops below 0.3 * (Hiring Delay + Average Assimilation Delay), no more addition would be made to the project's workforce i.e., the hiring rate will fall to zero. Thus, at that stage, if the project is behind schedule, project management would be coping only by pushing back the schedule completion date.

As has been repeatedly, stressed, while the above formulation does express (what we feel is) a representitive policy for manpower acquisition, it is by no means the only policy. A range of policies are possible here, capturing different strategies for how to balance workforce and schedule adjustments throughout the project to minimize overruns costs.

Policy (B) is one such policy. It is adopted by (at least) one group in a Massachusetts-based software development/consulting company. Policy (B) is similar in structure to the policy above, the only difference is that the "Willingness to Change Workforce" is formulated in terms of just the "Hiring Delay." This, of course, means that policy (B) is a more aggressive policy in terms of acquiring people. For example, while in the base case policy, management starts becoming reluctant to increase the workforce level when the perceived number of days remaining to complete the project drops below 1.5 * (Hiring Delay +

Average Assimilation Delay), i.e., below 1.5 * (40+80) = 180 days, under policy (B) this happens much further into the project's life cycle, i.e., when only 1.5 * 40 = 60 working days are perceived remaining. (This aggressive manpower acquisition policy, is justified, we were told, because the firm is experiencing an impressive growth rate, fueled by a sizable backlog of client assignments. Hiring new people to a project that is "winding down" is, therefore, not inhibitted by management since securing the future utilization of the new people is almost always guaranteed.) The WCWF-1 table function for policy (B) is shown in Figure V.15. It has exactly the same form as that of the base case (shown in Figure III.34.), the only difference is that the denominator of the x-axis is simply the "Hiring Delay" rather than being the sum of the "Hiring delay" and the "Average Assimilation Delay."

The result of adopting such a policy in project EXAMPLE is shown below, together with the results of both the base case and policy (A).

Manpower Acquisition Policy	Man-Days	Duration
Base Case	3,795	430
A	3,559	488
B	4,321.5	373

As the figures indicate, policy (B)'s cost is 14% higher

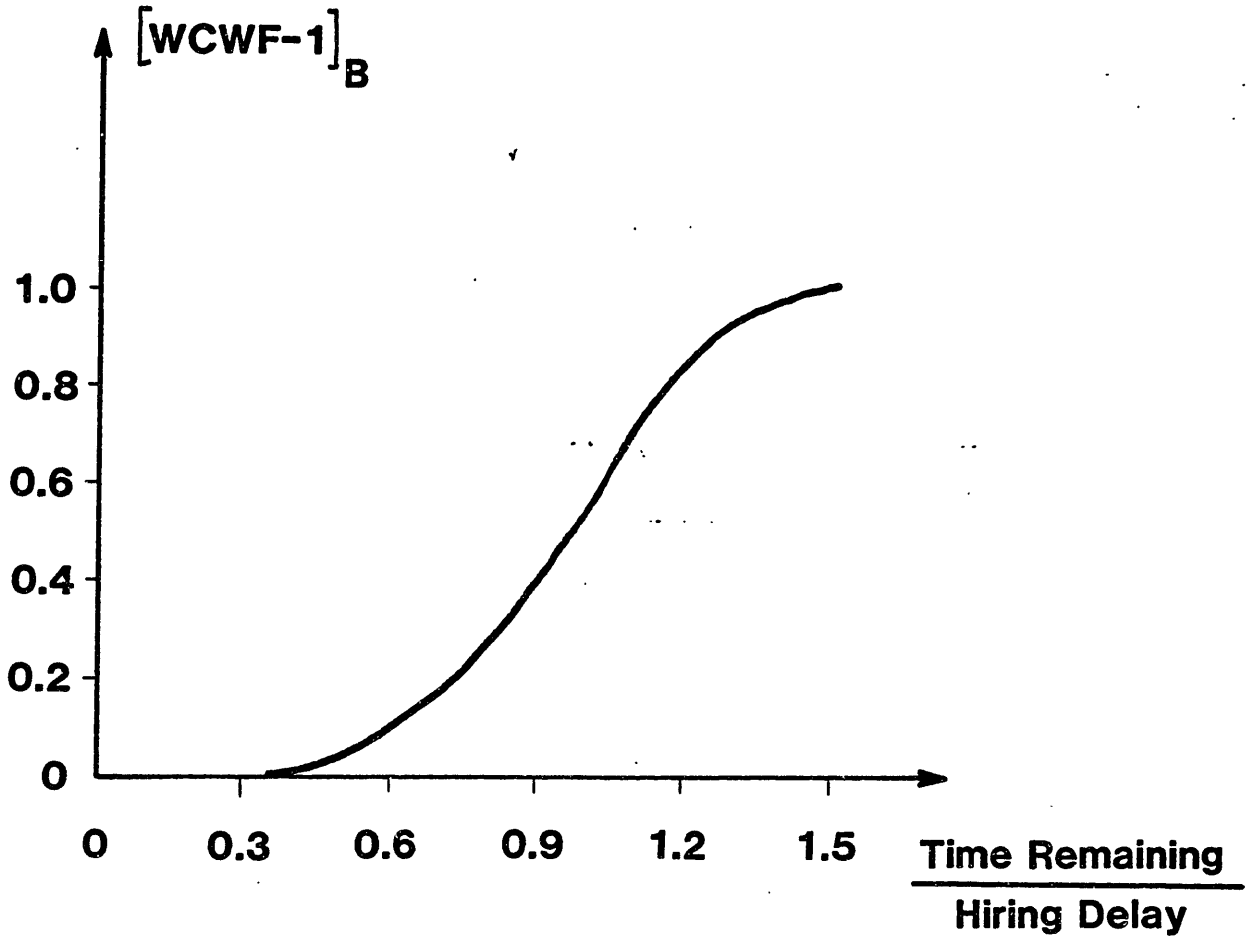

[WCWF-1]B
1.0
0.8
0.6
0.4
0.2
0
0
0.3
0.6
0.9
1.2
1.5
Time Remaining
Hiring Delay

Figure V.15

than the base case, and 21% higher than that of policy (A). On the other hand, under policy (B), the project takes 13% less time to complete than the base case, and almost 25% less time than when policy (A) is used. Both the increase in the cost and the decrease in the duration can be attributed to a single cause, namely, a higher workforce level. More people on the project means more work can be done faster. It also means that the project team's overall productivity would be lower because of the increased communication and training overheads.

Once again, it is important to reiterate that the objective of this exercise is not to decide which policy is better, since this can only be decided on the basis of what an organization's objectives are, but merely to establish that manpower acquisition policy does have an impact on what the project's costs will end up being, and should, therefore, be explicitly considered when project cost estimates are made.

From a pragmatic point of view, establishing the significance of a particular factor for cost estimation purposes is not enough. The factor must also be quantified, before it can be used in a quantitative cost estimation model. For example, paraphrasing Clapp (1976):

> Variables used in cost estimation tend to be those which are easier to measure, quantify, and estimate, even if

they are not the most significant.

We feel that our "Willingness to Change Workforce" table function formulation does provide the software engineering community with a valid measure of manpower acquisition policy that is also easy to measure. We must note, however, that this measure is not an original one, for it has been previously used in other System Dynamics models e.g., of R&D project management (Roberts, 1964). Our role, here, is, therefore, that of transferring a useful idea from the System Dynamics field to the software engineering community.

A final note. Notice that our results above seem to contradict Brooks' Law, which states that "Adding manpower to a late software project makes it later" (Brooks, 1978). The most aggressive of the three policies in terms of adding manpower, namely, policy (B), actually leads to the earliest completion date. What our results indicate is that "adding manpower to a late software project makes it more costly." More on this later in this chapter.

We turn next to the second category of variables, those addressing issues of effort distribution among the software development activity.

Effort Distribution Variables:

In planning a software project, management does not only provide estimates for the project's total man-days expenditure, it in addition plans the distribution of this total effort among the project's phases (McKeen, 1983), (Davis, 1974), (Gunther, 1978). Numerous authors have presented figures indicating life cycle resource distributions among phases. In some cases the source of their information has been reported; in most instances, it has gone unreported causing some difficulty with interpretation and application. In Figure V.16. a comparison of three authors' results, done by McKeen (1981), indicates that substantial differences do exist particularly in the coding and testing phases of development. Commenting on the situation, McKeen (1981) wrote:

> A major conclusion ... is that we do not possess an adequate understanding of resource consumption behavior over the life cycle development phases.

In McKeen's own research work, he studied 32 software development projects. He found "no real support ... for 'typical' or 'dominant' development profiles at all" (McKeen, 1981).

In this section, it is our objective to enhance our understanding of the "resource consumption behavior." In particular, we will investigate the impact of planned effort distribution among the project's phases on project cost.

Portions of the text
on the following page(s)
are not legible in the
original.

Comparison of Effort Breakdown by Activity for Different Authors

Life Cycle Phase/Activity	Percentage Resource Allocation		
	Davis	Zelkowitz	Shaw
Analysis[1]	25	20[2]	25
Design	20	15	10[3]
Coding	25	45[4]	30
System Test	n/a[5]	20	5
Implementation	15	n/a[6]	19

Notes:
1. Analysis encompasses all development activity prior to detailed design.
2. The analysis effort is probably understated. If, as speculated, this data is derived from system developments in a military environment, then initial activity such as feasibility analysis and preliminary systems study has been excluded.
3. Using the authors definitions, the activities of system specifications and technical requirements constitute detailed design activities as used here.
4. Coding effort and module test effort were combined. Programmers are typically responsible for unit, or module, testing each portion of the system they have coded.
5. This activity has been subsumed within the conversion stage by Davis.
6. This activity is not reported.

Figure V.16

Thibodeau and Dodson (1978) were the first to hypothesize the existance of such an impact:

> Past attempts to establish mathematical expressions that can predict the life cycle cost components for software systems have achieved only qualified success. The mathematical models for these relationships included only variables that describe the software characteristics and related environmental factors. This paper presents the hypothesis that software cost estimating relationships must include the effects of resources consumed in one life cycle phase on other phases. Such a model is difficult to validate. This is primarily due to the need for greater quantities of data of greater precision than is usually available.

In our view, the difficulty arises because of the phenomenon we discussed in detail in Section V.3.1., namely, that a different project estimate creates a different project. While, all the arguments we presented in Section V.3.1. were in terms of a project's <u>total</u> effort estimate, they do equally apply to estimates at the phase level. We can, therefore, restate the above assertion as follows: <u>A different distribution of estimated effort among a project's phases creates a different project</u>. And because of this, the impact of different effort distributions on the cost of a particular software project can only be determined by repeating the particular project under controlled conditions in which only the distribution of estimated effort among the project's phases would be allowed to change.

In the remainder of this section we will use the model to conduct an experiment using our prototype project EXAMPLE

to examine the impact of the distribution of effort among the project's activities on project cost. Again remember the objective of this exercise is not to determine what the optimal effort distribution is, but rather to establish that effort distribution decisions do have an impact on what the project's cost will end up being, and should, therefore, be explicitly considered when project cost estimates are made. (Optimal effort distributions will be examined in another experiment later in this chapter.)

The model has two effort distribution parameters. The first parameter allocates the project's estimated man-days among the model's two explicit phases, namely, development (which includes design and coding) and system testing. In the base case 80% of the effort is allocated to development and 20% to testing. The second effort distribution parameter is the "Planned Fraction of Manpower for QA," which is set to 15%. That is, 15% of the development effort is planned for QA activities during the design and coding stages. As was explained in Section V.2., these values were selected to conform to the TRW software development environment.

The selection of another effort distribution profile to experiment with and compare to the base case distribution was, in a sense, both easy and difficult. It was easy, because there was a large number of candidate profiles. As the remarks in the beginning of this discussion indicate,

there is a wide range of effort distribution profiles reported in the literature. However, the selection of an effort distribution profile was difficult, because, of the many that are reported, none seemed to be "typical" or "dominant" (e.g., as McKeen's (1981) study indicates). We finally decided to make our selection on the basis of our own data i.e., the data collected in our interviews. And from this we selected the case which we felt would provide the most interest. It involved one group at GM using the 40-20-40 effort distribution profile i.e., 40% for preliminary and detailed design, 20% for coding, and 40% for testing. We feel that this particular profile would interest many in the software engineering area because of the fact that this 40-20-40 rule is perhaps the most widely touted rule-of-thumb on the distribution of effort among the phases of software development projects (McKeen, 1981), (Bruce and Pederson, 1982), (Oliver, 1982), (Jensen and Tonies, 1979).

In terms of our model's effort distribution parameter this translates into a 60-40 distribution. That is, 60% of the total man-days would be allocated to development (i.e., design and coding) and 40% to system testing. As for the QA effort, the GM group allocated to it 20% of their development effort. This translates into a 0.20 value for the models "Planned Fraction of Manpower for QA."

The result of running project EXAMPLE with this new

effort distribution profile, call it (C), were as follows:

Effort Distribution Profile	Man-Days
Base Case	3,795
C	4,442.5

Thus, a change in project EXAMPLE's effort distribution profile from the base case to profile (C) leads to a 17% increase in cost. Four factors contributed to this increase in cost. The first obvious one is the (planned) increase in the QA effort. Secondly, and as a result of this increased QA effort, more errors were detected during development leading to a larger rework effort expenditure. Thirdly, the cost of development increased. The reason for this is, however, less obvious. Recall the sequence of steps followed in planning a project's various activities. First, total man-days is determined. Based on this total value, the project's schedule is calculated. Allocations to the development versus testing activities are then made. What this means is that, since this run's total man-day estimate is the same as that of the base case, the scheduled duration would also be the same in both cases. However, since in the current case a lower fraction of the manpower is devoted to development work, a larger team will be required to meet the schedule. A larger team means larger training and communication overheads, and hence the larger development cost. The fourth, and final factor, is an increase in the

testing effort. Notice that the testing effort increases (i.e., over the base-case situation) even though it "should" be lower. It should be lower because more effort was devoted to QA leading to the detection of a larger fraction of the errors. The testing effort increases inspite of a lower testing workload (because of the lower errors) because of a lower testing productivity. In the base-case, project members had to over-work during the testing phase, because there were more errors and less time. In the current case, on the other hand, there is more time, and the work expands to fill it.

What the above suggests, is that (for an EXAMPLE-type software project) if 40% of development effort is allocated to the testing phase, a 20% allocation to QA would be excessive. Or conversely, for a 20% allocation to QA, a 40% testing phase is excessive. What is more interesting, and would be more useful, to determine, of course, is the "optimal" combination. This will be investigated in Section V.6.

A Final Experiment:

Our objective in this section was to demonstrate the significant impact of a number of managerial variables on the cost of software developemt. We examined four managerial variables. Two variables related to the acquisition and

staffing of the project's workforce, namely, the "Average Daily Manpower per Staff" and the "Willingness to change Workforce." The other two variables concerned the distribution of effort among the project's different activities i.e., development, testing, and QA. The individual impact of the different variables on the project's cost was evaluated in separate experiments (except for the 2 effort distribution variables which were tested together). The results indicate that, individually, the variables can make as much as a 20% difference in project EXAMPLE's total cost (in man-days). What we would like to evaluate next, in this final experiment, is the combined effect of the four managerial variables on cost.

This is achieved by re-running project EXAMPLE with the following four adjustments:

1. Set the value of the "Average Daily Manpower per Staff" to 0.5. (The base-case value is 1.)

2. The "Willingness to Change Workforce" is formulated in terms of the "Hiring Delay," yielding a more aggressive manpower acquis ion policy. (In the base-case it is formulated in terms of the (Hiring Delay + Average Assimilation Delay).)

3. Allocation of effort among the development and

testing phases is set at 60% development and 40% testing. (In the base case it is 80-20.)

4. The "Planned Fraction of Manpower for QA" is set at 20%. (In the base-case it is 15%.)

The result of running project EXAMPLE with this different set of managerial policies is a total cost of 7,316 man-days. That is, a cost that is almost double the base-case cost of 3,795 man-days.

The implication of this significant result is clear: Because the above four managerial policies do vary from software development organization to another, the portability of software cost estimation models can be improved significantly if such variables are accounted for. Recall Mohanty's (1981) comments:

> ... each (cost estimation) model was developed for a cost data base collected in a given company environment. This data base thus embodies the specific nature of the organizational problems, work patterns, and management approaches and practices. When this data base is regressed to derive coefficients for use in a given model, the model reflects that company's environment only.

Heretofore, the impact that a company's managerial environment can have on the software development has not be quantified. We feel that our work can be useful in three aspects. First, we have established that the impact is a

significant one i.e., we have shown that the effect of four managerial variables can modify the cost of a software project by a factor of 2. Second, by quantifying the impact, we are making it harder on the software engineering community to ignore the issue. And, finally, we have identified four aspects of a company's managerial environment that are significant determinants of software development cost, and which are, therefore, deserving of future research efforts.

V.3.3. On the Analogy Method of Software Estimation:

While in the previous section our focus was on the state-of-the-art software estimation methods, namely, the quantitative models, in this section we turn our attention to the "state of the practice." In this section we focus on "Estimation by Analogy," probably the most commonly used method to estimate software projects.

Estimation by analogy is defined as follows:

> Estimation by analogy involves reasoning by analogy with one or more completed projects to relate their actual costs to an estimate of the cost of a similar new project (Boehm, 1981).
>
> To employ this method at least one project with similar features must have been completed previously. The new project must be clearly specified at least at the functional level, permitting comparison of similar elements (Benbasat and Vessey, 1980).

According to Aron of IBM, when methods of estimating are ranked, the list is headed by the analogy method (Aron, 1976). More recently, Oliver (1982) wrote: "The most common technique on making operational estimates is the use of experience gained on one or more similar projects." These assertions are supported by at least one empirical study. In his Ph.D. dissertation, Thayer (1979) surveyed 60 software development projects in the aerospace industry, and found that the analogy method was used in 60% of the cases, making it, by far, the most common estimation method used.

In the previous sections, we argued that software project estimation affects project behavior. That a project's estimate creates pressures and perceptions that directly influence the decisions that people make, and the actions they choose to take, throughout the project's life cycle. For example, the causal loop diagram of Figure V.17. depicts the influence of project estimation on hiring/firing decisions, perceived project status, and productivity. What this implies for the use of analogy in estimation is the existence of a feedback loop (see Figure V.18.): The estimation by analogy method produces project estimates and schedules, which affect the decisions and actions of the technical performers and their managers, which in turn affect work performance, which would then eventually influence future estimations.

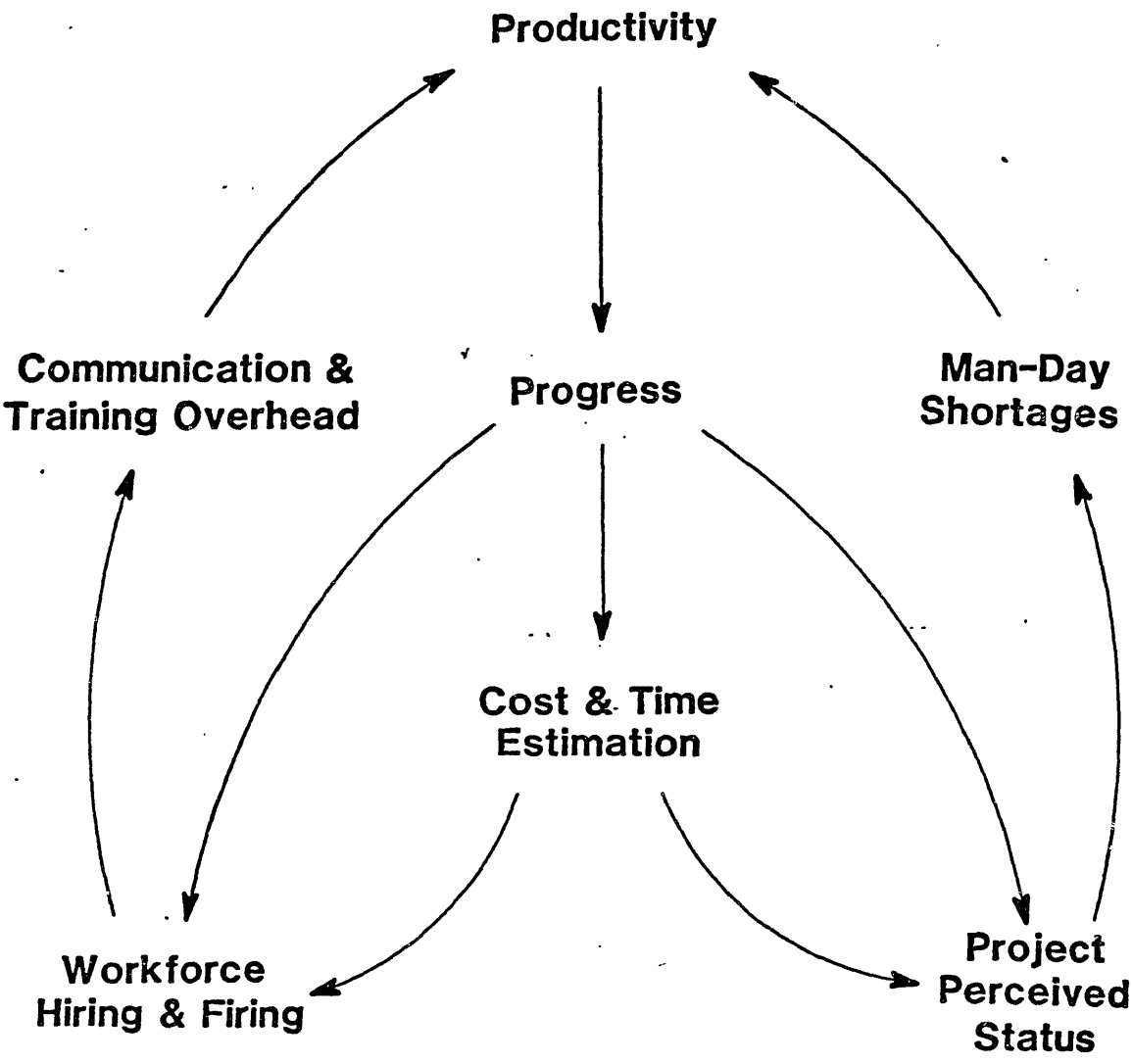

Productivity
Communication &
Training Overhead
Progress
Man-Day
Shortages
Cost & Time
Estimation
Workforce
Hiring & Firing
Project
Perceived
Status

Figure V.17

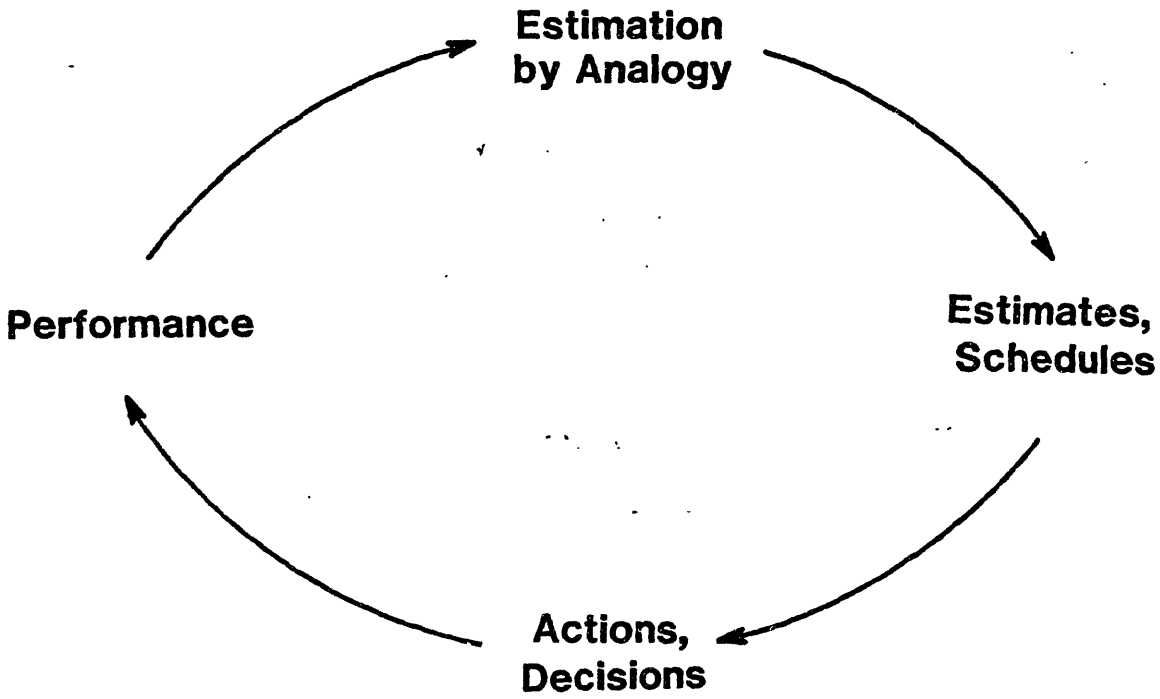

Estimation
by Analogy
Estimates,
Schedules
Actions,
Decisions
Performance

Figure V.18

But what does the existence of such a feedback loop mean? Is it good or bad? These are some of the questions which we will attempt to answer in this section's simulation experiment.

The experiment involves a hypothetical situation in which a company undertakes a sequence of five identical software projects, all identical to project EXAMPLE, our prototype project. On the first such project, and let us call it EXAMPLE1, the company (lacking the benefit of previous experience) underestimates the size of the project by 33%, that is, estimates the project's size to be only 48.22 KDSI, i.e., as in project EXAMPLE's base-case. And let us also assume that the base-case estimates for the project's man-days and duration were the estimates used in EXAMPLE1. That is, the project's man-days are estimated to be 2,359, and its development time is estimated to be 296 working days. In other words, EXAMPLE1 is conducted under our base case conditions.

As our base case analysis of Section V.2. indicates, EXAMPLE1 will end up actually consuming 3,795 man-days, and will be completed in 430 working days. After completing EXAMPLE1, the following is, therefore, learned:

* Project EXAMPLE1 is really 64 (and not 42.88) KDSI.
* It consumes 3,795 man-days.

* It takes 430 days to complete.

Some time later, when project EXAMPLE2 (which is identical to EXAMPLE1) comes along, project management will be in a better position to estimate its true size. In fact, we will assume that EXAMPLE2's size will be estimated perfectly, that is, to be 64 KDSI. Furthermore, realizing the analogy between the two projects, EXAMPLE1 and EXAMPLE2, management will estimate EXAMPLE2's man-days and duration to be 3,795 man-days and 430 days respectively i.e., the actual values for EXAMPLE1. Based on these figures, management estimates that a staff size of 3,795/430=9 (approx) full-time equivalent people will be required.

Conducting project EXAMPLE2 under such circumstances produces the following results: actual man-days expended = 3,787, and actual duration = 454 days. That is, while the project is almost perfectly on target in terms of the man-day expenditures, it still finishes late, approximately 6% beyond the "improved" schedule.

This result is not only surprising, it is also disturbing, the reason being that project EXAMPLE2 over-runs what amounts to be a "perfect" schedule estimate. And when we repeated the above sequence of actions and reactions three more times for projects EXAMPLE3 through EXAMPLE5, this surprising behavior persisted. That is, the schedule was

overrun in each case. As a result, project 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 an even longer schedule duration for the next project. The results for the five simulation runs are shown in Figure V.19.

It is important to pause here, and make one important clarification. The objective of this experiment is not to investigate the behavior of a sequence of five identical software projects! Such a scenario is admittedly unrealistic (recall that we carefully labelled our experiment as being hypothetical). Choosing to conduct such an "unreasonable" experiment, and being able to do so, is, however, one of the strengths of simulation modeling. For it allows us to conduct experimentation with absolute control over variables. Remember, our objective is to study the effects of using analogy in estimation on the management of software projects, and the effects of that on future scheduling. And only that. Studying such relationships in a setting where projects and managers vary (albeit more realistic) can only and unnecessarily confuse the issues and complicate the analysis. For example, in our experiment when project EXAMPLE2 overruns, we can definitively rule out under-scheduling as a cause, and instead look for a "better" explanation. If, however, EXAMPLE2 had not been identical to EXAMPLE1, we

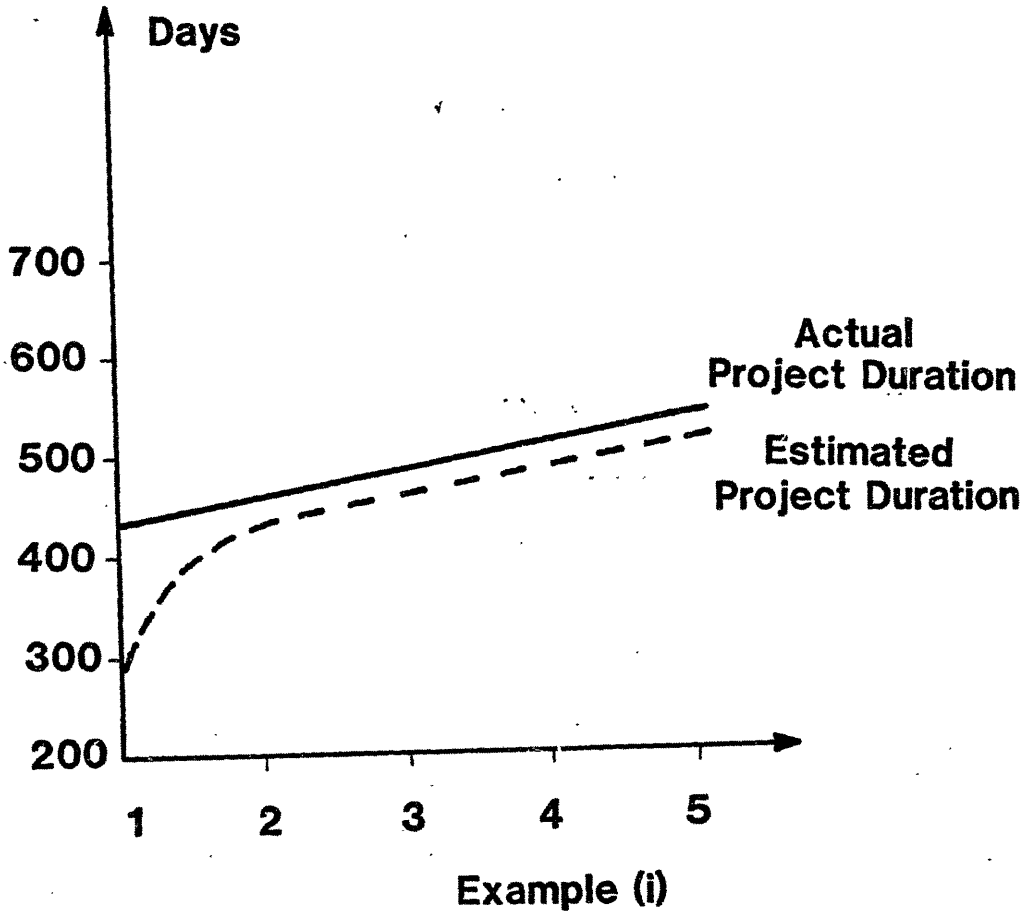

Days
700
600
500
400
300
200
1
2
3
4
5
Actual
Project Duration
Estimated
Project Duration
Example (i)

Figure V.19

would not have been able to make such an argument. Instead we would have had to make unnecessary diversions, e.g., to investigate the differences in scope between the two projects.

With this in mind, we can now proceed to interpret the experiment's results. There are two. First, there appears to be inherent factors in the management of a software project that would cause it to over-run even what amounts to a "perfect" schedule estimate. The second, more interesting finding, is that because of this inherent tendency to overshoot, the use of the analogy method in estimating would inject a bias in the scheduling process, a bias that generates in the long-run longer (than necessary) project schedules.

Concerning the first result, we have already noted that project EXAMPLE2 over-runs a schedule that was perfectly adequate to complete EXAMPLE1, which is a project identical to it (i.e., to EXAMPLE2). Through further experimentation with the model, it was possible to isolate the real cause of this persisting schedule-overrun problem. It turned out to be a consequence of the interaction of two factors, the manpower-acquisition policy and the turnover of project personnel.

As was explained in detail in Chapter III, in the

earlier stages of the project, the staffing of the project is maintained at that level which is perceived to be necessary and sufficient to complete the project on time i.e., on its (current) scheduled completion date. As the project proceeds towards its final stages, however, project management becomes increasingly reluctant to hire new people. This reluctance stems from the realization that most of the time remaining on the project would be "wasted" in the hiring proces and then in acquainting the new people with the mechanics of the project, in integrating them into the project team, and in training them in the necessary technical areas. If at that stage, the project runs into schedule problems, management would react, not by adding more people, but rather by pushing the schedule completion date back.

A project runs into scheduling problems whenever the "man-Days Perceived Still Needed" to complete the project exceeds the "man-Days Remaining." In previous sections we discussed how this can develop due to an increase in the former. For example, if the project's size was under-estimated, the value of the "Man-Days Perceived Still Needed" could rise as the undiscovered tasks are discovered. In our current situation (e.g., in project EXAMPLE2), though, this will not occur. Remember, we are assuming that the experience gained on project EXAMPLE1 will lead to a "perfect" estimate of EXAMPLE2's size. What can happen, however, is that the value of the "Man-Days Remaining" for

project EXAMPLE2 drops below the value of the "Man-Days Perceived Still Needed," and when this happens, EXAMPLE2 would run into scheduling problems. The value of the "Man-Days Remaining" is simply the product of the workforce level (in full-time equivalent employees) and the time remaining in the schedule. Thus, any drop in the workforce level due to turnover will, in turn, decrease the value of the man-days remaining, creating a scheduling problem. And when this happens towards the end of the project, when management is reluctant to add new people, the adjustment that will be made will be to push the scheduled completion date back i.e., resulting into a schedule over-run.

Thus far, we have been addressing only the first result of our experiment, namely, that a software project can still over-run what amounts to a "perfect" schedule estimate. The second result of the experiment can be stated as follows: because of the inherent tendency to overshoot, the use of the analogy method in estimating would inject a bias in an organization's scheduling process, a bias that generates in the long-run longer (than necessary) project schedules.

The "surprising" phenomenon we are observing here (i.e., of projects consuming longer and longer schedules), is a phenomenon that has been frequently encountered in system dynamics studies of organizational behavior (Sterman, 1981). It has been termed "The 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.

Richardson and Pugh (1981) provide an explanation for why social systems have this tendency to resist policies designed to improve behavior (e.g., why a software project would tend to resist the policy of estimation by analogy which is designed to solve the schedule over-run problem, and continues to over-run its schedule):

> (The) compensating feedback is a property of real systems, as well as system dynamics models, and is the reason real systems tend to be resistent to policies designed to improve behavior ...
> (A) parameter change may weaken or strength a feedback loop, but multi-loop nature of a system dynamics model naturally strengthens or weakens other loops to compensate. The result is often little or no overall change in model behavior.

In terms of our software project situation this is exactly what happens. To see how, let us first recall the steps followed to estimate a project. First, the estimates

of the project's man-days and its duration are made. These can be made using analogy, COCOMO, ... etc. On the basis of these two estimates, the project's average staffing level is calculated i.e., by dividing the man-days estimate by the estimate for the development time. For example, in EXAMPLE2 the estimates were MD = 3,795, TDEV = 430 and the average staffing level = 3,795/430 = 8.8 full-time equivalent employees. And we also know that EXAMPLE2's actual man-days and duration end up being 3,787 and 454 respectively. From these figures, we can also calculate EXAMPLE2's actual average staffing level, namely, 3,787/454 = 8.3 full-time equivalent employees. When the analogy method is then used to estimate EXAMPLE3, EXAMPLE2's actual values will be used, yielding: MD = 3,787, TDEV = 454, and an average staffing level of 8.3 full-time equivalent employees. Notice what is happening: EXAMPLE2's actual average staffing level ends up (because of the turnover problem) to be slightly less than what was planned for i.e., 8.3 instead of 8.8, and the actual (lower) value is the one passed over to the next project. In terms of Richardson and Pugh's explanation: extending the project's schedule (from 430 to 454) weakens the strength of the schedule pressure in the system, to which the hiring loop simply compensates by causing the project to start with a small workforce level target. It is also important to note that such compensating behavior is often invisible to the participants. For example, it is quite unlikely that EXAMPLE3's project managers will realize such compensating

behavior because, for one, the 8.8 figure is only a planning (not an actual) figure for EXAMPLE2. It is quite possible, therefore, that it would not be preserved in any project records. And even if it is, it is unlikely that EXAMPLE3's manager will use it, after all, by concentrating on EXAMPLE2's actual data, the manager would be behaving in what appears to be the rational way.

It is interesting to note, that this managerial dilemma is not at all unique to the management of software projects. Paraphrasing Forrester (1971):

> ... social systems are inferently 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 dynamic of a larger system.

In the case of software development, where a project over-runs its schedule, the situation provides us with an apparent cause, namely, that the project was poorly estimated. It is a cause that is quite plausible according to what we have learned e.g., that software estimation is not yet an exact science. Furthermore, and this is significant, it is often impossible in a real life situation to

demonstrate that under-estimation was not in fact the cause. (Note: Remember, we are excluding changes in requirements from our analysis.)

Conclusion:

A number of conclusions can be drawn from our "laboratory" experiment on the analogy method for estimating software projects:

* A software project can still over-run what amounts to a "perfect" schedule estimate.

* The software engineering community needs, therefore, to expand its research agenda on the causes of the schedule over-run problem, that is, beyond its current (limited) agenda on software estimation accuracy.

* We have identified one such cause, namely, the interaction of the manpower-acquisition policy and personnel turnover.

* Estimating by analogy injects a bias in an organization's scheduling process, a bias that generates, in the long run, longer (than necessary) project schedules.

V.4. The "90% Syndrome:"

In this section, we will focus on one control-type problem faced by many software project, managers, namely, the "90% syndrome" problem. Specifically, our aim is to demonstrate the model's capacity to generate this important phenomenon of software project management, and in the process provide some insights into its causes.

There is ample evidence in the literature to support the pervasiveness of the "90% syndrome" problem in the management of software development projects (e.g., see (Baber, 1982), (DeMarco, 1982), (Synnott and Gruber, 1981), and (Devenny, 1976).) Baber (1982) provides the following description of the problem:

> ... estimates of the fraction of 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.

To examine the model's capacity to generate the "90% syndrome" type of behavior, we simulated project EXAMPLE with three different initial conditions:

1. The base case, where the size is initially under-estimated by 33%. That is,

SIZE = 42.88 (and not 64) KDSI

MD = 2,359 man-days

TDEV = 296 days

2. When its size is properly estimated, but its man-days requirements are under-estimated by 33%. Such a situation could arise due to an under-estimate of the project's complexity or an over-estimate of the team's productivity, or both. As was mentioned before, COCOMO exists in a hierarchy of increasingly detailed forms. In its more detailed versions, the estimate of a project's man-day requirements can be adjusted by a number of multipliers to account for factors such as complexity, required reliability, team's capability, ... etc. For example, for a project that is perceived to have a "very low" complexity rating, the man-days estimate would be 30% below the "nominal" case. Thus, if a project is incorrectly perceived at its initiation as being "very low" in complexity, when in fact it is not, an under-estimate of its man-day requirements will result.

Thus, for this second case.

SIZE = 64 KDSI

$$MD = 0.67\ (MD_{NOMINAL})$$
$$= 0.67[2.4*19*(64)^{1.05}]$$
$$= 2,407 \text{ man-days}$$

$$TDEV = 47.5 * (2,407/19)^{0.38}$$
$$= 299 \text{ days}$$

3. When neither size nor man-day requirements are

under-estimated. In this case,

$$SIZE = 64 \text{ KDSI}$$

$$MD = 2.4 * 19 * (64)^{1.05} = 3,593 \text{ man-days}$$

$$TDEV = 47.5 * (3,593/19)^{0.38} = 348 \text{ days}$$

The results of these three simulation runs are shown in Figure V.20.

One result was expected, namely, that the "90% syndrome" arises only when a software project is initially under-estimated. Because of the lack of visibility in the earlier phases of development, progress is measured by the rate of expenditure of resources rather than by some count of actual accomplishments. By measuring progress by the rate of expenditure of resources, status reporting ends up being nothing more than an echo of the project's plan. This creates the "illusion" that the project is right on target. However, as the project approaches its final stages (e.g., when 80-90% of the resources are consumed), discrepancies between % of tasks accomplished and % of resources expended become increasingly more apparent. At the same time, and as the project advances towards its final stages, the project members become increasingly able to perceive how productive the workforce has actually been. This results in a better and better appreciation of the amount of effort actually remaining. As this appreciation develops, it would, in

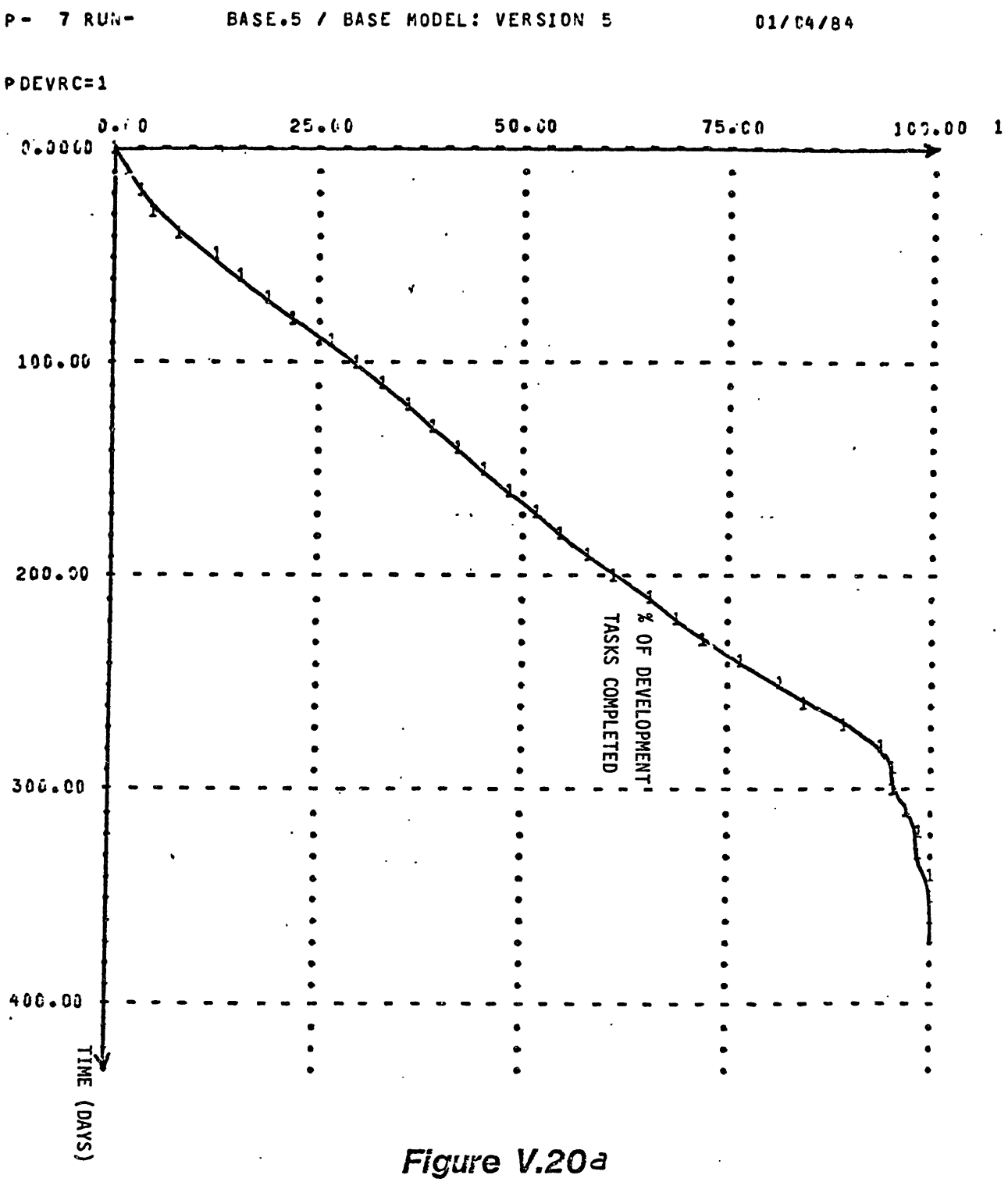

P- 7 RUN- BASE.5 / BASE MODEL: VERSION 5 01/04/84
PDEVRC=1
0.00
25.00
50.00
75.00
100.00
0.0000
100.00
200.00
300.00
400.00
% OF DEVELOPMENT TASKS COMPLETED
TIME (DAYS)

Figure V.20a

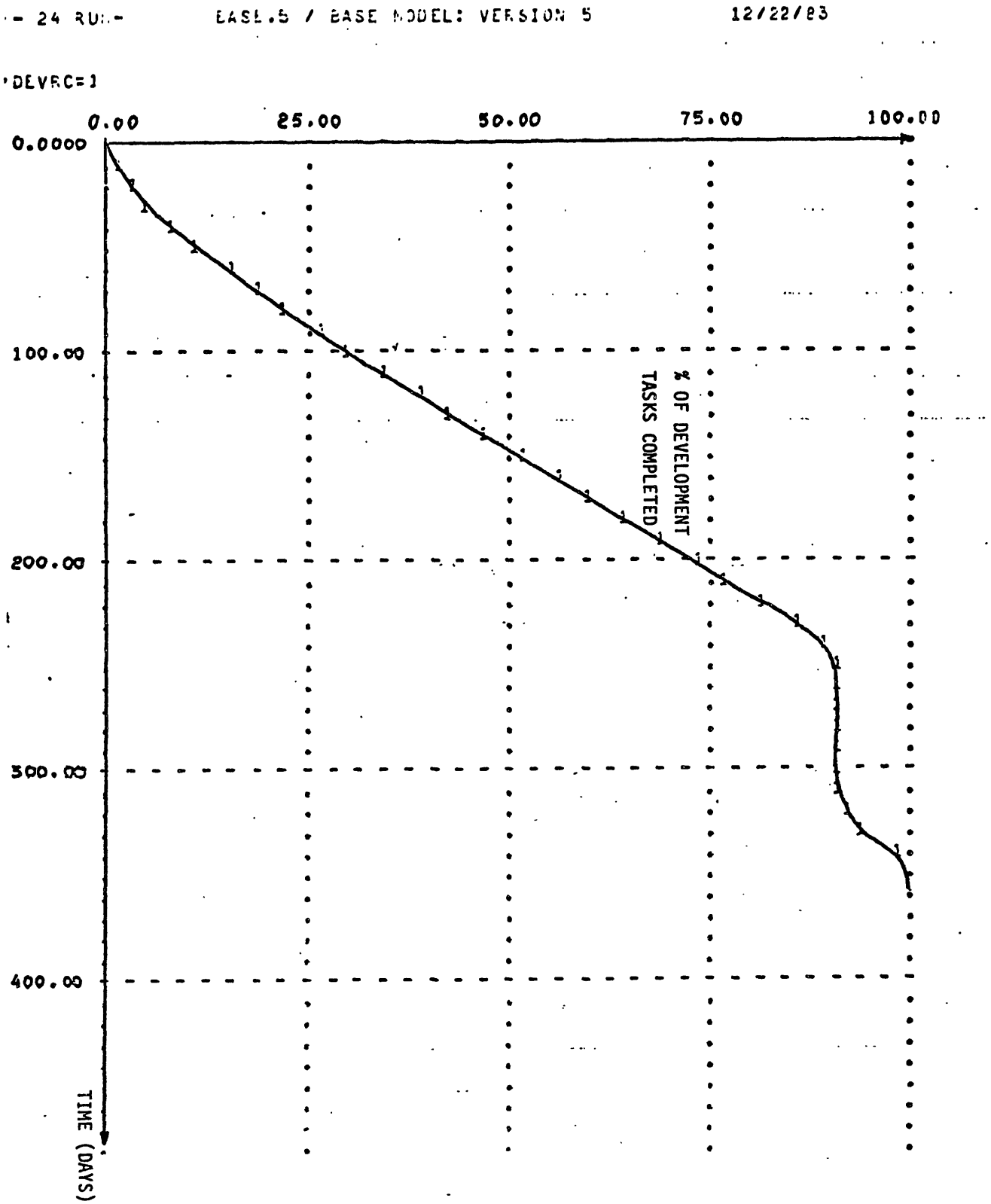

- 24 RU -
BASE.5 / BASE MODEL: VERSION 5
12/22/83
*DEVRC=1
0.00
25.00
50.00
75.00
100.00
0.0000
100.00
200.00
300.00
400.00
% OF DEVELOPMENT TASKS COMPLETED
TIME (DAYS)

Figure V.20 b

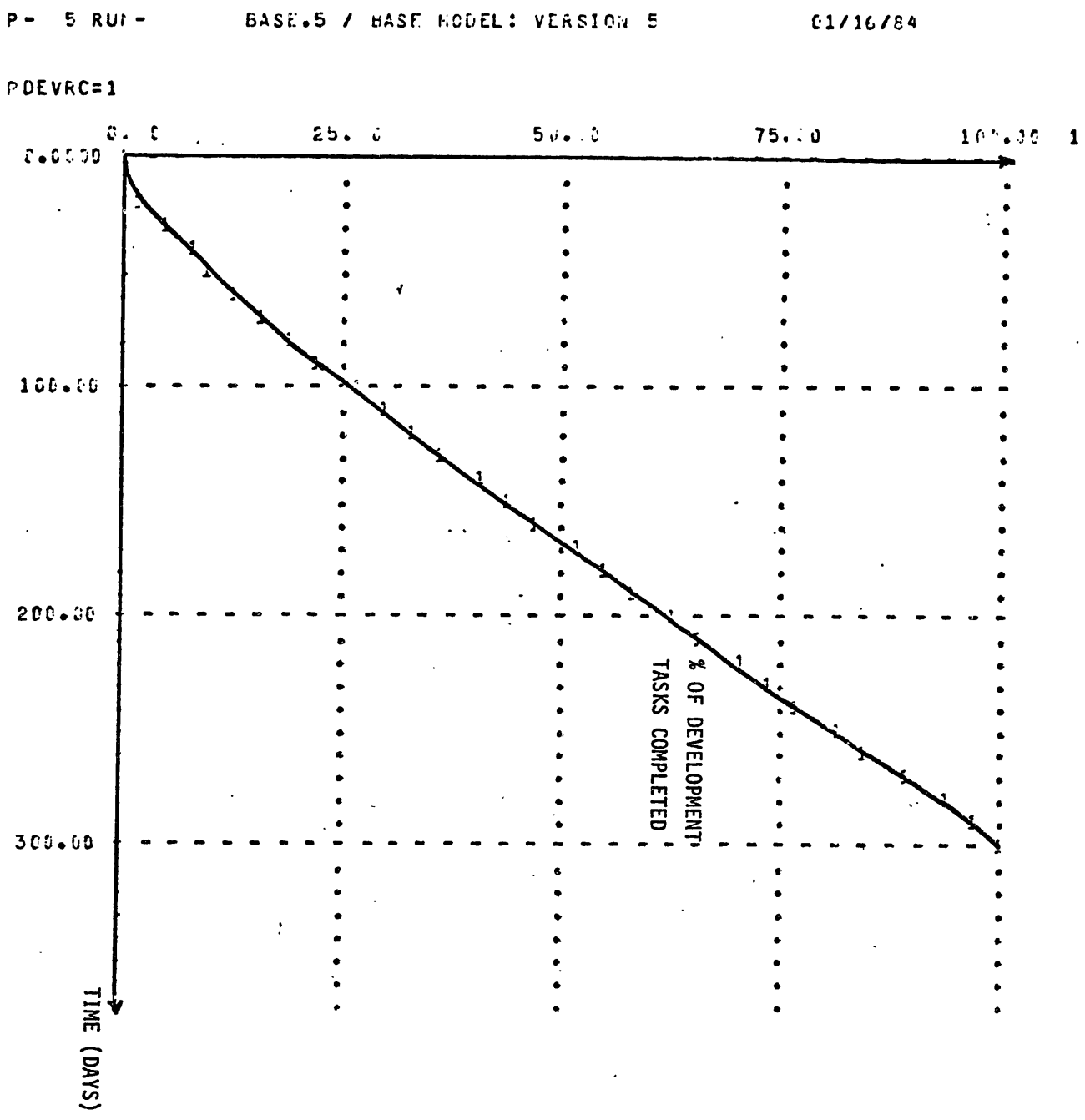

P- 5 RUN - BASE.5 / BASE MODEL: VERSION 5 01/16/84
PDEVRC=1
0.0000
25.00
50.00
75.00
100.00
100.00
200.00
300.00
% OF DEVELOPMENT TASKS COMPLETED
TIME (DAYS)

Figure V.20c

effect, be discounting the project's progress rate. Thus, as the project members proceed towards the final stages of the project, perhaps at a higher work rate, their net progress rate slows down considerably. This continues, until the project completes.

What, however, was unexpected, was the significant difference in the acuteness of the problem between the two types of under-estimates. Notice that the "90% syndrome" is much more acute when the project's man-days requirements are underestimated than it is when the under-estimate is in the project's size. With a little reflection we can see why. When the project's man-days requirements are under-estimated the problem would often remain largely undetected (as was explained above) until the final stages when, first, most of the project's resources (i.e., budgeted man-days) are consumed, and second the project members become more able to perceive how productive the workforce has actually been. When, on the other hand, the initial under-estimate is in the project's size, the situation is, in a sense, less severe. And the reason for this is that the problem tends to be detected faster. As we saw in project EXAMPLE's base case behavior in Section V.2., the "Undiscovered Job Tasks" do not remain undiscovered until the very last stages of the project, but, instead, start to be discovered in a significant way during the detailed design phase of the project. Any new task that is discovered is by definition

visible. And as we saw, when such tasks are discovered, adjustments to the project's man-days are often made. As a result of this, the project arrives at its final stages with its initial under-estimate largely detected, which, in turn, reduces the severity of the "90% syndrome" experienced.

Some Concluding Remarks: The "90% syndrome" arises because of the interaction of two factors, under-estimation and imprecise measurement of progress. The reason why progress tends to be imprecisely measured, is because imprecise surrogates are used to measure it. "A surrogate is a substitute measure of some phenomenon that is used because it is not feasible to measure the phenomenon directly "(Anthony and Dearden 1980). In the case of software, consumption of resources is the (imprecise) surrogate often used to measure progress.

To rectify this situation, attempts have been made to develop more precise measurements that would directly measure progress in a software project e.g., automated monitoring systems such as SIMON (Fleischer and Spitter, 1976).

However, primarily because such tools only address one aspect of the problem i.e., the imprecise measurement of progress, but not the under-estimation aspect, their use could possibly result in unintended and dysfunctional consequences. Consider, for example, the situation of

introducing an effective measurement tool in a (typical) environment where projects tend to be grossly under-estimated at their initiation. The better the measurement tool, the earlier it will detect the fact that progress is not keeping up with the grossly under-estimated schedule. When such a discrepancy is detected early in the development cycle, management will, more often than not, react by adding more people rather than adjusting the schedule. This happens, according to DeMarco (1982), for 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 result of sticking with a wrong schedule that is too tight, is often an increase in the project's cost (Boehm, 1981), e.g., due to a large workforce level. Thus, what an application of an effective measurement tool will result in, in such an environment, are projects that are compressed in duration, and inflated in cost. Such an outcome might not necessarily be expected or welcomed (e.g., in an organization where smaller costs are more critical than shorter durations).

Such a scenario of unintended and dysfunctional

consequences of some managerial intervention, it should be noted, is not at all unique to this particular situation:

> The chain of effects in going from a problem to immediate consequences then to second-order-consequence (i.e., those that appear subsequent to, or as a result of, the immediate and obvious consequences of an action) and newly created problems is one of the pervasive characteristics of modern social systems. Quite literally, in such systems everything depends on everything else and often in ways so complex and round about that it is difficult to understand the interrelationships (Cleland and King, 1975).

And as a result,

> ... apparently logical solutions may prove faulty as their consequences ramify. Furthermore, since the consequences of a decision often occur much later than the decision itself, it is difficult for the members to trace backward from the disruptive consequences to determine precisely what caused them. The members cannot make such an analysis, simply because there are too many competing explanations. Thus, the only thing members can do when a new problem arises is to engage in more localized problem-solving (Weick, 1979).

The reader might recall that the above two quotations, were used in Chapter I within our argument for an integrative perspective to the study of software project management. Indeed, even though the issues we are raising here, on the possible dysfunctional consequences of measurement tools, are beyond the scope of our current model, we do feel that our general integrative approach does provide the viable basis for future extensions to address them.

V.5. The Economics of Quality Assurance:

> The development of software systems involves a series of production activities where the opportunities for interjection of human fallibilities are enormous. Errors may begin to occur at the very inception of the process where the objectives of the software system may be erroneously or imperfectly specified, as well as during the later design and development stages where these objectives are mechanized. The basic quality factor for software is that it performs its functions in the manner that was intended by its architects. In order to achieve this quality, the final product must contain a minimum of mistakes in implementing their intentions as well as being void of misconception about the intentions themselves. Because of human inability to perform with perfection, software development is accompanied by a quality assurance activity (Deutsch, 1979).

Quality Assurance (QA) is, thus, a set of activities "... performed in conjunction with (the development of) a software product to guarantee the product meets the specified standards. These activities reduce doubts and risks about the performance of the product in the target environment" (Pressman, 1982).

Software quality assurance is approached by two distinct and complementing methodologies. The first is that of assuring that the quality is initially built into the product. This involves emphasis on the early generation of a coherent, complete, unambiguous, and nonconflicting set of requirements. Then as the product is designed and coded, review and testing of the product, the second quality tool, are encountered (Deutsch, 1979).

As was indicated in Section III.3. (on "Model Boundary") the model's development phase includes both the design and coding activities, but excludes the development of the requirements. It was also indicated then, that we will be assuming that software design commences (within the model's boundary) at the "successful completion" of a software requirements review (outside the model's boundary), and that there would be no subsequent changes or modifications in the system's requirements. As a result, the analysis of this section on the economics of QA only applies to the second QA tool above, namely, the review and testing of the product.

Several specific techniques are available for reviewing and testing the software product as it is designed and coded. These include, structured walkthroughs and technical reviews (Freedman and Weinberg, 1982), inspections (Fagan, 1976), code reading (a process where code logic and code format is scrutinized by a programmer other than the original designer) (Weinberg, 1971) and integration testing (Daly, 1977), (Jones, 1982). Not included in this activity is module or unit testing, which is commonly considered to be part of the coding process (McKeen, 1979).

In this section we will focus, not on the technical aspects of QA, but rather on the economics of the QA activity. We will investigate the tradeoff between the

benefits and costs of the QA effort in terms of the total project cost.

The utilization of QA tools and techniques adds cost to the development of software. For example, man-hours are expended in developing test cases, running test cases, conducting structured walkthroughs,... etc. This added cost is,

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

This "pressing concern" has not, however, been addressed in the literature. That is, as of yet, there are no published studies investigating "how cost efficient are the QA operations during the development cycle." We can propose three possible reasons for this deficiency in the field's research repertoire: (1) It is a managerial issue. Like many other aspects of software production, managerial considerations tend to attract less research attention. "Perhaps this is so because computer scientists believe that management per se is not their business" (Cooper, 1978). (2) "Software Quality assurance has only recently i.e., within the last four or five years, gained a place of formal status

and recognition within engineering hierarchies" (Stringer, 1979). The emphasis, until now, has been on "selling" this "young" concept to practicing managers ... hence the emphasis on stressing (only) the benefits (e.g., see (Ergott, 1979) and (Cooper and Fisher, 1979).) (3) The high cost of controlled experimentation in software engineering (Myers, 1978), (Glass, 1982).

In the remaining part of this section we will use our model to investigate, not whether QA is justified, but how much QA is justified. To do this, we simulated project EXAMPLE under different levels of manpower commitments to the QA function and observed the benefits and costs in each case.

The primary goal of QA is "that errors be detected and corrected as early as possible and only a minimal amount of problems be allowed to slip from one phase of the development to the next" (Tsui and Priven, 1976). Several studies have established the significant cost savings gained by the early detection and correction of errors. For example, in a study by Shooman reported in McClure (1981), it was determined that detecting and correcting a design error during the design phase (i.e., through the QA activities) is one-tenth the effort that would be needed to detect and correct it later during the system testing phase because of the additional inventory of specifications, code, user and maintenance manuals, ... etc., that would require correction in the

later case.

An important relationship to investigate is, therefore, the one between the QA effort expended and the % of errors detected during development. Such a relationship was obtained from our experiment, and is exhibited in Figure V.21.

The significant feature of the relationship is the "diminishing returns" of QA exhibited as QA expenditures extend beyond 20-30% of development effort. This type of behavior is supported by two types of results in the literature. First, Shooman (1983) observed that "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." The second result, reported by Boehm (1981) and shown in Figure V.22., is a compilation of a number of studies that provide single points on error-removal functions.

What the results of Figure V.21. suggest is that the savings in the cost of processing errors that result from the application of QA, flattens out as QA expenditures extend beyond 20-30% of development effort. This result is shown in Figure V.23. As can be seen, the combined costs of rework (i.e., correcting errors during development) and testing flatten out as QA expenditures extend beyond 20%. On the

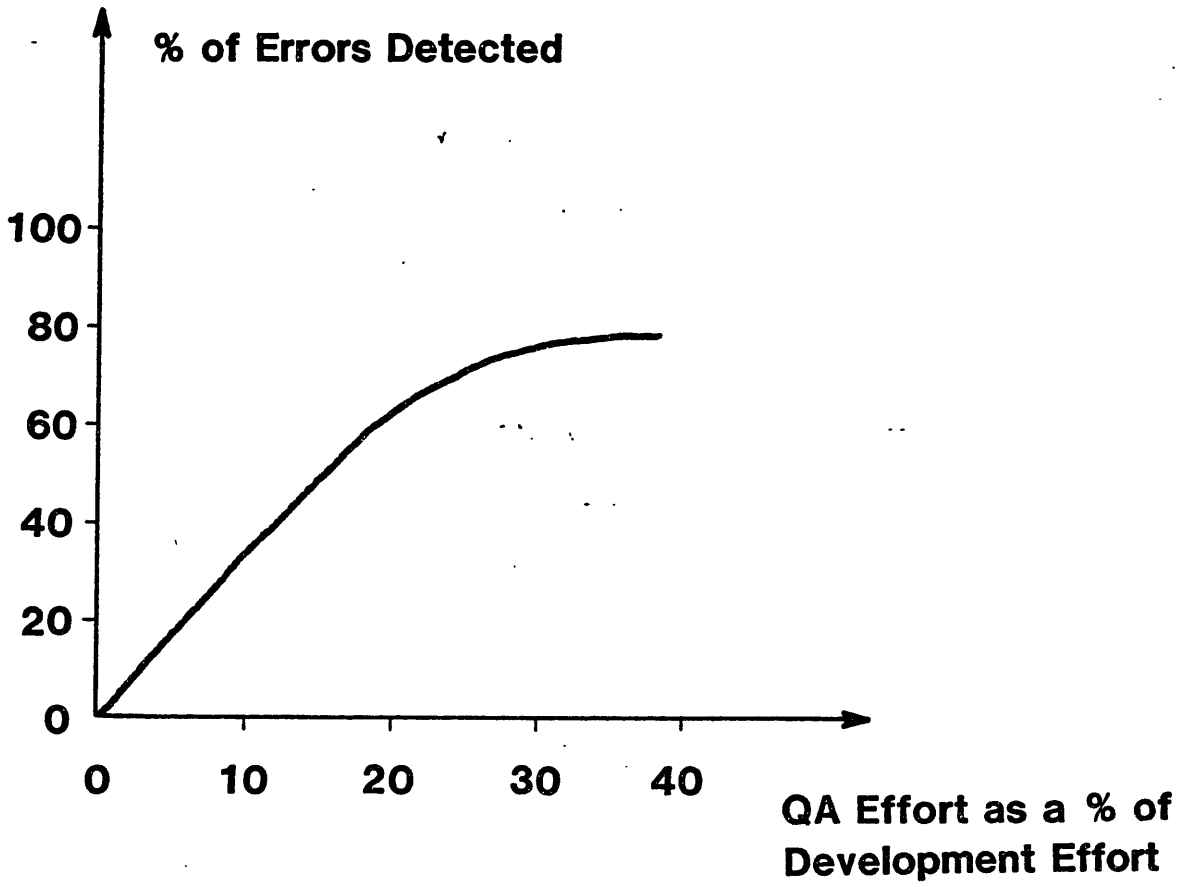

% of Errors Detected
100
80
60
40
20
0
0
10
20
30
40
QA Effort as a % of
Development Effort

Figure V.21

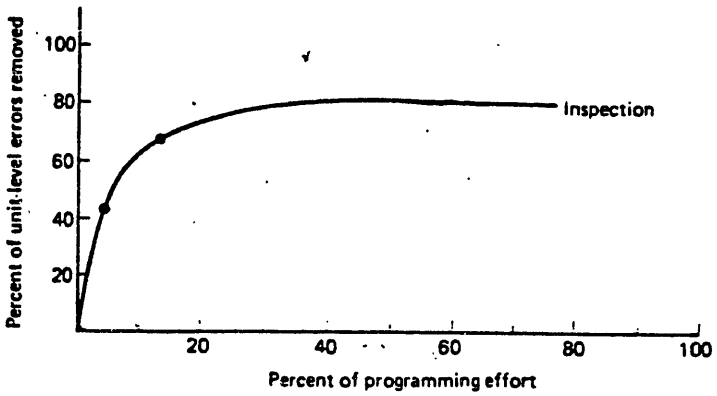

Percent of unit-level errors removed
100
80
60
40
20
Inspection
20
40
60
80
100
Percent of programming effort

Figure V.22

other hand, notice that increasing QA as a % of the development effort results in an exponential increase in QA's absolute cost (in man-days). The reason why this happens is that as a larger fraction of the development effort is allocated to QA, the development effort itself increases. And the reason why this in turn happens is that as more man-days are allocated to QA (without corresponding reductions in Rework + Testing man-days), the project's total size in man-days goes up. This in turn leads to the acquisition of a larger workforce. A large workforce, in turn, means a less productive workforce (e.g., due to training and communication overheads) which, as a result, drives the project's development man-days effort higher.

The final, and perhaps most useful, question to address concerns the "optimal" QA effort expenditure. For project EXAMPLE, the answer is shown in Figure V.24., which plots EXAMPLE's total cost (in man-days) against QA effort defined in terms of % of development man-days. As can be seen, the "optimal" QA effort expenditure is 16% of the development man-days.

Two important conclusions can be drawn from Figure V.24. The first, more generalizable conclusion, is that QA policy does have a significant impact on total project cost. As can be seen from the figure, project EXAMPLE's cost ranges from a low of 3,770 man-days, to values in the range of 5,000

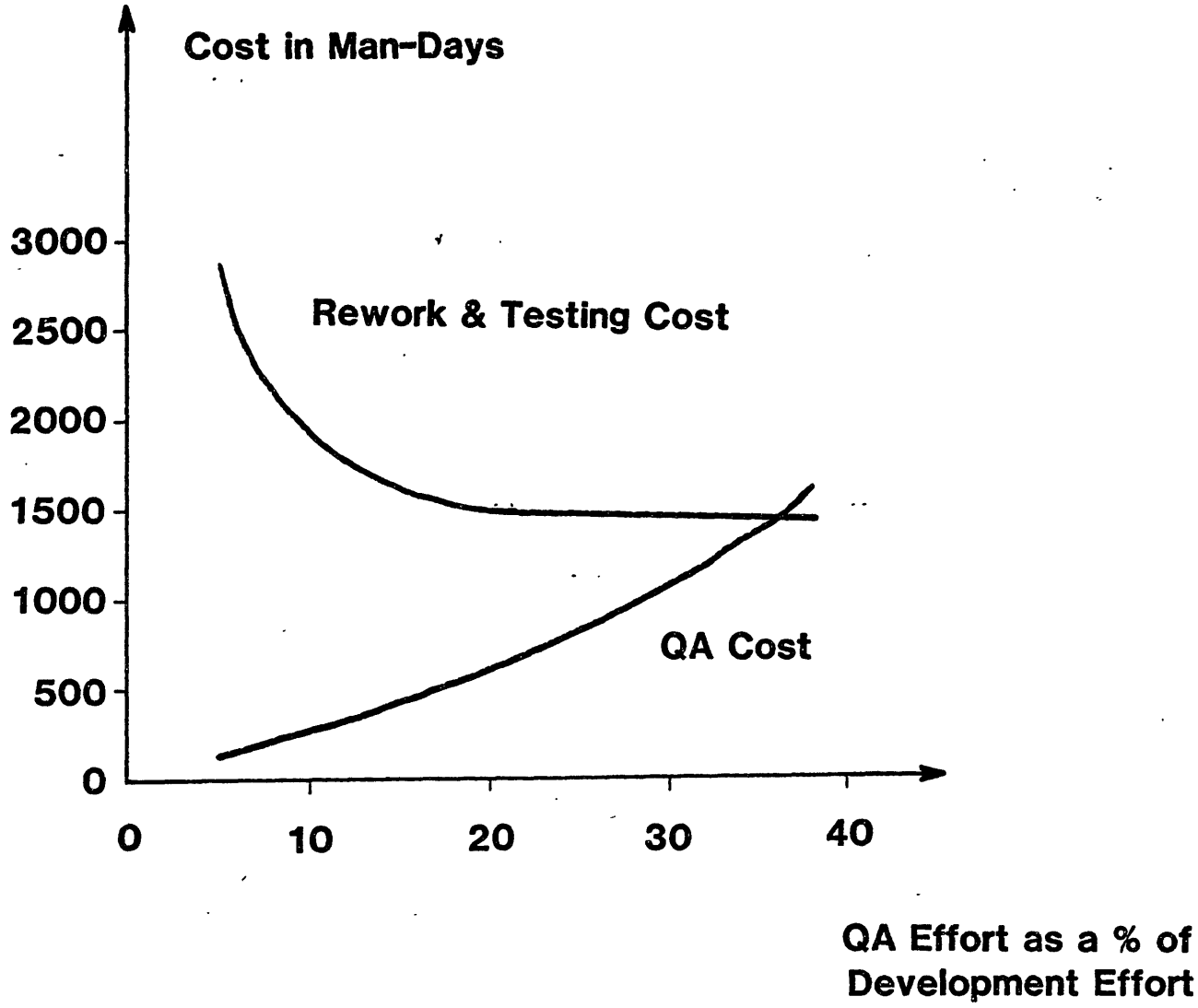

Cost in Man-Days
3000
2500
2000
1500
1000
500
0
Rework & Testing Cost
QA Cost
0
10
20
30
40
QA Effort as a % of
Development Effort

Figure V.23

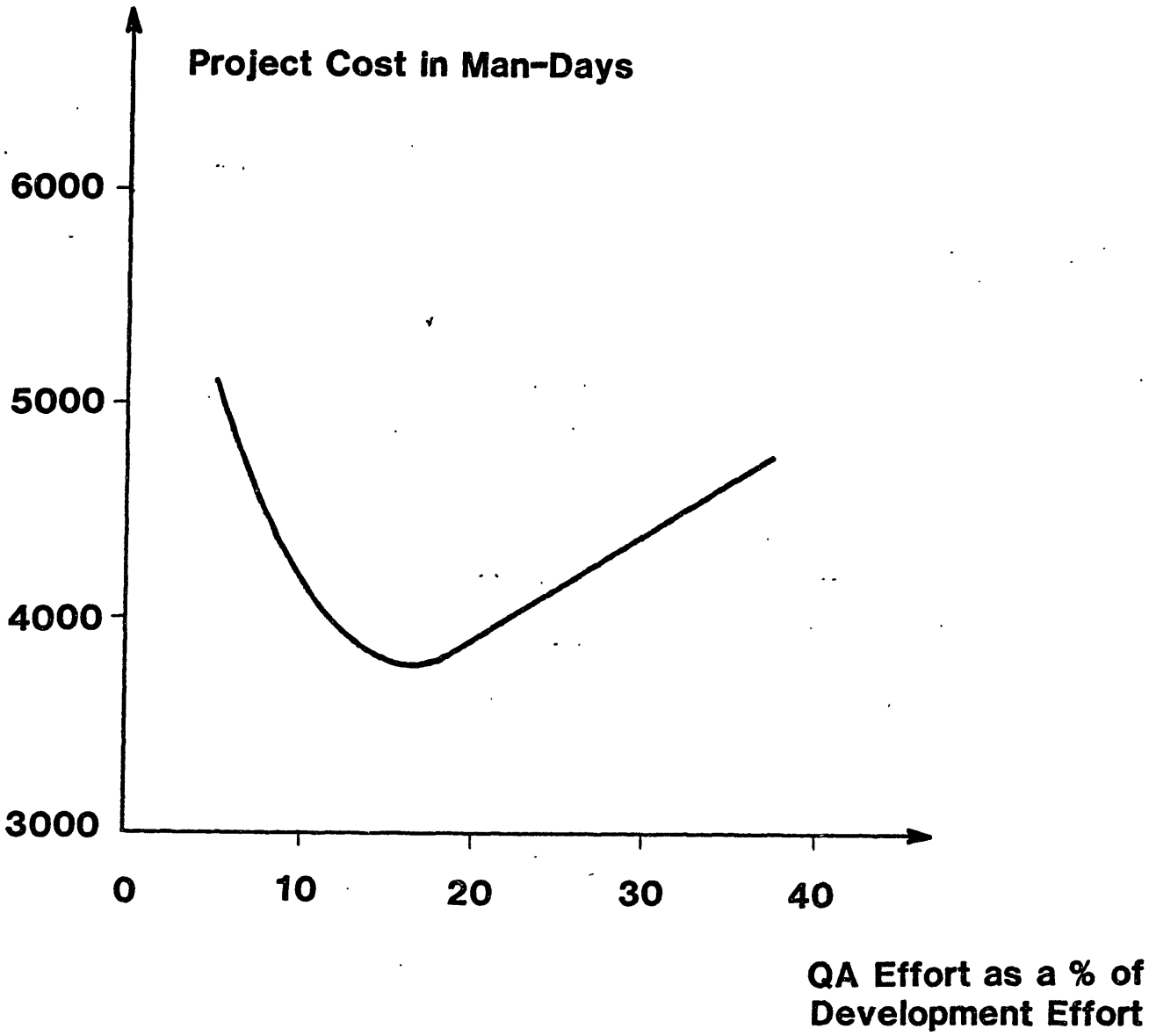

Project Cost in Man-Days
6000
5000
4000
3000
0
10
20
30
40
QA Effort as a % of
Development Effort

Figure V.24

man-days i.e., values that are 33% higher. At low values of QA expenditures this increase in cost results from the large cost of the testing phase. On the other hand, at high values of QA expenditures, the excessive QA expenditures are themselves the culprit. The second result is, of course, deriving the optimal QA expenditure level of 16%. What, in our opinion, is really significant about this result is not its value, since this cannot be generalized beyond an EXAMPLE-type software project, but rather the process of deriving it, namely, our integrative system dynamics approach. Beyond controlled experimentation (which are too costly and time consuming to be practically feasible), as far as we know, this model provides the first capability to quantitatively analyze the costs/benefits of QA policy for software production. And this, it is encouraging to note, is generalizable, in the sense that one can customize models for different software development environments to derive environment-specific optimality conditions.

But why is the optimal value of 16% derived above not generalizable? To address this question we will test its sensitivity to two project variables, which can change from project to project and/or from organization to organiztion. Such an investigation will have two useful outcomes: First, we will derive results of the form "An increase in factor (X) warrants a greater QA expenditure," which will be generalizable beyond our specific project EXAMPLE

environment. Such results could, for example, be useful "rules-of-thumb" for organizations to use when adapting published results or results from other organizations to their own environments. Secondly, such "rules-of-thumb" can, in the same way, be applied to adapt and adjust our own results above, thus increasing their generalizability.

The first project variable we consider concerns the distribution of effort among the project phases. In planning a software project, management does not only estimate the project's total effort in man-days, it in addition allocates that effort among the project's phases (Gunther. 1978). As was explained in detail in Section V.3.2., substantial differences in opinion exist on how this effort distribution is or should be made (McKeen, 1983). In project EXAMPLE's base-case, we assumed a distribution of 80% for development (i.e., design and coding) and 20% for testing. As was explained in Section V.2., these values were chosen to conform to the TRW software development environment. In this experiment, we will examine the effect of another distribution, namely, the 40-20-40 effort distribution profile i.e., 40% for preliminary and detailed design, 20% for coding, and 40% for testing. Which, as was mentioned before, is perhaps the most widely touted rule-of-thumb for the distribution of effort among the phases of software development projects (McKeen, 1981), (Bruce and Pederson, 1982), (Oliver, 1982), (Jensen and Tonies, 1979). [As was

explained earlier, this effort profile is translated in our model into 60% for development (i.e., design and coding) and 40% for system testing.]

Before we present the experiment's results, there is an important comment to make. Notice that we are examining the affects of how much effort is allocated to the testing phase on how much effort should be allocated to QA! It appears as though we have confused what the independent and dependent variables are. After all, QA is utilized not only earlier in the development cycle, but also for the explicit purpose of affecting the testing phase (i.e., minimizing its cost). Our experiment's (seemingly) lopsided set-up is, however, really a reflection of what the state-of-the-practice is in software project management. Both in the literature (e.g., (Boehm, 1981)) as well as in the organizations we interviewed (e.g., based on discussions with (McGowan, 3), (O'Conner, 10), (Landolfi, 11), (Sheldon, 12), and (Hisamune, 15)) the sequence of steps followed in allocating the planned man-day expenditures are as a follows: First, the total project's effort is estimated. Then, the effort is distributed among the life-cycle phases (e.g., using the 40-20-40 rule). And then effort is allocated to QA as % of the development effort. For example, in Boehm's Software Engineering Economics, he uses a case study titled "The Hunt National Bank EFT System" to outline how COCOMO would be used to estimate and allocate a project's man-day expenditures. The

following sequence of steps is followed:

1. COCOMO's effort and schedule equations are used to estimate the project's man-days, and development time.
2. Next, using guidelines for the distribution of effort among the project's life cycle phases, man-days are allocated to development (i.e., design and coding) and testing.
3. Finally, effort is allocated to QA activities using some guidelines expressing QA as a % of development man-days.

A final note. This lopsided approach to planning a software project is probably a result of how the (young) software engineering field has grown. First, there was no explicit development life cycle with the emphasis almost totally placed on the programming phase of a project. Next, we realized the value of breaking the development process into distinct life cycle phases, and emphasizing its earlier requirements and design phases. And, only recently have we also come to realize the importance of emphasizing quality during the development of a software project. However, what the above lopsided planning sequence suggests is that the field has not yet grown to full maturity.

Running project EXAMPLE with the new effort distribution profile i.e., where 40% of the man-days are allocated to

testing rather than the base-case's 20%, produced the result shown in Figure V.25. (for experiment #1). That is, the optimal QA expenditure level drops to 11% of development effort.

The fundamental reason for this is that effort expenditures are not only a function of the actual workload, but they are also a function of planned expenditures. This phenomenon was explained in detail in Section V.3. Thus, by allocating more to the testing activity, the testing effort will expand even though the workload itself might not. What our experiment's results is therefore suggesting, is that we "accomodate" this phenomenon of organizational behavior (rather than fight it). In other words, since the testing effort will expand anyhow (as a result of management's increased allocation to testing), it makes sense to also increase the workload itself and, in a sense, reap the most return from the increased investment in testing. And this, of-course, would be achieved by decreasing the investment in QA. (Note: the 11% allocation to QA is still within the range of QA expenditures reported both in the literature and in the organizations we studied. See Chapter III.)

The second project variable we will consider concerns software development productivity. Recall that in our formulation of productivity we made a clear distinction between two sets of factors that can affect how productive

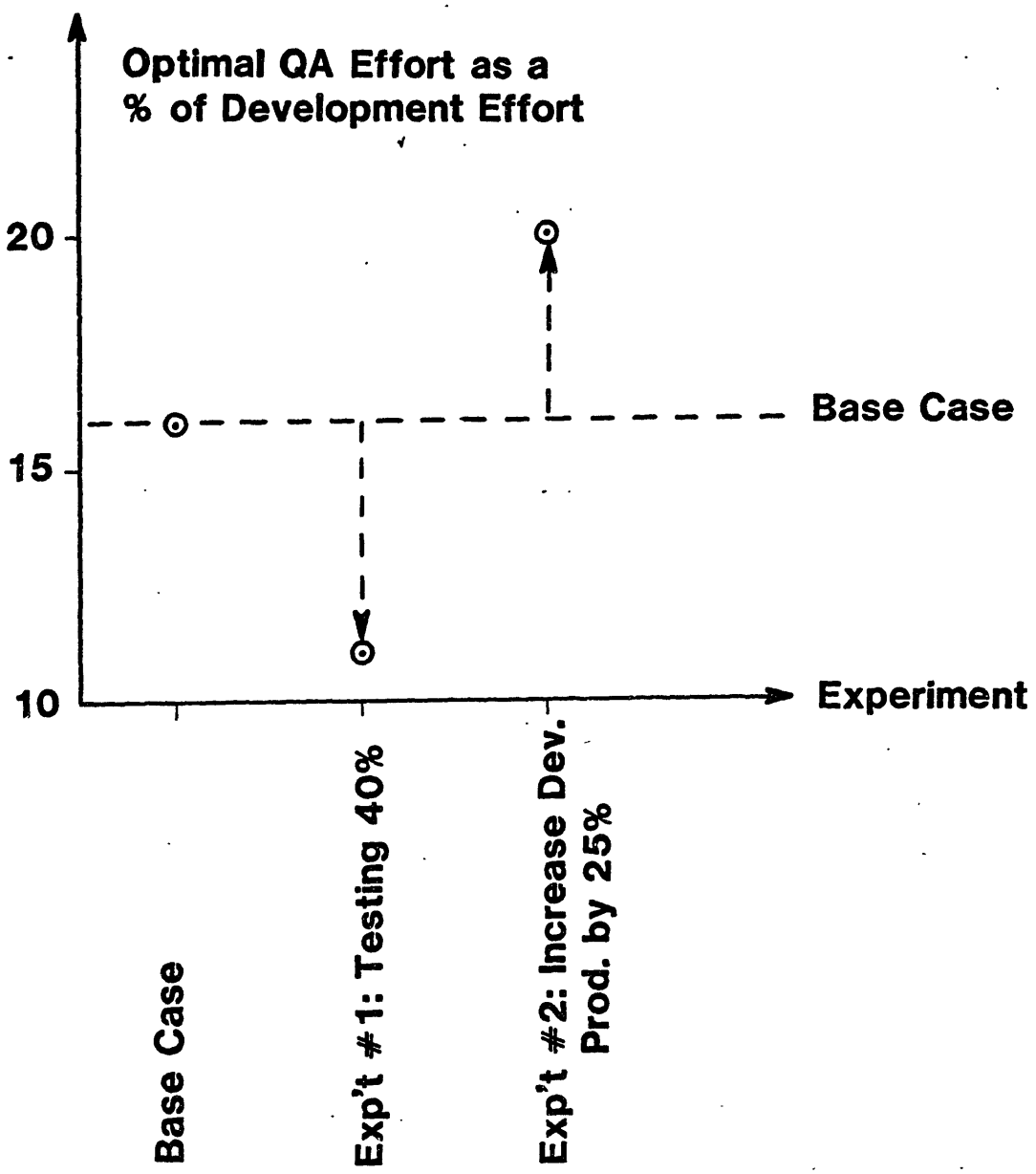

Optimal QA Effort as a
% of Development Effort
20
15
10
Base Case
Experiment
Base Case
Expt #1: Testing 40%
Expt #2: Increase Dev.
Prod. by 25%

Figure V.25

people will be on a software project. The first set included those factors that affect productivity dynamically throughout the development of a single project. These, included: workforce experience, learning, motivation, and communication. The second set included environmental factors which tend to remain invariant during the life of a single project. This set included factors such as: availability of software tools, computer-hardware characteristics, programming language, product complexity, ... etc. Because this second set of factors does not play any dynamic role during the life of a single project, we were able to capture them through a single parameter, namely, the project's "Nominal Potential Productivity." What we would like to investigate here is the following: The effect on the optimal QA expenditure of changes in potential productivity (i.e., due to changes in the software development environment).

In Section V.2., we set project EXAMPLE's "Nominal Potential Productivity" to 60 DSI/man-day. (Actually, it was set to 1 Task/man-day, where a task was then defined to be 60 DSI.) This was done to conform to the TRW software development environment. In this, Experiment #2, we examine the effect of increasing the value of "Nominal Potential Productivity" by 25% i.e., to become 75 DSI/man-day. Notice that such an increase only affects the productivity of software development. Such an increase has no direct effect on the productivity of processing errors (i.e., detecting

them and correcting them). Of course, one could argue that there is some correlation between the two productivities e.g., higher quality people would be both more productive in producing code and in detecting and correcting errors. And that we, therefore, need to make corresponding adjustments to the error processing productivities in the model. While perfectly feasible to do, such adjustments would, however, defeat the purpose of this experiment, and which we can now elaborate in more precise terms: We would like to examine the effects of increasing the differential between development productivity and error-processing productivity in an organization.

The results of the experiment are shown in Figure V.25. That is, an increase in development productivity warrants an increase in QA expenditures relative to development expenditures. Higher development productivities mean that each man-day expended on the development of software will yield more software. As a result more QA effort would be required to handle this increased output. It is important to note here this increased output will not, in and of itself, trigger adjustments in the amount of QA expended. And that the required increases in QA must, therefore, be explicitly planned for. The reason for this has to do with the "Parkinsonian" execution of the QA activity. As was discussed in Chapter III, both our own findings as well as findings reported in the literature, suggest that the QA rate

is often independent of the QA effort allocated. What usually happens is that the QA effort is planned and allocated, usually in the form of a fixed schedule of periodic group-type functions (Mitchell, 1980). For example, a 2-hour walkthrough for the 5 members of team (A) is scheduled for every Friday. During these periodic "QA Windows," all tasks developed since the previous one are supposed to be processed. And what we were surprised to find was that, in an almost perfect realization of Parkinson's Law, irrespective of how many tasks need to to be processed within the specified "QA Window" they almost always do. No backlogs, therefore, develop in the QA pipeline. Even when QA activities are relaxed or suspended because of schedule pressure, no backlogs develop. That is, when walkthroughs are suspended for a while on a project, the requirement for a "walkthrough" is also suspended, not postponed (Hart, 1982).

We can propose an explanation for how and why this happens. Since the objective of the QA activity is to detect invisible errors, invisible that is until they are detected, it becomes almost impossible to tell whether the QA job was completely done (i.e., all those invisible errors were in fact detected). By the same token, it is as difficult to tell that the job has not been completely done (except much later in the life cycle). Under such circumstances it becomes quite easy to rationalize both to oneself and to management that the QA job that was possible to do, was not

insufficient. Furthermore, the QA effort that is possible to expend (i.e., in terms of available time and effort), is usually what is expended and not more (e.g., even if called for due to a larger than expected workload of developed tasks) because there seems to be no significant incentives to do otherwise. Firstly, at the psychological level, there are actually disincentives for working harder at QA, since it only "exposes" more of one's mistakes (Weinberg, 1971). And secondly, at the organizational level there are seldom any award mechanisms in place that promote quality or quality-related activities (Cooper and Fisher, 1979).

V.6. Staffing: Brook's Law Revisited

Our objective in this section is, in some sense, the reverse of that of the previous section. In Section V.5. Our aim was to generate new results that are generalizable. In this section, on the other hand, we will be questioning the generalizability of an old "result," namely, "Brooks' Law."

Brooks' Law was first publicized in Dr. Fred Brooks' 1975 book titled The Mythical Man-Month: Essays on Software Engineering. The book embodies a number of insights into the management of large software projects gained through Brooks' experience in managing the development of IBM's OS/360. Paraphrasing Brooks (1978):

> After leaving IBM in 1965 to come to Chapel Hill as originally agreed when I took over OS/360, I began to analyze the OS/360 experience to see what management and technical lessons were to be learned ...
> My own conclusions are embodied in the essays that follow, which are intended for professional programmers, professional managers, and especially professional managers of programmers.

Brook's Law is stated as follows: "Adding manpower to a late software project makes it later" (Brooks, 1978).

The lack of interchangeability between men and months was recognized by Brooks as being caused by two factors, training and intercommunication overheads:

> Each worker must be trained in the technology, the goals of the effort, the overall strategy, and the plan of work. This training cannot be partitioned, so this part of the added effort varies linearly with the number of workers.
> Intercommunication is worse. If each part of the task must be separately coordinated with each other, the effort increases as $n(n-1)/2$. Three workers require three times as much pairwise intercommunication as two; four require six times as much as two ...
> Since software construction is inherently a systems effort ... an exercise in complex interrelationships ... communication effort is great ... Adding more men then lengthens, not shortens, the schedule (Brooks, 1978).

Since its "enactment," Brooks' Law has been widely endorsed in the literature (e.g., see (Synnott and Gruber, 1981), (Paretta and Clark, 1976), (Pressman, 1982), (Jensen and Tonies, 1979), and (Boehm, 1981).) Furthermore, it has often been endorsed indiscriminately i.e., for not only

large, but also small projects, and not only systems programming type projects, but also applications software systems. This, even though Brooks was quite explicit in specifying the domain of applicability of his insights, including his Brooks' Law, i.e., to what he calls "Jumbo" systems programming projects. For example, Pressman (1982) extends Brooks' Law to 6-10 man-year projects, while in (Jensen and Tonies, 1979) and (Synnott and Gruber, 1981) it is extended to the domain of applications software systems.

Interestingly, this wide-spread endorsement of Brooks' Law has taken place, even though the "law" has not been formally verified. Our objective in this section is to do just that. Specifically we will investigate whether Brooks' Law does apply to the environment of "medium-sized applications projects developed in a familiar, in-house development environment," i.e., to our prototype project EXAMPLE.

As we have seen in Section V.2., project EXAMPLE's size is (as are many such software projects) initially under-estimated. As a result the project experiences scheduling problems, and does in fact overshoot its original schedule. (The reader is advised to refer to the detailed description furnished in Section V.2.) We also saw that when the project's scheduling problems surface management first reacts by adjusting the project's workforce level i.e.,

adding more people. However, as the project proceeds towards its final stages, with its scheduling problems still persisting, management becomes increasingly reluctant, because of workforce stability considerations, to add more people, and as a result reacts instead by adjusting the project's schedule.

Management's policy on how to balance workforce and schedule adjustments is captured in the model through the formulation of the variable "Willingness to Change Workforce." Through adjusting this variable we can, therefore, examine the impact of more aggressive manpower acquisition policies on the project's cost and duration. That is, examine whether a policy (A) in which management continues adding more people to project EXAMPLE even as the project proceeds towards the end of its system testing phase, results in a larger schedule overshoot than does a policy (B) in which management refrains from adding more people much earlier e.g., towards the end of the development phase. Brooks' Law suggests that policy (A) would produce a longer project duration.

In the base case (and based on discussions with (Lombardi, 23), (Garett, 24) and (Nichols, 25)), the "Willingness to Change Workforce" is formulated in terms of a time parameter that is the sum of the "Hiring Delay" and the "Assimilation Delay." Specifically, in the early stages of

the project when "Time Remaining" would generally be much larger than the sum of the "Hiring Delay and the "Assimilation Delay" management would be willing to adjust the workforce level to meet the project's scheduled completion date. As the number of days perceived remaining drops below 1.5 * (Hiring Delay + Assimilation Delay), though, management starts becoming reluctant, and increasingly so, to increase workforce level. In the base case the values of the "Hiring Delay" and the "Assimilation Delay" are 40 and 80 working days, respectively. Thus, as "Time Remaining" drops below 180 days, management, in the base case, starts becoming reluctant to hire new people, even though the time and effort perceived remaining might imply that more people are needed. The reluctance stems from the realization that most of those remaining 180 days, would be "wasted" in the hiring process and then in acquainting the new people with the mechanics of the project, in integrating them into the project team, and in training them in the necessary technical areas. And when the "Time Remaining" drops below 0.3 * (Hiring Delay + Assimilation Delay) i.e., below 48 working days, no more additions would be made to the project's workforce i.e., the hiring rate falls to zero.

It should now be clear how we can model more aggressive manpower acquisition policies through the "Willingness to Change Workforce" formulation. We can do that simply by decreasing the value of the time parameter. For example, if

we set the time parameter to 30 working days (instead of its base-case value of 40 + 80 = 120) we would be modeling a situation where management's willigness to add to the workforce continues until much later into the project. In the base case, management starts becoming reluctant to increase the workforce level when the perceived number of days remaining to complete the project drops below 180 days, and stops hiring completely when it drops below 48 working days. Under the current more aggressive policy, management starts becoming reluctant at 45 days and stops manpower additions completely at 9 working days, or two weeks, before the perceived completion date.

Thus, by adjusting the value of the time parameter we are able to examine the scheduling consequences of a number of manpower acquisition policies, ranging from the base-case policy to the above (somewhat extreme) policy. The results are depicted in Figure V.26.

As can be seen from the figure the results do not support Brooks Law. What our results show is that adding more people to a late project causes it to become more costly, but not to complete later. The increase in the cost of the project is caused by the increased training and communication overheads, and which in effect decrease the productivity of the average team member, and thus increase the project's man-day requirements. For the project's

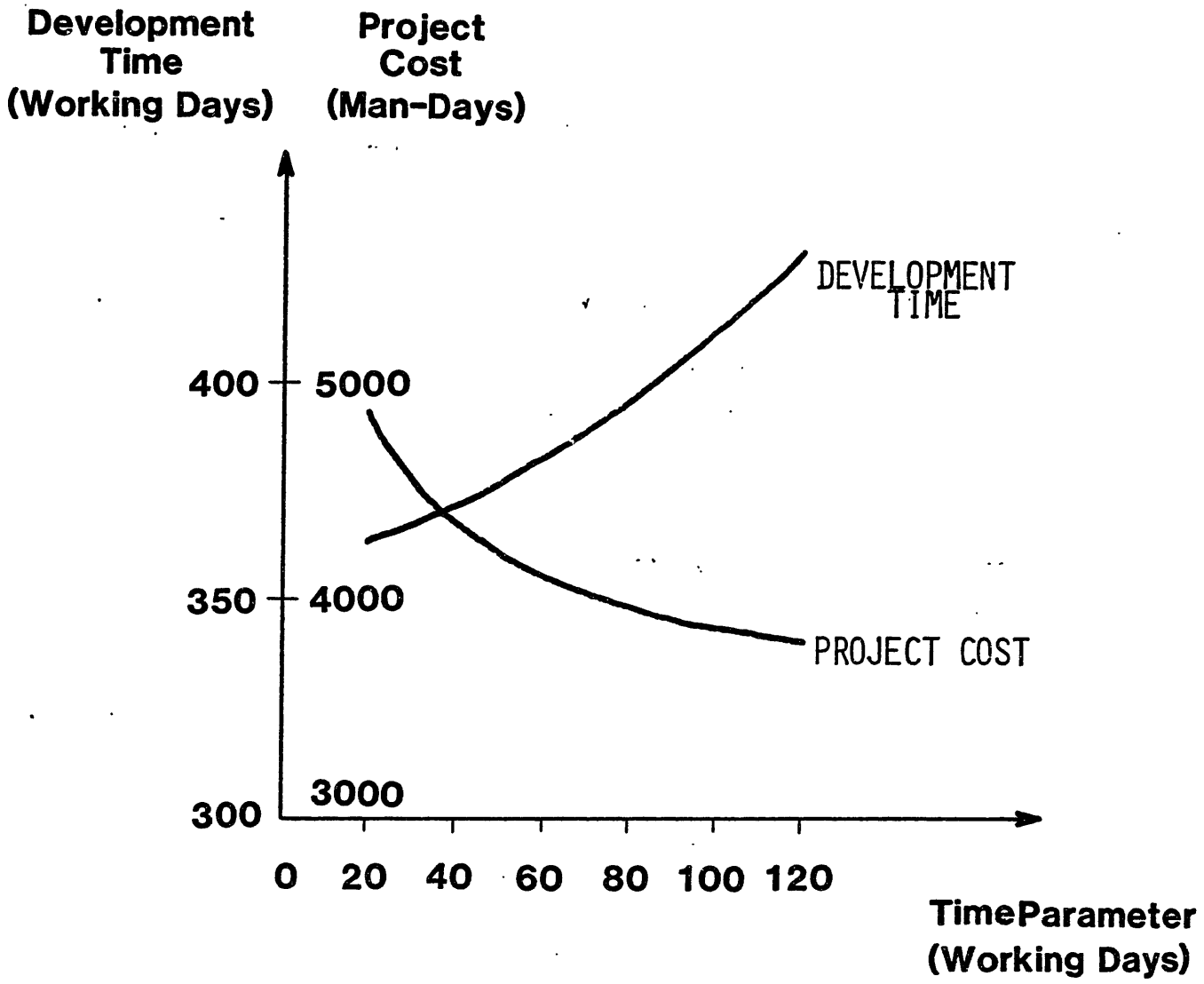

Development Time (Working Days)
Project Cost (Man-Days)
DEVELOPMENT TIME
PROJECT COST
400
350
300
5000
4000
3000
0
20
40
60
80
100
120
TimeParameter (Working Days)

Figure V.26

schedule to also suffer, the drop in productivity must be large enough to render an additional person's contribution to the project to be, in effect, a negative contribution. Our results indicate this is not the case in project EXAMPLE.

The conclusion that we can draw from our experiment's results is that Brooks' Law does not universally apply to all software development environments. And that, in particular, it does not seem to apply to the EXAMPLE-type project environment i.e., the medium sized application project developed in a familiar, in-house development environment. It is, therefore, not necessarily an invalidation of Brooks' "Brooks's Law," but rather a disqualification of the notion (implied not by Brooks but by the writings of others in the literature) that "Brooks' Law" is a universal law of software development.

The question, however, still remains: under what conditions would Brooks' Law apply? While the complete answer to this question lies beyond the scope of this research, we are, however, able to present some preliminary results. One of the advantages of simulation modeling is the flexibility it provides in experimenting on the modeled system under perturbed conditions. The results of one such experiment i.e., on a "perturbed EXAMPLE" project, is shown in Figure V.27.

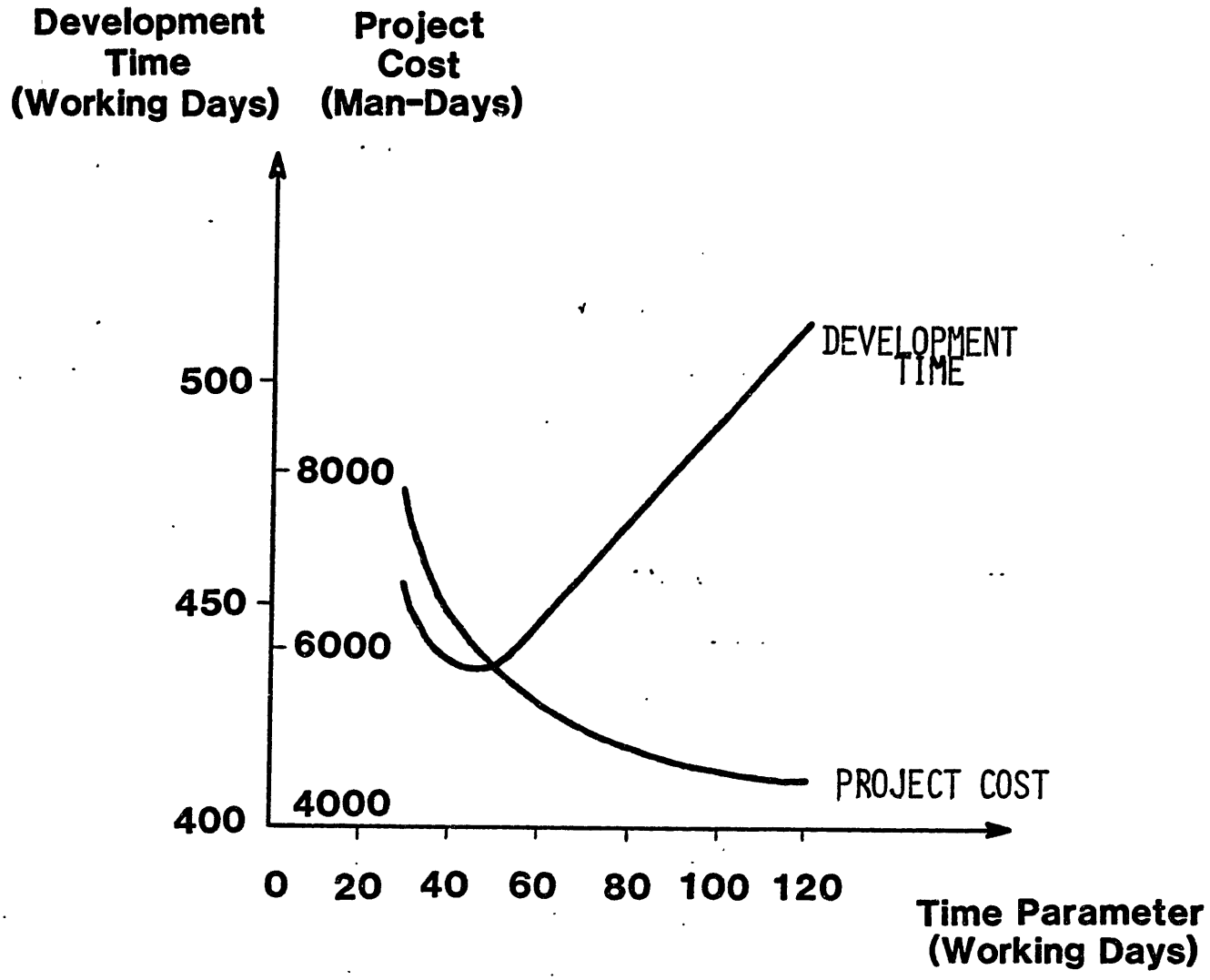

Development Time (Working Days)
Project Cost (Man-Days)
500
450
400
8000
6000
4000
DEVELOPMENT TIME
PROJECT COST
0
20
40
60
80
100
120
Time Parameter (Working Days)

Figure V.27

In this experiment, we quadrupled the training overhead for project EXAMPLE. In the base case, a new hiree consumes in training overhead, on the average, the equivalent of 20% of an experienced full-time employee's time for the duration of a training period that extends for 4 months. In this current experiment, a new hire consumes 40% of an experienced full-time employee's time for a training duration that extends for 8 months. Such an increase in the training overhead, while admittedly somewhat extreme, is never-the-less the kind of perturbation that we would need to make if we were to model the software development environment of large and complex systems programming software (Corbato and Clingen, 1980), e.g., such as the IBM OS/360.

Notice that, even with such a large training overhead, Brooks' Law does not always hold. It only holds, in this experiment, when the Time Parameter is less than 50 working days. As was explained earlier, a smaller Time Parameter means that management's willingness to add more people to the project is maintained until later in the project's life cycle. Specifically, when the Time Parameter is set to 50 working days, management would be willing to add more people up until the point in time when it is perceived that the time remaining to complete the project is less than 0.3 * 50 = 15 working days i.e., 3 weeks. That is, until the final stages of the testing phase. It is at such extremely aggressive manpower acquisition policies that Brooks' Law

holds for our "perturbed EXAMPLE" project.

There are several conclusions that we can draw from this analysis:

* Adding more people to a late project does not necessarily make it later.
* In particular, Brooks' Law does not seem to apply to the EXAMPLE-type software project environment i.e., the medium sized application project developed in a familiar, in-house development environment.
* In such an environment, adding more people to late project does, however, make it more costly.
* But even in a particular software development environment, our results indicate that adding more people to a late project may or may not make the project later. It depends on where in the project's life cycle the people are added.

V.7. Summary:

In this chapter we used our integrative system dynamics model of software project management as an experimentation vehicle to study/predict the dynamic implications of an array of managerial actions, policies, and procedures pertaining to the development of software. Four areas were studied: (1) Scheduling;

(2) Controlling; (3) Quality Assurance; and (4) Staffing.

Three experiments were conducted in the software scheduling area. We examined the impact that schedules have on project performance in the first experiment, the portability of the quantitative software estimation tools in the second, and in the third experiment we investigated the long-term impact of the "estimation by analogy method."

In the area of project control, we examined the "90 % Syndrome" phenomenon, and provided an analysis of its causes, namely, the lack of visibility and underestimation.

The third area of investigation concerned the economics of software quality assurance. Two sets of experiments were conducted in this area. The objective of the first set was to investigate, not whether QA was justified, but how much QA was justified. In the second set of experiments, we examined the sensitivity of the derived "optimal" QA expenditure level, to two project variables, namely, the project's planned effort distribution profile, and the software development productivity.

Finally, in the area of project staffing, we tested the applicability of Brooks' Law to our prototype project environment (i.e., to the domain of medium-sized applications projects developed in a familiar, in-house development environment).

VI. CONCLUSIONS AND SUGGESTIONS FOR FUTURE RESEARCH

The purpose for a concluding chapter is to provide the opportunity for the researcher to look back in order to assess what has been accomplished, and at the same time, to furnish an occasion for the researcher to look ahead in order to suggest future avenues for prospective research. These activities, while of different orientation, are closely interrelated; any statement of what has been done invites inquiry as to what remains to be done. This chapter, coming at the culmination of the research, provides the vantage point from which the researcher can fulfill these express purposes. The following sections of the chapter entitled "Summary of Results" and "Suggestions for Future Research" provide the "look back" and "look ahead," respectively.

VI.1. Summary of Results:

The objective of this research effort is to enhance our

understanding of, and gain insight into, the general process by which software development is managed. To achieve this objective we accomplished the following three tasks:

1. Developed an integrative system dynamics model of software development project management.
2. Conducted a case study to test the model.
3. Used the model as an experimentation vehicle to study/predict the dynamic implications of an array of managerial policies and procedures.

In the remaining part of this section, we will elaborate further on the above three research accomplishments.

Model Development:

The development of the integrative system dynamics model of software development project management constitutes the following set of accomplishments:

1. The model integrates our knowledge of the micro components of software development project management (e.g., programming, prductivity, planning, controlling, ...etc.) into an integrated continuous view of the software development process.

A major defect in much of the research to date has been its inability to integrate our knowledge of such micro

components for deriving implications about the behavior of the organization in which the micro components are embedded (Thayer, 1979). Paraphrasing Jensen and Tonies (1979):

> There is much attention on individual phases and functions of the software development sequence, but little on the whole life cycle as an integral, continuous process --- a process that can and should be optimized.

Clearly, this "micro-oriented" type of work is a useful beginning in helping us obtain a better understanding of the software development activity. However, before we can say that we have a <u>complete</u> understanding of any such activity, "... it is necessary to show that our knowledge of the individual components can be put together in a total system, i.e., an organization can be synthesized, which allows for the interactions of all the relevant variables and all the structural components" (Cohen and Cyert, 1965).

In addition to the benefit of helping us achieve overall understanding, an integrative perspective is useful in two more "tactical" ways: problem diagnoses and solution evaluation. The interactions and interdependicies which characterize our management systems, will similarly characterize the problems that beset such systems (Cleland and King, 1975). In Brooks' words: "... no one thing seems to cause the difficulty (in software projects) ... But the accumulation of

simultaneous and interacting factors..." (Brooks, 1978). An integrative perspective is, therefore, useful because it both "prompts" as well as "facilitates" the search for the multiple, and potentially diffused, set of factors that are interacting to cause software development problems. An example of this is the schedule overshoot problem, which, as was shown in Chapter V, can arise, not only because of schedule underestimation, but also as a result of management's hiring/firing policies.

Again, because of the interactions and interdependencies that characterize management systems, managerial intervention (e.g., to solve a perceived problem) often leads to second- and third-order consequences and newly created problems (Weick, 1979). By providing us with a comprehensive world view, the model is a useful tool to fully assess such second- and third-order consequences. An example of this has been our analysis of the "Safety Factor Policy" in scheduling software projects. It was shown that while such a policy "succeeds" in producing more accurate project estimates, the intended consequence of the policy, it also tended to "create" more costly projects, which is both an unintended and a dysfunctional consequence.

2. The model identifies feedback mechanisms, and uses

them to structure and clarify relationships in software project management. While the significance and applicability of the feedback systems concept to the study of managerial systems has been substantiated in a large number of studies outside software engineering, it still remains largely foreign to the software engineering project management community. We, therefore, view our work as having an "educational" value to the software engineering community.

3. The mathematical formulation of a system dynamics model forces explication i.e., structural relationships between variables must be explicitly and precisely defined. As such, the model sets the foundation for the development of a theory of software project management. Paraphrasing Dubin (1971):

> A theory is the attempt of a man to model some aspects of the empirical world ... A theory tries to make sense out of the observable world by ordering the relationships among 'things' that constitute the theorist's focus of attention in the world 'out there' ... The process of putting things or units together in lawful relation to each other establishes the fundamental building blocks out of which a theory is constructed.

4. The high degree of explication required in the model helped us ferret out "knowledge gaps" in the literature. And a set of 27 interviews with software development managers in 5 organizations helped us fill these knowledge gaps. The model, therefore, incorporates new

findings about the management of software project management (e.g., on manpower acquisition policies under different scheduling considerations).

Case Study:

The model was developed on the basis of both an extensive review of the literature and information gathered through the set of 27 interviews. After the model was developed, we then conducted a case-study in a sixth organization, namely, the Systems Development Section of NASA's Goddard Space Flight Center. The objective of the case-study was to examine the model's ability to reproduce the dynamic behavior patterns of a completed software project.

The DE-A project was selected for the case-study by NASA. This project was selected so as to satisfy three criteria (furnished by us): (1) to be medium in size; (2) recent; and (3) "typical" i.e., one that would be considered as having been developed in a familiar in-house software development environment.

To simulate the DE-A project, the model was first parameterized. The process involved setting 14 model parameters that capture the particular DE-A project environment. The parameter values were obtained from two sources, namely, interviews at NASA and project

documentation. The 14 model parameters that were set (e.g., "Hiring Delay," "Turnover Rate," ... etc.), it is important to note, do not involve any changes in the formulation of the model's policy structures. The parameter set merely defines the (DE-A) environment within which the policies are exercised. This is significant, since the dynamic behavior patterns generated are largerly a result of the interaction of the model's (unchanged) policy structures.

The model was highly accurate in reproducing the actual development history of the DE-A software project. Specifically, it accurately reproduced the dynamic behavior patterns of the project's completion-date estimates, man-day estimates, cost in man-days, and workforce loading.

Experimentation:

If "understanding" is the intellectual outcome of a theoretical model, then "prediction" is its practical outcome (Dubin, 1971). The model was used as an experimentation vehicle to study/predict the dynamic implications of an array of managerial policies and procedures. Three areas were studied:

1. Software cost and schedule estimation. Three experiments were conducted in this area. In the first, we examined the impact that schedules have on project performance. We showed that "a different schedule

creates a different project." An important implication that follows from this is that both the project manager as well as the student of software estimation should reject the notion that a software estimation model can be adequately judged on the basis of how accurately it can estimate historical projects. Because of the significant influence that a schedule has on the behavior of a software project, the only real test of an estimation method is to try it. Furthermore, an estimation method should not be judged only on how accurate it is, but in addition it should be judged on how costly the projects it "creates" are.

The second experiment concerned the portability of quantitative software estimation tools. Evidence in the literature indicates that the portability of the currently available quantitative software estimation tools (i.e., from the companies in which they were developed to another) is poor (e.g., see (Boehm, 1981) and (Benbasat and Vessey, 1980)). A primary reason for this is that almost all the current models fail to explicitly account for the managerial characteristics of the software producing organization, and which tend to vary significantly from one organization to another (Mohanty, 1981). A major stumbling block has, heretofore, been the inability to quantify the impact of managerial-type factors on the cost of software

development. In this experiment we take a first step towards rectifying this situation. Specifically, we identified four aspects of a company's managerial environment (manpower acquisition, manpower allocation, effort distribution, and QA allocation) that significantly impact the cost of software development, and we quantified that impact. Because the four areas identified are variables that the project manager can objectively evaluate at the beginning of a software project, it should be feasible to incorporate them explicitly in future cost estimation models. This, we feel, will improve both the accuracy as well as the portability of such models.

The third and final experiment in this area concerned the analogy method of software estimation. The experiment generated two interesting insights. First, it revealed that there are inherent factors in the management of a software project (resulting from the interaction of manpower acquisition policies and personnel turnover) that would cause it to over-run even what would amount to be a "perfect" schedule estimate. The second, more interesting finding, is that because of this inherent tendency to overshoot, the use of the analogy method in estimating would inject a bias in the scheduling process, a bias that generates, in the long-run longer (than necessary) project schedules.

2. The economics of quality assurance (QA). Two sets of experiments were conducted in this area. The objective of the first set was to investigate, not whether QA is justified, but how much QA is justified. To do this we first examined the relationship between the QA effort expended and the % of errors detected during development. A significant feature of this relationship is the "diminishing returns" of QA exhibited as QA expenditures extend beyond 20-30% of development effort. We then derived the "optimal" QA expenditure level i.e., that level that minimizes total project cost. The "optimal" QA effort expenditure level (for our prototype project) was found to be 16% of the development man-days. What, in our opinion, is really significant about this result is not its value, since this cannot be generalized beyond the type of project used in our experiment, but rather the process of deriving it, namely, our integrative system dynamics approach. Beyond controlled experimentation (which is too costly and time consuming to be practically feasible) this model, as far as we know, provides the first capability to quantitatively analyze the costs/benefits of QA policy for software production. And this, it is encouraging to note, is generalizable, in the sense that one can customize models for different software development environments to derive

environment-specific optimality conditions. The results of this first set of QA experiments have also clearly demonstrated that QA policy does have a significant impact on total project cost. That is, QA expenditures that are significantly lower or significantly higher than the "optimal" can result in a significant increase in the project's total cost. At low values of QA expenditures this increase in cost results from the large cost of the testing phase. On the other hand, at high values of QA expenditures, the excessive QA expenditures are themselves the culprit.

The objective of the second set of QA experiments we conducted was to examine the sensitivity of the above results to two project variables, namely, the project's planned effort distribution profile (i.e., how management plans the distribution of effort among the development versus testing phases of the project), and the software development productivity. The findings constitute "rules-of-thumb" that organizations can use to adapt published results, or results from other organizations, to their own environment.

3. Staffing. Our objective in this, the third and final area of investigation, was to test the applicability of Brooks' Law to the domain of "medium-sized applications projects developed in a

familiar, in-house development environment."

Since its "enactment," Brooks' Law has been widely endorsed in the literature (e.g., see (Synnott and Gruber, 1981), (Paretta and Clark, 1976), (Pressman, 1982), (Jensen and Tonies, 1979), and (Boehm, 1981)). Furthermore, it has often been endorsed indiscriminately i.e., for not only large, but also small projects, and not only systems programming type projects, but also applications software systems. For example, Pressman (1982) extends Brooks' Law to 6-10 man-year projects, while in (Jensen and Tonies, 1979) and (Synnott and Gruber, 1981) it is extended to the domain of applications software systems. Brooks was quite explicit in specifying the domain of applicability of his Brooks' Law to what he calls "Jumbo" systems programming projects.

Our experimental results do <u>not</u> support Brooks' Law, for the type of project studied in this research. What our results show is that adding more people to a late project causes it to become more costly, but not to complete later.

The conclusion that we can draw from our experiment's results is that Brooks' Law does not universally apply to all software development

environments. And that, in particular, it does not seem to apply to the medium sized appliction project developed in a familiar, in-house development environment. It is, therefore, not necessarily an invalidation of Brooks' "Brooks' Law," but rather a disqualification of the notion (implied not by Brooks but by the writings of others in the literature) that "Brooks' Law" is a universal law of software development.

In a follow-up experiment, we re-tested Brooks' Law after quadrupling the training overhead in the project. Such an increase in the training overhead, while admittedly somewhat extreme, is never-the-less the kind of perturbation that we would need to make if we were to approximate the software development environment of large and complex systems programming software (Corbato and Clingen, 1980), e.g., such as the IBM OS/360. Under such conditions Brooks' Law applies, sometimes. The key is where in the life cycle people are added. Adding manpower to a late project can make it later only (our results indicate) if this takes place towards the end of the project's testing phase.

VI.2. Suggestions for Future Research:

According to Nobel Prize Winner Alfred Kastler "All

knowledge is provisional --- never final." This is certainly the case in this field where research is in the infancy stage. It is believed that this research has pointed up several areas requiring more intensive research.

Model Enhancements:

Further research needs to be performed within the framework of the existing model. We propose the following set of model extensions:

1. Incorporating the requirements definition/analysis phase into the model's life cycle. "The technology of defining the requirements for a software system is an area in most urgent need for improvement and itself constitutes a major portion of the so-called 'software-bottleneck'" (Bacon, 1982). Many in the field have hypothesized about the disruptive effects of changes in system requirements on software production, and on the direct link between such disruptions and cost/schedule slippages (Boehm, 1981). The system dynamics modeling approach provides a viable vehicle to test out such hypotheses, and to furnish a quantitative assessment of the claims made.

2. Extend the model to capture the development of multiple projects e.g., two software projects developed in parallel. In such an environment project competition

for company resources becomes a significant dimension, presenting an opportunity to examine the effects of various resource allocation policies e.g., of the manpower resource.

3. Extending the model to other project environments. Particularly interesting (and challenging) would be an extension to the larger DOD-type software projects (e.g., projects that are more than 1 million lines of code in size). Such an extension would entail a number of enhancements to the model. The development phase would be disaggregated into "finer" phases e.g., preliminary design, detailed design, and coding, with a set of formal milestones separating the phases e.g., preliminary design review, critical design review, ... etc. Interesting questions to investigate here are the cost/benefits of such milestones e.g., administrative overhead versus visibility benefits. It would be also of interest to investigate how and when serially planned phases are overlapped under schedule pressures, and the effects of such unplanned overlapping on the project. Another needed enhancement would be to restructure the QA activity, which in such projects tends to be conducted by an independent organization. As a third enhancement, it would be useful to capture the deep vertical structures that characterize the management of such "jumbo" projects, representing the communication

paths within the organization and including the various levels of information filtering and processing and of decision making.

4. Another interesting extension would be to capture the quality of the produced software product. The first issue to address here is formulating the measure(s) of software quality (e.g., usability, maintainability, .. etc.). A valuable resource to tap in this area is the work done in the software metrics field e.g., see Perlis et al, Software Metrics (1981). A number of model enhancements would then be required. For example, software errors could be disaggregated into different types, some more serious than others. Another more challenging enhancement would be tc capture the effects of motivational factors on quality. For example, experiments have shown that explicit project goals (e.g., "produce code as fast as you can" versus "produce maintainable code") significantly impact project behavior e.g., productivity, error rates, ... etc. (Weinberg and Schulman, 1974). This motivational issue is particularly interesting because the different software development objectives conflict with each other in practice. For example, pure concentration on minimizing the software development budget and schedule is likely to have negative effects on software quality, and vice versa (Boehm, 1981).

New Modeling Applications:

Rather than continuing to focus on software development projects per se, the system dynamics modeling approach outlined in this thesis could be extended to investigate a broader set of issues pertaining to the software development organization. That is, rather than trace the lifecycle(s) of one or more software projects, one would focus, instead, on the operations of a software development department as a continous 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, amd matrix) in different software development environments; (2) Personnel turnover, its costs (e.g., recruiting and training overheads), its benefits (e.g., access to new ideas and methodologies), and its causes (i.g., schedule pressures, maintenance load, ... etc.); (3) The impact of such management approaches as Management By Objectives (MBO) in both the short-term and the long-term (a system dynamics study in the R & D area showed that the short-run effect of MBO on increasing motivation and productivity may be reversed in the long-run if social interaction and communication are allowed to erode); and (4) the organizational/environmental determinants of productivity e.g., standards, software tools, use of librarians,

documentation requirements ... etc. Again, one needs to investigate both short-term as well as long-term implications. For example, because the software industry is unique in that we develop our own production tools, an investment in developing powerful software development tools (e.g., compilers, automated testing tools, ... etc.), while it might hamper productivity in the short-run, often leads to better software, which in turn could lead to even more powerful tools.

THE END

BIBLIOGRAPHY

1. Abdel-Hamid, T. K. and Madnick, S. E. "The System Dynamics Approach to Designing Software Project Planning and Control Systems: A Research Proposal." Technical Report, MIT, Sloan School of Management, January, 1982a.

2. Abdel-Hamid, T. K. and Madnick, S. E. "A Model of Software Project Management Dynamics." The 6TH Int'l Computer Software and Applications Conference (COMPSAC), (Nov., 1982b).

3. Abdel-Hamid, T. K. and Madnick, S. E. "An Integrative Approach to Modeling the Software Management Process: A Basis for Identifying Problems and Evaluating Tools and Techniques." IEEE Computer Society Workshop on Software Engineering Technology Transfer, (April, 1983).

4. Abdel-Hamid, T. K. and Madnick, S. E. "The Dynamics of Software Project Scheduling: A System Dynamics Perspective." Comm. of the ACM, (May, 1983), 340-346.

5. Ackoff, R. L. The Art of Problem Solving Accompanied by Ackoff's Fable. New York: John Wiley & Sons, Inc., 1978.

6. ADL, Inc. EDP and Systems Development: Some Management Issues. Report by Arthur D. Little, Inc.

7. Adrion, W. R. et al. "Validation, Verification, and Testing of Computer Software." ACM: Computer Surveys, Vol. 14, no. 2, (June, 1982), 159-192.

8. Aharonian, D. J. "Project Management Through the Accomplishment value Procedure (AVP)." National Computer Conference, 1979.

9. Alberts, D. S. "The Economics of Software Quality Assurance." National Computer Conference, 1976.

10. Albrecht, A. J. "Measuring Application Development Productivity." Proceedings of the Joint SHARE/GUIDE/IBM Apllication Development Symposium, (Oct., 1979).

11. Alloway, R. M. "User Managers' Systems Needs." MIS Quarterly, (June, 1983), 27-41.

12. Alpin, J. C. and Cosier, R. A. "Managing Creative and Maintenance Organization." The Business Quarterly, (Spring, 1980), 56-63.

13. Anthony, R. N. and Dearden, J. Management Control Systems. Illinois: Richard D. Irwin, Inc., 1980.

14. Anthony, R. N. Planning and Control Systems: A Framework for Analysis. Cambridge, Mass.: Harvard University Press, 1979.

15. Aron, J. D. "Estimating Reasons for Large Programming Systems." Software Engineering: Concepts and Techniques. Edited by J. M. Buxton, P. Naur and B. Randell. Litton Educational Publishing, Inc., 1976.

16. Arthur, L.J. Programmer Productivity. New York: John Wiley & Sons, Inc., 1983.

17. Artzer, S. P. and Neidrauer, R. A. "Software Engineering Basics: A Primer for the Project Manager." Unpublished thesis, Naval Postgraduate School, Monterey, Calif., 1982.

18. Arseven, S. M. "A System to Monitor and Control the Development and Documentation of a Computer programming Project." Unpublished Ph.D. dissertation, College of Texas A&M University, 1975.

19. Avery, R. D. and Hoyle, J. C. "A Guttman Approach to the Development of Behaviorally Based Rating Scales for Systems Analysts and Programmer/Analysts." Journal of Apllied Psychology, Vol. 59, No. 1, 61-68.

20. Ashenurst, R. L., ed. "Curriculum Recommendations for Gradute Professional Programs in Information Systems." ACM, Vol. 15, No. 15, (May, 1972), 363-398.

21. Ashton, R. H. "Deviation-Amplifying Feedback and Unintended Consequences of Management Accounting Systems." Accounting, Organization and Society,

Vol. 1, No. 4, (1976).

22. Auerbach Publishers Inc. "A Survey of Software Cost Models." Auerbach Publishers, Inc.

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

24. Bacon, G. "Software." Science, Vol. 215, (Feb., 1982), 775-779.

25. Baker, F. T. "Chief Programmer Team Management of Production Programming," IBM Systems Journal, Vol. 11, No. 1, 1972.

26. Baker, F. T. and Mills, H. D. "Chief Programmer Teams." Datamation, (Dec., 1973), 58-61.

27. Bailey, J. W. and Basili, V. R. "A Meta-Model for Software Development Resource Expenditures," Proceedings, 5th Int'l Conference on Software Engineering, IEEE/ACM/NBS, (Mar., 1981), 107-116.

28. Barker, F. T. "System Quality Through Structured Programming." AFIPS, (Fall, 1972), 339-343.

29. Barndt, S. E. "Upward Communication Filtering in the Project Management Environment." Project Management Quarterly, (March, 1981), 39-43.

30. Bartol, K. M. and Martin, D. C. "Managing Information Systems Personnel: A Review of the Literature and Managerial Implications." MIS Quarterly, (Dec., 1982), 49-70.

31. Bartol, K. M. "Professionalism as a predictor of Org'l Commitment, Role, Stress, and Turnover: A Multidimensional Approach." Academy of Mgmt Journal, Vol. 22, No. 4, (Dec., 1979), 815-821.

32. Bartol, K. M. "Turnover Among DP Personnel: A Causal Analysis." Working Paper, Univerisity of Maryland, College Park, Maryland, 1981.

33. Basili, V. R. and Zelkowitz, M. V. "Measuring Software Development Characteristics in the Local Environment." Computers & Structures, Vol. 10, 1979, 39-43.

34. Basili, V. R. and Weiss, D. M. "Evaluation of a Software Requirements Document by Means of Change Data." Proceedings, 5th Int'l Conference on Software Engineering, IEEE, March, 1981.

35. Basili, V. R. and Zelkowitz, M. V. "Analyzing Medium - Scale Software Development." Proceedings of the 3rd Int'l Conference on Software Engineering, IEEE/ACM/NBS, (May, 1978).

36. Basili, V. R. "Improving Methodology and Productivity Through Practical Measurement." A Lecture at the Wang Institute of Graduate Studies, Lowell, Mass., (Nov., 1982).

37. Basili, V. R. and Weiss, D. M. Evaluating Software Development by Analysis of Changes: The Data From the Software Engineering Laboratory. Computer Science Technical Report Series, TR-1236. College Park, Maryland, University of Maryland, (Dec., 1982).

38. Basili, V. R. and Zelkowitz, M. V. "Measuring Software Development Characteristics in the Local Environment." Computer & Structures, Vol, 10, 1979, 39-43.

39. Basili, V. R. and Reiter, R. W., Jr. "An Investigation of Human Factors in Software Development." Computer, (Dec., 1979), 21-38.

40. Basili, V. R. "Resource Models." Software Metrics. Edited by A. J. Perlis et al. Cambridge, Mass: The M.I.T. Press, 1981.

41. Bauer, F. L., ed. Software Engineering. Berlin: Springer-Verlag, 1977.

42. Bauer, F. L. "Software Engineering." Software Engineering. Edited by F. L. Bauer. Berlin: Springer-Verlag, 1977.

43. Bauer, F. L. "A Trend for the Next 10 Years of Software Engineering." Software Engineering. Edited by H. Freeman and P. M. Lewis II. New York: Academic Press, Inc., 1980.

44. Beck, L. L. and Perkins, T. E. "A Survey of Software Engineering Practice: Tools, Methods, and Results." IEEE Trans. on Software Engineering, Vol. SE-9, No. 5, (Sept., 1983).

45. Beeler, J. "Exec. Identifies Seven Reasons Why DPers Quit." ComputerWorld, (April, 1982), 25.

46. Belady, L. A. and Lehman, M. M. "Characteristics of Large Systems." Research Directions in Software Technology. Edited by Wegner. Cambridge, Mass: The M.I.T. Press, 1979.

47. Belford, P. C., et al. "An Evaluation of the Effectiveness of Software Engineering Techniques." IEEE COMPCON, (Fall, 1977).

48. Benbasat, I. and Vessey, I. "Programmer and Analyst Time/Cost Estimation." MIS Quarterly, Vol. 4, No. 2, (June, 1980), 31-43.

49. Biggs, C. L., et al. Managing the Systems Development Process. Englewood Cliffs, N.J.: Prentice-Hall, Inc., 1980).

50. Blake, R. and Mouton, J. S. Productivity: The Human Side. New York: American Management Assoc., 1981.

51. Blake, R. "Effects of Modern Programming Practices on Software Development Costs." IEEE COMPCON, (Fall, 1977).

52. Block, R. The Politics of Projects. New York: Yourdon Press, 1983.

53. Blum, B. I. "A Methodology for Information Systems Production." Society for General Systems Research, (Jan., 1982).

54. Boebert, W. E. "Software Quality Through Software Management." Software Quality Management. Edited by J. D. Cooper and M. J. Fisher. New York: Petrocelli Book, Inc., 1979.

55. Boehm, B. W. "An Experiment in Small-Scale Application Software Engineering." IEEE Trans. Software Engineering. 1981.

56. Boehm, B. W. Software Engineering Economics. Englewood Cliffs, N.J.: Prentice-Hall, Inc., 1981.

57. Boehm, B. W. "Software Engineering: R & D Trends and Defense Needs." Research Directions in Software Technology. Edited by P. Wegner. Cambridge, Mass: The M.I.T. Press, 1979.

58. Boehm, B. W. "An Experiment in Small-Scale Application Software Engineering." IEEE Trans. Software Engineering, 1981.

59. Boehm, B. W. and Wolverton, R. W. "Software Cost Modeling: Some Lessons Learned." Journal of Systems and Software, Vol. 1, No. 3, 1980.

60. Boehm, B. W. "Software Engineering." IEEE Trans. Computers, (Dec., 1976).

61. Boehm, B. W. "Software and its Impact: A Quantitative Assessment." Datamation, 1976.

62. Boehm, B. W., et al. "Some Experience with Automated Aids to the Design of Large-Scale Reliable Software." Proceedings of the Int'l Conference on Reliable Software, (April, 1975).

63. Boehm, B. W. "Software Engineering." Software Engineering. Edited by H. Freeman and P. M. Lewis II. New York: Academic Press, Inc., 1980.

64. Bott, H. S. "The Personnel Crunch." Perspectives on Information Management. Edited by J. B. Rochester. New York: John Wiley & Sons, Inc., 1982.

65. Brandon, D. H. "The Economics of Computer Programming." On The Management Of Computer Programming. Edited by G. Weinwurm. Princeton, N. J.: Auerbach Publishing, Inc., 1970.

66. Brodie, S. "Managers Can Sow the Seeds of Productivity." Data Management, (Mar., 1983).

67. Brooks, F. P. Jr. The Mythical Man-Month. Reading, Mass: Addison-Wesley Publishing Co., 1978.

68. Brooks, F. P. "Why is the Software Late." Data Management, (Aug., 1971), 18-21.

69. Brooks, R. "Towards a Theory of the Cognitive Processes in Computer Programming." Int'l Journal Man-Machine Studies, vol. 9, (1977), 737-751.

70. Brooks, W. D. "Software Technology Payoff: Some Statistical Evidence." IBM-FSD, Bethesda, MD, (April, 1980), 2-7.

71. Brown, J. R. "Programming Practices for Increased Software Quality." Software Quality Management. Edited by J. D. Cooper and M. J. Fisher. New York: Petrocelli Books, inc., 1979, 197-208.

72. Bruggere, T. H. "Software Engineering: Management, Personnel and Methodology." Proceedings of the 4th Int'l Conference Software Engineering, 1979.

73. Bruce, P. and Pederson, S. M. The Software Development Project: Planning and Management. New York: John Wiley & Sons, Inc., 1982.

74. Bryan, W., et al. "Software Configuration Management."

Proceedings of the 4th Int'l Computer Software & Applications Conference, IEEE, N.Y., 1980.

75. Bryant, J. H. "Survey of Values and Sources of Dissatisfaction." Data Management, (Feb., 1976), 34-37.

76. Burchett, R. "Avoiding Disaster in Project Control." Data Processing Digest, Vol. 28, No. 6, (June, 1982), 1-3.

77. Burton, B. J. "Manpower Estimating for Systems Projects." Journal of Systems Management, (Jan., 1975), 29-33.

78. Burrows, J. "Software Engineering." Research Directions in Software Technology. Edited by P. Wegner. Cambridge, Mass: The M.I.T. Press, 1979.

79. Cameron, K. S. and Whetten, D. A. "Perceptions of Organizational Effectiveness over Organizational Life Cycle." Admins. Science Quarterly, Vol. 26, (1981), 525-544.

80. Canning, R. G. "Managing Staff Retention and Turnover." EDP Analyzer, (Aug., 1977), 1-13.

81. Canning, R. G. "Project Management Systems." EDP Analyzer, (Sept., 1976), 1-13.

82. Canning, R. G. "Progress in Project Management." EDP Analyzer, (Dec., 1977), 1-11.

83. Canning, R. G. "Using Some New Programming Techniques." EDP Analyzer, (Nov., 1977), 1-13.

84. Canning, R. G. "Progress in Software Engineering: Part 1." EDP Analyzer, (Feb., 1978), 1-13.

85. Canning, R. G. "Progress in Software Engineering: Part 2." EDP Analyzer, (Mar., 1978), 1-13.

86. Caudill, R. "Understanding the Development Life Cycle." National Computer Conference, (1977), 269-275.

87. Cave, W. C. and Salisbury, A. B. "Controlling the Software Life Cycle - The Project Management Task." IEEE Trans. on Software Engineering, Vol. SE-4, No. 4, (July, 1978), 326-334.

88. Chmura, L. J. and Weiss, D. M. The A-7E Software Requirements Document: Three Years of Change Data. NRL Memorandum Report 4938, Washington, D.C., (Nov., 1982).

89. Cho, C. An Introduction to Software Quality Control. New York: John Wiley & Sons, Inc., 1980.

90. Chrysler, E. "Some Basic Determinants of Computer Programming Productivity." Comm of ACM, Vol. 21, No. 6, (June, 1978), 472-483.

91. Chrysler, E. "Programmer Performance Standards." Journal of Systems Management, (Feb., 1978), 18-25.

92. Chrysler, E. "The Impact of Program and Programmer Characteristics on Program Size." National Computer Conference, (1978), 581-587.

93. Chrysler, E. "Improved Management of Information." Journal of Systems Management, (Mar., 1970), 6-13.

94. Chrysler, E. "Computer Programming Productivity." Advances in Computer Programming Management. Edited by T. A. Rullo. Philadelphia, Pa: Heyden & Sons, Inc., 1980.

95. Clapp, J. A. "A Review of Software Cost Estimation Methods." MITRE Technical Report, (June, 1976), 1-55.

96. Cleland, D. I. and King, W. R. Management: A Systems Approach. New York: McGraw-Hill Book, Inc., 1972.

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

98. Cohen, K. J. and Cyert, R. M. "Computer Models in Dynamic Economics." Quarterly Journal of Economics, Vol. 75, (1961), 112-127.

99. Cohen, K. J. and Cyert, R. M. "Simulation of Organizational Behavior." Handbook of Organizations. Edited by J. G. March. Chicago: Rand McNally & Co., 1965.

100. Comper, F. A. "Project Management for System Quality and Development Productivity." Share-Guide, 1979, 17-23.

101. Connor, D. A. "Application Systems Development Methodologies." ComputerWorld.

102. Conway, M. E. "How do Committees Invent." Datamation, (April, 1968), 28-31.

103. Cooper, J. D. "Software Engineering: R & D Trends and Defense Needs." Research Directions in Software

Technology. Edited by P. Wegner. Cambridge, Mass: The M.I.T. Press, 1979.

104. Cooper, J. D. and Fisher, M. J., eds. Software Quality Management. New York: Petrocelli Book, Inc., 1979.

105. Cooper, K. G. "Naval Ship Production: Aclaim Settled and a Framework Built." Interfaces, Vol 10, No. 6, (Dec., 1980).

106. Corbato, F. J. and Clingen, C. T. "A Managerial View of the Multics System Development." Research Directions in Software Technology. Edited by P. Wegner. Cambridge, Mass: The M.I.T. Press, 1979.

107. Cougar, J. D. "Circular Solutions." Datamation, (Jan., 1983), 135-142.

108. Cougar, J. D. and Zawacki, R. A. Motivating and Managing Computer Personnel. New York: John Wiley & Sons, Inc., 1980.

109. Craig, C. E. and Harris, R. C. "Total Productivity Measurement at the Firm Level." Sloan Management Review, (Spring, 1973), 13-29.

110. Crossman, T. D. "Taking the measure of Programmer Productivity." Datamation, (May, 1979), 144-147.

111. Crowley J. D. "The Application Development Process: What's Wrong With it?" JDC Associates.

112. Cruickshank, R. D. and Lesser, M. "An Approach to Estimating and Controlling Software Development Costs." The Economics of Information Processing. Edited by R. Goldberg and H. Lorin. New York: John Wiley & Sons, Inc., 1982.

113. Curtis, B. Evaluation of Software Life Cycle Data From the Pave Paws Project. Final Technical Report, RADC-TR-80-28, (Mar., 1980).

114. DACS. Quantitative Software Models. New York: Griffiss Air Force Base, 1979.

115. Daly, E. B. "Management of Software Development." IEEE Trans. on Software Engineering, (May, 1977).

116. Daly, E. B. "Organizational Philosophies Used in Software Development." The Economics of Information Processing. Edited by R. Goldberg and H. Lorin. New York: John Wiley & Sons, Inc., 1982.

117. Davis, G. B. Management Information System: Conceptual Foundations, Structure and Development. New York: McGraw-Hill, Inc., 1974.

118. DeMarco, T. Yourdon Project Survey. (Sept., 1981).

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

120. DeMillo, R. A., et al. Software Project Forecasting. NTIS, U.S. Dept. of Commerce, (Oct., 1980).

121. DeRose, B. C. and Nyman, T. H. "The Software Life Cycle." Research Directions in Software Technology, Edited by P. Wegner. Cambridge, Mass: The M.I.T. Press, 1979.

122. Deutsch, M. S. "Verification and Validation." Software Engineering. Edited by R. W. Jensen and C. C. Tonies. New Jersey: Prentice-Hall, Inc., 1979.

123. Devenny, T. J. "An Exploration Study of Software Cost Estimating at the Electronic Systems Division." NTIS, U.S. Dept. of Commerce, (July, 1976).

124. Dickson, G. W. and Wetherbe, J. C. Increasing the Productivity of MIS Personnel: The Motivation Issue. MISRC-WP-81-80. University of Minnesota, 1981.

125. Dickson, G. W., et al. "The Management Information Systems Area: Problems, Challenges, and Opportunities." Data Base, Vol. 14, No. 1, (Fall, 1982), 7-12.

126. Dijkstra, E. W. "Notes on Structured Programming." Structured Programming. Edited by G. J. Dahl and C. A. R. Hoare. New York: Academic Press, 1971.

127. Dimino, S. A. "Management of Systems Analysts." Journal of Systems Management, (Mar., 1982), 38-40.

128. Distaso, J. R. "Software Management - A Survey of the Practice in 1980." Proceedings of the IEEE, Vol. 68, No. 9, (Sept., 1980), 1103-1119.

129. DOD. Strategy for a DOD Software Initiative. Dept. of Defense, 1982.

130. DOD. Strategy for a DOD Software Initiative: Vol. II, Appendices. Dept. of Defense, 1982.

131. Donaldson, H. A Guide to the Successful Management of Computer Projects. New York: John Wiley & Sons, Inc., 1978.

132. Donelson, W. S. "Project Planning and Control." Datamation, (June, 1976).

133. Driscoll, A. J. "Software Visibility and the Program Manager." Defense Systems Management Review, Vol. I, No. 2, 12-27.

134. Dubin, R. The Organization, Management and Tactics of Social Research. Edited by R. O'Toole. Cambridge, Mass: Schenkman Publishing Co., Inc., 1971.

135. Edelman, R. M. "Engineering Manpower Resource Management in a Multi-Project Environment." Unpublished S.M. Thesis, M.I.T., Sloan School of Management, Cambridge, Mass, 1975.

136. Ely, E. H. Software Management: A Dynamic Approach. Report for the Defense Systems Management College, (May, 1977).

137. Emery, J. C. Organizational, Planning and Control Systems: Theory and Technology. New York: Macmillan Publishing Co., Inc., 1969.

138. Endres, A. B. "An Analysis of Errors and their Causes in System Programs." IEEE Trans. Software Engineering. (June, 1975), 140-149.

139. Ergott, H. L. Jr. "Introduction: Software Quality Management as a Discipline." Software Quality Management. Edited by J. D. Cooper and M. J. Fisher. New York: Pertocelli Books, Inc., 1979.

140. Esterling, B. "Software Manpower Costs: A Model." Datamation, (Mar., 1980), 164-170.

141. Etzioni, A. "Two Approaches to organizational Analysis: A Critique and a Suggestion." Admin. Science Quarterly, 257-278.

142. Fagan, M. E. "Design and Code Inspections to Reduce Errors in Program Development." IBM Systems Journal, Vol. 15, No. 3, 1976.

143. Farbman, D. M. "Myths That Miss." Datamation, (Nov., 1980), 109-114.

144. Farquhar, J. A. A Preliminary Inquiry into the Software Estimation Process. Technical Report, AD F12 052,

Defense Documentation Center, Alexandria, Va., (Aug., 1970).

145. Ferrentino, A. B. "Making Software Development Estimates 'Good'." Datamation, (Sept., 1981), 179-182.

146. Fife, D. W. Industrial Dynamics. Cambridge, Mass: The M.I.T. Press, 1961.

147. Fireworker, R. B. and Bogner, L. J. Jr. "Improved Software Development Through Project Management." Data Management, (Dec., 1980), 27-39.

148. Fisher, M. J. and Light, W. R. Jr. "Definitions in Software Quality Management." Software Quality Management. Edited by J. D. Cooper and M. J. Fisher. New York: Petrocelli Books, Inc., 1979.

149. Fitz-enz, J. "Who is the DP Professional?" Datamation, (Sept., 1978), 125-128.

150. Fleckenstein, W. O. "Challenges in Software Development." Computer, Vol. 16, No. 3, (Mar., 1983), 60-64.

151. Fleischer, R. J. and Spitler, R. W. "Simon: A Project Management System for Software Development." Computer Software Engineering, (April, 1976).

152. Ford, A. "A Practical Approach to Sensitivity Testing of System Dynamics Models." The 1983 Int'l System Dynamics Conference. Chestnut Hill, Mass, (July, 1983).

153. Ford, J. A. "The Suitability of Matrix Management for Development Project - A Review." Project Management Quarterly. (Mar., 1982).

154. Forrester, J. W. "Industrial Dynamics - After the First Decade." Management Science. Vol. 14, No. 7, (Mar., 1968), 398-415.

155. Forrester, J. W. System Dynamics - Future Opportunities. D-3108-1, (July, 1979).

156. Forrester, J. W. "Counter Intuitive Behavior of Social System." Technology Review, Vol 74, No. 3, (Jan., 1971).

157. Forrester, J. W. Information Sources for Modeling the National Economy. D-3114-1, (Aug., 1979).

158. Forrester, J. W. Industrial Dynamics. Cambridge, Mass:

The M.I.T. Press, 1961.

159. Fox, J., ed. Computer Software Engineering. New York: Polytechnic Press of the Polytechnic Institute of New York, 1976.

160. Frank, W. L. "The New Software Economics." Perspectives on Information Management: A Critical Selection of ComputerWorld Features Articles. Edited by J. B. Rochester. New York: John Wiley & Sons, Inc., 1982.

161. Frank, W. L. Critical Issues in Software: A Guide to Software Economics, Strategy, and Profitability. New York: John Wiley & Sons, Inc., 1983.

162. Freedman, D., et al. "Organizing and Training for a New Software Development Project - That Big First Step." National Computer Conference, 1977.

163. Freedman, D. P. and Weinberg, G. M. Handbook of Walkthroughs, Inspections, and Technical Reviews. Boston: Little, Brown and Co., Inc., 1982.

164. Freeman, H. and Lewis, P. M. II, eds. Software Engineering. New York: Academic Press, Inc., 1980.

165. Fries, M. J. Software Error Data Acquisition. Boeing Aerospace Co., Seatle, WA., AD/A-039 9 16, (April, 1977).

166. Gaffeny, J. E. "Maximize Design Effort and Minimize Program Control Complexity - To vity." IEEE COMPSAC, (Oct., 1980).

167. Gaffeny, J. E. "A Macro-Analysis Methodology for Assessment of Software Development Costs." The Economics of Information Processing. Vol. 2: Operations, Programming, and Software Models. Edited by R. Goldberg and H. Lorin. New York: John Wiley & Sons, Inc., 1982.

168. Gagliardi, U. Classnotes, Harvard University, Cambridge, Mass, 1981.

169. Gansler, J. S. "Keynote: Software Management." The Symposium on Computer Software Engineering. Polytechnic Institute of New York, (April, 1976).

170. Gayle, J. B. "Multiple Regression Techniques for Estimating Computer Programming Costs." Journal of Systems Management. (Feb., 1971), 13-16.

171. Gehring, P. F. Jr. "A Quantitative Analysis of Estimating Accuracy in Software Development." Unpublished Ph.D. dissertation, Texas A&M University, 1976.

172. Gehring, P. F. and Pooch, V. W. "Software Development Management." Data Management, (Feb., 1977), 14-38.

173. General Research Corp. "Cost Reporting Elements and Activity Cost Tradeoffs for Defense System Software." Santa Clara, Calif., (May, 1977).

174. Geogopoulos, B. S. and Tannenbaum, A. S. "A Study of Organizational Effectiveness." The Annual Conference of the American Assoc. for Public Opinion Research, (May, 1957).

175. Gilb, T. Software Metrics. Winthrop, Cambridge, Mass, 1977.

176. Gildersleeve, T. R. "Organizing the Data Processing Functions." Datamation, (Nov., 1974), 46-50.

177. Glass, R. L. The Universal Elixir and Other Computing Projects which failed. Seatle, Wa: R. L. Glass, 1977.

178. Glass, R. L. Modern Programming Practices: A Report From Industry. New Jersey: Prentice-Hall, 1982.

179. Glass, R. L. Software Reliability Guidebook. New Jersey: Prentice-Hall, 1979.

180. Glassman, S. Comparative Studies in Software Acquisition. Lexington, Mass: D. C. Heath & Co., 1982.

181. Gluckson, F. A. "Professional Development for Computer Programmers." Advances in Computer Programming Management. Edited by T. A. Rullo. Philadelphia, Pa: Heyden & Sons, Inc., 1980.

182. Goldberg, R. and Lorin, H., eds. The Economics of Information Processing. Vol. 1: Management Perspectives. New York: John Wiley & Sons, Inc., 1982.

183. Goldberg, R. and Lorin, H., eds. The Economics of Information Processing. Vol. 2: Operations, Programming and Software Models. New York: John Wiley & Sons, Inc., 1982.

184. Golden, J. R., et al. "Software Cost Estimating: Craft or Witchcraft." Data Base, Vol. 12, No. 3,

(Spring, 1981), 12-14.

185. Goodenough, J. B. and McGowan, C. L. "Software Quality Assurance: Testing and Validation." Proceedings of the IEEE, Vol 28, No. 9, (Sept., 1980).

186. Goodman, L. P. and Goodman, R. A. "Some Management Issues in Temporary Systems: A Study of Professional Development and Manpower - The Theater Case." Admins. Science Quarterly, Vol. 21, (Sept., 1976), 494-501.

187. Gordon, R. L. and Lamb, J. C. "A Close Look at Brooks' Law." Datamation, (June, 1977), 81-86.

188. Gotterer, M. H. "The Computer Manager and his Job." Proceedings of the 7th Annual Computer Personnel Research Conference. Edited by M. Flin, June, 1969.

189. Gotterer, M. H. "Management of Computer Programmer." Joint Computer Conference. (Spring, 1969).

190. Gould, J. D. and Drongowski, P. "An Explanatory Study of Computer program Debugging." Human Factors, Vol. 16, No. 3, 1974, 258-277.

191. Gould, J. D. "Some Psychological Evidence on How People Debug Computer Programs." Int'l Journal of Man-Machine Studies, Vol. 7, 1975, 151-182.

192. Gould, R. S. "A Self-Assessment Dealing Management." Comm. of ACM, Vol. 25, No. 12, (Dec., 1982), 883-887.

193. Graham, A. K. "Parameter Estimation in System Dynamics Modeling." Elements of the Systems Dynamics Methods. Edited by J. Randers. Cambridge, Mass: The M.I.T. Press, 1980.

194. Green, L. H. "Organizing for Project Management." Systems Development Management. Edited by J. Hannan. New Jersey: Auerbach Publishers, Inc., 1982.

195. Green, P. E. and Tull, D. S. Research for Marketing Decisions. Englewood Cliffs, New Jersey: Prentice-Hall, Inc., 1978.

196. Gremillion, L. L. "Systems Development and Implementation Costs Using Standardized Applications Systems." The Economics of Information Processing. Vol. 2: Operations, Programming and Software Models. Edited by R.

Goldberg and H. Lorin. New York: John Wiley & Sons, Inc., 1982.

197. Guetzkow, H. S., et al. Simulation in Social & Admins. Science: Overviews and Case Examples. Englewood Cliffs, New Jersey: Prentice-Hall, 1972.

198. Gunther, R. C. Management Methodology for Software Product Engineering. New York: John Wiley & Sons, Inc., 1978.

199. Hales, K. A. "Software Management Lessons Learned - The Hard Way." The 6th Int'l Computer Software and Applications Conference (COMPSAC), (Nov., 1982a).

200. Hallam, S. F. "An Empirical Investigation of the Objectives and Proceeding Depts." Academy of Management Journal, (Mar., 1975).

201. Hallam, S. F. "EDP Objectives and the Evaluation Process." Data Management, (May, 1979), 40-50.

202. Hammer, C. "Life Cycle Management." Information & Management, Vol. 4, (1981), 71-80.

203. Hancock, W. C. "Practical Application of Three Basic Algorithms in Estimating Software Systems." The Economics of Information Processing. Vol 2: Operations, Programming, and Software Models. Edited by R. Goldberg and H. Lorin. New York: John Wiley & Sons, Inc., 1982.

204. Hannan, J., ed. Computer Programming Management. Pennsauken, New Jersey: Auerbach Publishers, Inc., 1982.

205. Hannan, J., ed. {Systems Development Management. New Jersey: Auerbach Publishers, Inc., 1982.

206. Hart, J. J. "The Effectiveness of Design and Code Walkthrough." The 6th Int'l Computer Software and Applications Conference (COMPSAC), (Nov., 1982a).

207. Hartwick, R. D. "Software Testing." Advances in Computer Programming Management. Edited by T. A. Rullo. Philadelphia, Pa: Heyden & Sons, Inc., 1980.

208. Hausen, H. L. and Mullerburg, M. "Software Engineering Environment: State of the Art, Problems and Perspectives." The 6th Int'l Computer Software and Applications Conference (COMPSAC), (Nov., 1982a).

209. Herd, J. H., et al. Software Cost Estimation Study.

Vol I. Maryland: Doty Assoc. Rockville, NTIS, 1977.

210. Herndon, M. A. and Lane, J. A. "Analysis of Software Errors for Cost Factors." American Institute of Aeronautics and Astronautics, Inc., 1977.

211. Hollornan, D. J. "Systems Development Quality Control." MIS Quarterly, Vol. 2, No. 4, (Dec., 1978), 1-14.

212. Hormer, J. B. "A Dynamic Model for Analyzing the Emergence of New Medical Technologies." Unpublished Ph.D. dissertation, M.I.T., Cambridge, Mass, 1983.

213. Houghton, R. C. "Software Development Tools: A Profile." Computer. Vol. 16, No. 5, (May, 1983), 63-70.

214. Ibrahim, R. L. "Software Development Information System." Journal of Systems Management, (Dec., 1978), 34-39.

215. Inbar, and Stoll, Simulation and Gaming in Social Science.

216. Ingham, A. G. "The Ringelmann Effect: Studies of Group Size and Group Performance." Journal of Experimental Social Psychology. Vol. 10, (1974), 371-384.

217. Ingrassia, F. S. "The Unit Development Folder (UDF): An Effective Management Tool for Software Development." TRW Technical Report. TRW-SS-76-11.

218. Issac, S. and Michael, W. Handbook in Research and Evaluation. San Diego, Ca: Edits Publishers, 1971.

219. Ives, B. and Olson, M. H. "Manager or Technician? The Nature of the IS Manager's Job." MIS Quarterly, Vol. 5, No. 4, (Dec., 1981), 49-83.

220. Jacques, E. Equitable Payment. New York: John Wiley & Sons, Inc., 1961.

221. Jahnig, F. F. "Skills Matrixing." Datamation, (Sept., 1975), 71-76.

222. Jensen, R. W. and Tonies, C. C. Software Engineering. Englewood Cliffs, New Jersey: Prentice-Hall, 1979.

223. Johnson, B. "People, Money at Root of DP Exec's Problems." ComputerWorld, (Sept, 1982), 16.

224. Johnson, J. R. "A Working Measure of Productivity." Datamation, (Feb., 1977), 106-109.

225. Johnson, J. R. "Advanced Project Control." Journal of Systems Management, (May, 1977).

226. Johnson. J. R. Managing for Productivity in Data Processing. Wellesley, Mass: QED, Information Sciences, Inc., 1980.

227. Jones, M. M. and McLean, E. R. "Management Problems in Large Scale Software development Projects." Industrial Management Review, (Spring, 1970), 1-15.

228. Jones, T. C. "Defect Removal: A Look at the State of the Art." ITT CommNet, Vol. 1, No. 3, (Dec., 1981).

229. Jones, T. C. "Measuring Programming Quality and Productivity." IBM Systems Journal, Vol. 17, No. 1, 1978, 39-63.

230. Jones, T. C. "The Limits of Prgramming Productivity." Proceedings of the 14th Annual Conference of the Society for Information Management, Chicago, Sept., 1982.

231. Jones, T. C. "Program Quality and Programmer Productivity." IBM, TR 02.764, 28, (Jan., 1977).

232. Jones, T. C. "A Survey of Programming Design and Specification Techniques." Proceedings of the IEEE Specifications of Reliable Software Conference, (Mar., 1979).

233. Katz, R. "The Effects of Group Longevity on Project Communication and Performance." Admins Science Quarterly, Vol. 27, (1982), 81-104.

234. Kay, R. H. "The Management and Organization of Large Scale Software Development Projects." Joint Computer Conference, (Spring, 1969), 425-433.

235. Keider, S. P. "Why Projects Fail." Datamation, (Dec., 1974).

236. Kelly, T. J. "The Dynamics of R & D Project Management." Unpublished M.S. Thesis, M.I.T., Sloan School of Management, Cambridge, Mass, 1970.

237. Kerzner, H. "Tradeoff Analysis in a Project." Journal of Systems Management, (Oct., 1982), 6-13.

238. Kirby, E. J. "The Systems Development Manager." Systems Development Management. Edited by J. Hannan. New Jersey: Auerbach Publishers, Inc., 1982.

239. Kleiner, B. H. "Integrating Major Motivational Theories." Journal of Systems Management, (Feb., 1983), 26-29.

240. Knight, B. M. "Organizational Planning for Software Quality." Software Quality Management. Edited by J. D. Cooper and M. J. Fisher. New York: Petrocelli Books, Inc., 1979.

241. Knutson, J. "Developing the Project Plan." Advances in Computer Programming Management. Philadelphia, Pa: Heyden & Sons, Inc., 1980.

242. Kolence, K. "Software Engineering Management and Methodology." Software Engineering: Report on a Conference Sponsored by the NATO Science Committee. Edited by P. Naur and B. Randell. Vol. 13, (Oct., 1968).

243. Koolhass, Z. Organization Dissonance and Change. New York: John Wiley & Sons, Inc., 1982.

244. Kootz, H. and O'Donnel, C. Principles of Management: An Analysis of Management Functions. 5th ed. New York: McGraw-Hill Books Co., 1972.

245. Kossiakoff, A., et al. DOD Weapon Systems Software Management Study. NTIS, Ad/A-022, 160, (June, 1975).

246. Kotter, J. P. Organizational Dynamics: Diagnosis and Intervision. Reading, Mass: Addison-Wesley Publishing Co., Inc., 1978.

247. Kraft, P. Programmers and Managers - The Routinization of Computer Programming in the U.S. New York: Springer-Varlag N. Y., Inc., 1977.

248. Kustanowitz, A, L. "System Life Cycle Estimation (Slice:) A New Approach to Estimating Resources for Application Program Development." COMPSAC, 1977.

249. LaBelle, C. D., et al. "Solving the Turnover Problem." Datamation, (April, 1980), 144-152.

250. Larkin, J. E. "The Psychology of DP Professional: A Career Planning Tool." ComputerWorld, (CW-0221).

251. Lasden, M. "Overcoming Obstacles to Project Success." Computer Decisions, (Dec., 1981), 114-177.

252. Latane, B., et al. "Many Hands Make Light the Work: The Causes and Consequences of Social Loafing." Journal of Personality and Social Psychology, Vol. 37, No. 6, (1979), 822-832.

253. Lave, C. A. and March, J. G. An Introduction to Models in the Social Sciences. New York: Harper & Row, 1975.

254. Lawler, E. E. and Rhode, J. G. Information and Control in Organizations. Pacific Palisades, Ca: Goodyear Publishing Co., Inc., 1976.

255. Leavitt, H. J. Managerial Psychologoly. 4th ed. Chicago: The University of Chicago Press, 1978.

256. Lehman, M. M. Laws and Conservation in Large Program Evaluation. Second Software Life Cycle Management Workshop. Atlanta, Ga, (Aug., 1978), 21-22.

257. Lehman, J. H. "How Software Projects are Really Managed." Datamation, (Jan., 1979), 119-129.

258. Levene, A. A. "Reducing the Risk of Failure in Computer System Development." Information Technology for the Eighties. Edited by R. D. Parslow. Philadelphia, Pa: Heyden & Sons. Ltd., 1981.

259. Lewis, R. O. "Software Verification and Validation." Software Quality Management. Edited by J. D. Cooper and M. J. Fisher. New York: Petrocelli Books, Inc., 1979.

260. Lientz, B. P. and Swanson, E. B. "Problems in Application Software Maintenance." Comm of ACM, Vol. 24, No. 11, (Nov. 1981), 763-769.

261. Loftin and Moosbruker. "Organization Development Methods in the Management of the Information Systems Function." MIS Quarterly, (Sept., 1982).

262. Lorsch, J. W. "Organization Design: A Situational Perspective." Organizational Dynamics, (Autumn, 1977).

263. Lyneis, J. M. Corporate Planning and Policy Design - A System Dynamics Approach. Cambridge, Mass: The M.I.T. Press, 1980.

264. Maciariello, J. A. Program - Management Control Systems. New York: John Wiley & Sons, Inc., 1978.

265. Mantei, M. "The Effort of Programming Team Structures on Programming Tasks." Comm. of A ACM, Vol 24, No. 3, (Mar., 1981), 106-113.

266. Martin, E. W. "A Framework for MIS Software Development Projects." MIS Quarterly, Vol. 3, No. 1, (Mar., 1979), 29-38.

267. Martin, J. Application Development Without Programmers. Englewood Cliffs, New Jersey: Prentice-Hall, Inc., 1982.

268. Matejka, J. W. and Sandler, G. H. "Software Project Control - Yesterday's Dream, Tomorrow's Reality." AIAA.S

269. McCall, J. A. "An Introduction to Software Quality Matrics." Software Quality Management. Edited by J. D. Cooper and M. J. Fisher. New York: Petrocelli Books, Inc., 1979.

270. McCracken, D. D. and Jackson, M. A. "Life Cycle Concept Consideration Harmful." Comm. of ACM, Vol. 7, No. 2, (April, 1982), 29-32.

271. McCue, G. M. "IBM's Santa Teresa Lab. - Architecture Design for Program Development." IBM Systems Journal, Vol. 17, No. 1, 1978, 4-25.

272. McFarlan, F. W. "Effective EDP Project Management." Managing the Data Resource Function. Edited by R. L. Nolan. St. Paul: West Publishing Co., 1974.

273. McFarlan, F. W., et al. Information Systems Administration. New York: Holt, Rinehart and Winston, 1973.

274. McGill, M. M. "Assessing the Effectiveness of Organization Development (OD) Programs." Organization and Administrative Sciences, Vol. 17, No. 1-2, (Spring- Summer, 1976), 123-128.

275. McGowan, C. L. "Management Planning for Large Software Projects." IEEE, 1978.

276. McGowan, C. L. and McHenry, R. C. "Software Management." Research Directions in Software Technology. Cambridge, Mass: The M.I.T. Press, 1979.

277. McHenry, R. C. and Walston, C. E. "Software Life Cycle Management: Weapons Process Developer." IEEE Trans. on Software Engineering, Vol. Se-4, No.

4, (July, 1978), 334-344.

278. McKeen, J. D. "Activity Analysis: An Approach to Understand the Systems Development Process." ASAC - Conference, University of Ottawa, 1982.

279. McKeen, J. D. The Nature of Inter-Activity Relationships Within the Systems Development Cycle. Queen's University, HKingston, Ontario, Canada, Sept., 1981.

280. McKeen, J. D. "An Empirical Investigation of the Process and Product of Application System Development." Unpublished Ph.D. dissertation, University of Minnesota, 1981.

281. McKeen, J. D. "Successful Development Strategies for Business Application Systems." MIS Quarterly, Vol. 7, No. 3, (Sept., 1983).

282. McLaughlin, R. A. "That Old Bugaboo, Turnover." Datamation, (Oct., 1979), 97-101.

283. Meadows, D. H. "The Unavoidable A Priori." Elements of the Systems Dynamics Method. Edited by J. Randers. Cambridge, Mass: The M.I.T. Press, 1980.

284. Mercer, B. D. "Weapon System Software Acquisition and Support: A Theory of System Structure and Behavior." Unpublished M.S. Thesis, Air Force Institution of Technology, 1982.

285. Merwin, R. E. "Software Management: We Must Find a Way." IEEE, 1978.

286. Metzger, P. W. Managing a Programming Project. 2nd ed. Englewood Cliffs, New Jersey: Prentice-Hall, Inc., 1981.

287. Miller, J. G. "Toward a General Theory for the Behavior Sciences." The American Psychologist, Vol. 10, (Sept., 1955), 513-539.

288. Miller, J. G. "Potentials and Pitfalls of Path Analysis: A Tutorial Summary." Elsevier Scientific Publishing Co., Amsterdam, Netherlans.

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

290. Mills, H. "Top Down Programming in Large Systems." Debugging Techniques in Large Systems. Edited by R. Rustin. Englewood Cliffs, New Jersey:

Prentice-Hall, 1971.

291. Mills, H. "Update." Computer, Vol. 8, No. 1, (Jan., 1975), 9.

292. Mills, H. D. Chief Programmer Teams: Principles and Procedures. IBM Report, FSC 71-5108, IBM Fed Syst. Div. Gaitherway, MD., 1971.

293. Mills, H. D. "Software Engineering Education." Proceedings of the IEEE, Vol. 68, No. 9, (Sept., 1980).

294. Mills, H. D. "Software Development." IEEE Trans. on Software Engineering, Vol. Se-2, No. 4, (Dec., 1976).

295. Mitchell, J. R. "Observations on the Use of Seven Structured Programming Techniques." IEEE, 1980.

296. Mizuno, Y. "Software Quality Improvement." Computer, Vol. 16, No. 3, (Mar., 1983), 66-72.

297. Mohanty, S. N. "Software Cost Estimation: Present and Future." Software - Practice and Experience, Vol. 11, (1981), 103-121.

298. Moore, J. H. "A Framework for MIS Software Development Projects." MIS Quarterly, Vol. 3, No. 1, (Mar., 1979), 29-38.

299. Morecroft, J. D. W. "A Critical Review of Diagramming Tools for Conceptualizing Feedback System Models." The IEEE Conference on Cybernetics and Society, 1980.

300. Morecroft, J. D. W. "Rationality and Structure in Behavioral Models of Business Systems." The Int'l System Dynamics Conference, Chestnut Hill, Mass, (July, 1983).

301. Morecroft, J. D. W. and Abdel-Hamid, T. K. "A Generic System Dynamics Model of Software Project Management." The Int'l System Dynamics Conference, (July, 1983).

302. Morley, E. and Silver, A. "A Film Director's Approach to Managing Creativity." Harvard Business Review, (Mar-April, 1977), 59-70.

303. Myers, G. Estimating the Costs of a Programming System Development Project. Systems Development Div., Poughkeepsie Lab., IBM. (May, 1972).

304. Myers, G. J. Software Reliability: Principles and Practices. New York: John Wiley & Sons, Inc., 1976.

305. Myers, W. "Productivity as a Microeconomic principle." ComputerWorld, June, 1982.

306. Myers, G. J. "A Controlled Experiment in Program Testing and Code Walkthrough/Inspections." Comm of ACM, Vol. 21, No. 9, (Sept., 1978), 760-768.

307. Myers, W. "The Need for Software Engineering." Computer, (Feb., 1978).

308. Nay, J. N. "Choice and Allocation in Multiple Markets: A Research and Development Systems Analysis." Unpublished M.S. Thesis, M.I.T., Dept. of Electric Engineering, Cambridge, Mass, 1965.

309. Nelson, B. and Lowery, J. "Experienced Programmers or Trainees? A Productive Time/Cost Model." ComputerWorld.

310. Nelson, E. D. "Managing the Economics of Computer Programming." Proceedings ACM National Conference, 1968.

311. Nelson, E. C. "Software Reliability, Verification and Validation." Proceedings of the TRW Symposium on Reliable, Cost-Effective, Secure Software, Redondo Beach, Ca: TRW, Inc., 1974.

312. Nelson, E. A. Management Handbook for the Estimation of Computer Programmer Costs. Ad-A648 750, Systems Development Corp., (Oct., 1966).

313. Nelson, E. D. "Some Recent Contributions to Computer Programming Management." On the Management of Computer Programming. Edited by G. F. Weinwurm. Prenceton, New Jersey: Auerbach Publishing, Inc., 1970.

314. Newburn, R. M. "Measuring Productivity in Organizations with Unquantifiable End-Products." Personnel Journal, (Sept., 1972), 655-657.

315. Nicholas, J. M. "Organization Development in Systems Management." Journal of Systems Management, (Nov., 1979), 24-30.

316. Norden, P. V. "Useful Tools for Project Management." Operation Research in Research and Development. Edited by B. V. Dean. New York: John Wiley & Sons, Inc., 1963.

317. Noyes, C. J. and Parker, T. E. "Organizational Variables in Air Force Program/Project Environment." Project Management Quarterly, (June, 1982), 34-43.

318. Okada, M. "Software Development Effort Estimation Study - A Model from CAD/CAM System Development Experiences." The IEEE Computer's 6th Int'l Computer Software & Applications Conference, Chicago, (Nov., 1982).

319. Oliver, "Estimating the Cost of Software." Computer Programming Management. Edited by J. Hannan. Pennsauken, New Jersey: Auerbach Publishers, Inc., 1982.

320. Osborn, R. W. "Theories of Productivity Analysis." Datamation, (Sept., 1981), 212-216.

321. Oucho, W. G. and Maguire, M. A. "Organizational Control: Two Functions." Admins. Science Quarterly, Vol. 20, (Dec., 1975).

322. Paretta, R. L. and Clark, S. A. "Management of Software Development." Journal of Systems Management, (April, 1976).

323. Parikh, G. and Zvegintzev, N. "The World of Software Maintenance." IEEE Tutorial on Software Maintenance, IEEE, 1983.

324. Parnas, D. "Information Distribution. Aspects of Design Methodology." IFIP Congress Computer Software, 1971, 26-30.

325. Parnas, D. L. "A Technique for Software Module. Specifications with Examples." Comm of ACM, Vol. 15, No. 5, (1972), 330-336.

326. Parr, F. N. "An Alternative to the Rayleigh Curve Model for Software Development Effort." IEEE Trans. on Software Engineering, Vol. SE-6, No. 3, (May, 1980).

327. Paster, D. L. "Experience with Application of Modern Software Management Controls." 5th Int'l Conference Software Engineering. San Diego, Ca., 1981.

328. Pearson, A. W. and Gunz, H. P. "Project Groups." Groups at Work. Edited by R. Payne and C. Cooper. New York: John Wiley & Sons, Inc., 1981.

329. Perlis, A. J., et al. Software Metrics. Cambridge, Mass: The M.I.T. Press, 1981.

330. Perlis, A. J. "Software Engineering Education." Software Engineering Techniques: Report on a Conference Sponsored by the Nate Science Committee. Edited by J. N. Baxton and B. Randell. (Oct., 1969).

331. Perrow, C. "A Framework for the Comparative Analysis of Organizations." American Sociological Review, 1967.

332. Perrow, C. Organzational Analysis: A Sociological View. Belmont, Ca: Wadsworth Publishing Co., Inc., 1970.

333. Peschke, R. E. and Sherrill, M. L. "Management Cybernetics: An Application to the Development of a Conceptual Model of the Software Acquisition Management Discipline." Unpublished M.S. Thesis, Air Force Institution of Technology, Ohio, 1979.

334. Peters, L. J. "Design Practices to Effect Software Quality." Software Quality Management. Edited by J. D. Cooper and M. J. Fisher. New York: Petrocelli Books, Inc., 1979.

335. Pietrasanta, A. M. "Resource Analysis of Computer Project System Development." On the Management of Computer Programming. Edited by G. F. Weinwurm. Princeton, New Jersey: Auerbach Publishing, Inc., 1970.

336. Pietrasanta, A. M. "Managing the Economics of Computer Programming." Proceedings ACM National Conference, 1968.

337. Plotkin, S. "The Real Cost of DP Professionals." The Real Cost Work Book. Edited by S. Plotkin. Glen Mills, Pa: Stephen Plotkin, 1982.

338. Pooch, U. W. and Gehring, P. F. Jr. "Toward a Management Philosophy for Software Development." Advances in Computer Programming Management. Philadelphia, Pa: Heyden & Sons, Inc., 1980.

339. Powers, R. F. and Dickson, G. W. "MIS Project Management: Myths, Opinions, and Reality." California Management Review, Vol. XV, No. 3, (Spring, 1973), 147-156.

340. Powers, R. F. "An Empirical Investigation of Selected Hypothesis Related to the Success of Management

Information Systems Projects." Unpublished Ph.D. dissertation, University of Minnesota, 1971.

341. Presser, L. "Reversing the Priorities." Datamation.

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

343. Price, J. L. "The Effects of Turnover on the Organization." Organization and Admins. Sciences, Vol. 7, No. 1,2, (Spring-Summer, 1976).

344. Pugh, A. L. III. Dynamo Users' Manual. 5th ed. Cambridge, Mass: The M.I.T. Press, 1976.

345. Putman, L. "A General Empirical Solution to the Macro Software Sizing and Estimating Problem." IEEE Trans. on Software Engineering, 1971, 26-30.

346. Putman, L. H. and Fitzsimmons, A. "Estimating Software Costs, Part I." Datamation, (Sept, 1979).

347. Putman, L. H. and Fitzsimmons, A. "Estimating Software Costs, Part II." Datamation, (Oct., 1979).

348. Putman, L. H. and Fitzsimmons, A. "Estimating Software Costs. Part III." Datamation, (Nov., 1979).

349. Putman, L. H. "The Real Metrics of Software Development." EASCON 80, 1980, 310.

350. Quinnan, R. E. "The Management of Software Engineering Part V: Software Engineering Management Practices." IBM Systems Journal, Vol. 19, No. 4, 1980.

351. Radice, A. "Productivity Measures in Software." The Economics of Information Processing Vol. 2: Operations, Programming and Software Models. Edited by R. Goldberg and H. Lorin. New York: John Wiley & Sons, Inc., 1982.

352. Randers, J. "Guidelines for Modern Conceptualization." Elements of the System Dynamics Method. Edited by J. Randers. Cambridge, Mass: The M.I.T. Press, 1980.

353. Randers, J. ed. Elements of the System Dynamics Methods. Cambridge, Mass: The M.I.T. Press, 1980.

354. Reed, E. A. ""Time Sheet Accounting." Journal of Systems Management, (Jan., 1979), 32-35.

355. Reeves, T. K. and Woodward, J. "The Study of Managerial Control." Industrial Organization: Behavior and Control. Edited by J. Woodward. Oxford University Press, 1970.

356. Reifer, D. J. A Poor Man's Guide to Estimating Software Costs. Reifer Consultants, Inc., Torrance, Ca., (June, 1982).

357. Reifer, D. J. "The Nature of Software Management: A Primer." Tutorial: Software Management. Edited by D. J. Reifer. IEEE Computer Society, 1979.

358. Reifer, D. J. ed. Tutorial: Software Management. IEEE Computer Society, 1979.

359. Reifer, D. J. How Do I know I'M in Trouble: A Review Checklist. Reifer Consultants, Inc., Torrance, Ca., (Mar., 1982).

360. Reifer, D. J. What Software People Do: A Work Breakdown Structure. Reifer Consultant, Inc., Torrance, Ca., Feb., 1982.

361. Reifer, D. J. "Software Quality Assurance Tools and Techniques." Software Quality Management. Edited by J. D. Cooper and M. J. Fisher. New York: Petrocelli Books, Inc., 1979.

362. Reihl, J. W. "A Examination of Management Practices in the Development of Business Information Systems." An Unpublished Ph.D. dissertation, George Washington University, 1977.

363. Reynolds, "What's Wrong With Computer Programming Management?" On the Management of Computer Programming. Edited by G. F. Weinwurm. Princeton, New Jersey: Auerbach Publishing, Inc., 1970.

364. Rich, S. "The Ins and Outs of Choosing a Consultant." ComputerWorld, (July, 1982).

365. Richardson, G. P. "The Feedback Concept in American Social Science, with Implications for Systems Dynamics." Int'l System Dynamics Conference. Chestnut Hill, Mass, (July, 1983).

366. Richardson, G. P. :Sources of Rising Product Development Times." Technical Report D-3321-1, SD Group, Cambridge, Mass, M.I.T., 1982.

367. Richardson, G. P. and Pugh, G. L. III. Introduction to Systems Dynamics Modeling and DYNAMO.

Cambridge, Mass: The M.I.T. Press, 1981.

368. Richardson, G. P. Sources of Rising Product Development Times. Systems Dynamics Group, M.I.T., Cambridge, Mass, Jan., 1982.

369. Richmond, D. "No Nonsense Recruitment." Perspectives on Information Management: A Critical Selection of ComputerWorld Feature Articles. Edited by J. B. Rochester. New York: John Wiley & Sons, Inc., 1982.

370. Roberts, E. B., ed. Managerial Applications of System Dynamics. Cambridge, Mass: The M.I.T. Press, 1981.

371. Roberts, E. B. "The Dynamics of Research and Development." Published Ph.D. dissertation, M.I.T., Cambridge, Mass, 1964.

372. Roberts, E. B. "A Simple Model of R & D Project Dynamics." Managerial Applications of System Dynamics. Edited by E. B. Roberts. Cambridge, Mass: The M.I.T. Press, 1981.

373. Roberts, N. et al. Introduction to Computer Simulation: The System Dynamics Approach. Reading, Mass: Addison-Wesley Pushishing Co., 1983.

374. Rochester, J. B. ed. Perspectives on Information Management: A Critical Selection of ComputerWorld Feature Articles. New York: John Wiley & Sons, Inc., 1982.

375. Rolefson, J. F. "Project Management - 6 Critical Steps." Journal of Management Science, (April, 1978), 10-17.

376. Rubin, H. A. Macro-estimation of Software Development Parameters: The Estimacs System. Dept. of Computer Science, the City University of N.Y., New York.

377. Rullo, T. A., ed. Advances in Computer Programming Management. Philadelphia, Pa: Heyden & Sons, Inc., 1980.

378. Sampson, W. F., et al. "Organizational Strategies for Producing Better Software."

379. Sanders, J. "Barriers to Estimating DP Projects Effectivity." Infosystems, (Dec., 1980), 64-70.

380. Sawyer, S. K. "If I Flunk Out of Chemistry I May Seriously Consider the Computer Field." Perspectives on Information Management: A Critical° Selection of ComputerWorld Feature Articles. Edited by J. B. Rochester. New York: John Wiley & Sons , Inc., 1982.

381. Scacchi, W. "Managing Software Engineering Projects: A Social Analysis." University of S. Calif., L.A., CA., (Dec., 1982).

382. Schainblatt, A. H. "How Companies Measure the Productivity of Engineers and Scientists." Research Management, (May, 1982), 10-18.

383. Schein, E. H. Organizational Psychology. 3rd ed. Englewood Cliffs, New Jersey: Prentice-Hall, Inc., 1980.

384. Schindler, M. "Todays' Software Tools Point to Tomorrow's Tool Systems." Electronic Design, (July, 1981), 73-110.

385. Schneider, V. "Prediction of Software and Project Duration - Four New Formulas." SIGPLAN Notices, (June, 1978).

386. Schultz, B. "Five Factors Held Critical in Motivating DPers." ComputerWorld. (Nov., 1981).

387. Scott, R. F. and Simmons. D. B. "Predicting Programming Group Productivity - A Communications Model." IEEE Trans. on Software Engineering, Vol. SE-1, No. 4, (Dec., 1975).

388. Scott, R. F. and Simmons, B. D. "Programmer Productivity and the Delphi Technique." Datamation, (May, 1974), 71-73.

389. Selltiz, C., et al. Resaerch Methods in Social Relations. 3rd ed. New York: Holt, Rivehart & Winston, 1976.

390. Semprevivo, P. C. Teams - In Information Systems Development. New York: Yourdon, Inc., 1980.

391. Senn, J. A. "Structured Walkthroughs." Systems Development Management. Edited by J. Hannan. Pennsauken, New Jersey: Auerbach Publishers, Inc., 1982.

392. Shackleton, J. "Systems Development Methodology Packages." Systems Development Management. Edited by J. Hannan. Pennsauken, New Jersey: Auerbach

Publishers, Inc., 1982.

393. Shaw, D. E. "Managing a Software Engineering." Datamation, (Nov., 1976), 48-50.

394. Shell, R. L. "Work Measurement for Computer Programming Operations." Industrial Engineering, (Oct., 1972), 32-36.

395. Sheppard, S. B., et al. "Modern Coding Prectices and Programmer Performance." Computer, (Dec., 1977), 41-49.

396. Shneiderman, B. Software Psychology - Human Factors in Computer and Information Systems. Cambridge, Mass: Winthrop Publishers, Inc., 1980.

397. Shneiderman, B. "Group Processes in Programming." Datamation, (Jan., 1980), 138-141.

398. Shooman, M. L. Software Engineering - Design Reliability and Management. New York: McGraw_hill, Inc., 1983.

399. Shooman, M. L., et al. "Types, Distribution, and Test and Correction Times for Programming Errors." Proceedings of the Int'l Conference on Reliable Software, (April, 1975).

400. Shooman, M. L. and Natarajan, S. "Effect of Manpower Development and Bug Generation on Software Error Models." Rome Air Development Center RADC-TR-76-400, (Jan., 1977).

401. Shooman, M. L. "Tutorial on Software Cost Models." Workshop on Quality Software Models for Reliability, Complexity and Cost, (Oct., 1979).

402. Singer, L. M. "People, the Forgotten Resource." ComputerWorld.

403. Smith, W. and Jones, T. C. "Practical Productivity Improvement Through Quality Assurance. Proceedings of the 4th Annual Conference of SIM, Chicago, Ill., (Sept., 1982).

404. Snyder, T. R. "Rate Charting." Datamation, (Nov., 1976), 44-47.

405. Snyders, J. and Lasden, M. "Managing Programmers to Work Harder and Happier." Computer Decisions, (Oct., 1980), 34-47.

406. Snyders, J. "Evaluating Programmers and Analysts."

Advances in Computer Programming Management, Vol. I. Edited by T. A. Rullo. Philadelphia, Pa: Heyden & Sons, Inc., 1980.

407. Synnott, W. R. and Gruber, W. H. Information Resource Management. New York: John Wiley & Sons, Inc., 1981.

408. Social Security Administration. Productivity Measurement in Software Engineering. (June, 1983).

409. Softech, Inc. Softech's Approach to Software Development. TP113, Waltham, Mass, (July, 1979).

410. Spiro, B. E. "The Management of Structured Programming." Advances in Computer Programming Management. Vol. I. Edited by T. A. Rullo. Philadelphia, Pa: Heyden & Sons, Inc., 1980.

411. Stalnaker, A. W. and Mayer, D. B. "Selection and Evaluation of Computer Personnel." Proceedings of 23rd National Conference ACM, Brandon/Systems Press, Inc., 1968, 657-670.

412. Steele, A. C. "How to Get More Productivity From Your Staff." ComputerWorld, (July, 1982).

413. Steers, R. M. "Problems in the Measurement of Organizational Effectiveness." Admins. Science Quarterly, Vol. 20, (Dec., 1975).

414. Steffey, R. E. An Analysis of the RCA Price-S Cost Estimation Model as it Relates to Current Air Force Computer Software Acquisition and Management. Air Force Institution of Technology, Wright Pattern Air Force Base, Ohio, (Dec., 1979).

415. Steiner, I. D. "Models for Inferring Relationships Between Group Size and Potential , 273-283.

416. Steiner, I. D. Group Process and Productivity. New York: Academic Press, 1972.

417. Stephenson, W. E. "An Analysis of the Resources Used in the Safeguard System Software Development." Proceedings of the 2nd Int'l Conference on Software Engineering, 1976.

418. Sterman, J. CLass Notes, M.I.T., Cambridge, Mass, 1981.

419. Sterman, J. D. "Appropriate Summary Statistic for Evaluating the Historical Fit of SD Models." The Int'l System Dynamics Conference. Chestnut Hill, Mass, (June, 1983).

420. Stevens, B. A. "Probing the DP Psyche." Perspectives on Information Management: A Critical Selection of ComputerWorld Feature Articles. Edited by J. B. Rochester. New York: John Wiley & Sons, Inc., 1982.

421. Stevens, W. P. et al. "Structured Design." IBM Systems Journal, Vol. 13, No. 2, 1974, 115-139.

422. Strasser, S. and Deniston, O. L. "A Comparative Analysis of Goal and System Models Designed to Evaluate Health Organization Effectiveness."

423. Stringer, J. D. "Current Software Quality Management Activities." Software Quality Management. Edited by J. D. Cooper and M. J. Fisher. New York: Petrocelli Books, Inc., 1979.

424. Stroh, P. "Team Building and System Development." Datamation.

425. Tanniru, M. R., et al. "Causes of Turnover Among DP Professionals." Proceedings of the 8th Annual Computer Personnel Research Conference, Miami, Florida, (June, 1981).

426. Tausworthe, R. C. "The Work Breakdown Structure in Software Project Management." Journal of Systems and Software, 1980, 181-186.

427. Tausworthe, R. C. Standardized Development of Computer Science. Englewood Cliffs, New Jersey: Prentice-Hall, Inc., 1977.

428. Thamhain, H. J. and Wilemon, D. L. "The Effective Management of Conflict in Project-Oriented Work Environment." Defense Management Journal, (July, 1975), 29-40.

429. Thamhain, H. J. and Wilemon, D. L. "Conflict Management in Project Life Cycles." Sloan Management Review, (Spring, 1975), 31-50.

430. Tharrington, J. "Growth from Within." ComputerWorld.

431. Thayer, R. H., et al. "Validating Solutions to Major Problems in Software Engineering Project Management." Computer, (Aug., 1982).

432. Thayer, R. H. "Organizational Structures used in Software Development by the U.S. Aerospace Industry." Journal of System and Software, Vol. 1, 1980, 283-297.

433. Thayer, R. H., et al. "The Challenge of Software Engineering Project Management." Computer, (Aug., 1980).

434. Thayer, R. H., et al. "Major Issues in Software Engineering Project Management." IEEE Trans. on Software Engineering, Vol. SE-7, No. 4, (July, 1981).

235. Thayer, R. H. and Lehman, J. H. Software Engineering Project Management: A Survey Concerning U.S. Aerospace Industry Management of Software Development Projects. Sacramento Air Logistics Center, McClellan Air Force Base, Calif., (Nov., 1977).

436. Thayer, R. H. "Modeling A Software Engineering Project Management System." Unpublished Ph.D. dissertation, University of Calif., Santa Barbara, 1979.

437. Thayer, T. A., et al. Software Reliability: A Study of Large Project Reality. New York: North Holland, 1978.

438. Thibodeau, R. and Dodson, E. N. "Life Cycle Phase Interrelationships." Journal of Systems and Software, Vol. 1, 1980, 203-211.

439. Thomsett, R. People Project Management. New York: Yourdon Press, Inc., 1980.

440. Thorpe, A. J. L. "Family Programming Teams." Datamation, (Mar., 1976).

441. Tichy, N. M., et al. "Strategic Human Resource Management." Sloan Management Review, (Winter, 1982), 47-61.

442. Toellner, J. "Project Estimating." Journal of Systems Management, (May, 1977), 6-9.

443. Tripp, L. L. and Wali, P. N. How much Planning in Systems Development." Journal of Systems Management, (Oct., 1980), 6-15.

444. Tsichritzis, "Project Management." Software Engineering. Edited by F. L. Bauer. Berlin: Springer-Verlag, 1977.

445. Tsui, F. and Priven, L. "Implementation of Quality Control in Software development." National Computer Conference, 1976.

446. Turn, R., et al. "A Management Approach to the development of Computer-Based Systems." American Institute of Aeronautics & Astronautics, Inc., 1977.

447. Van de Van, A. H. and Delbecq, A. L. "A Task Contingent Model of Work-Unit Structure." Admins. Science Quarterly, Vol. 19, 1974.

448. Vaughan, A. H. "Plan for Project Success." Journal of Systems Management, (Dec., 1974).

449. Walker, M. G. Managing Software Reliability - The Paradiagmatic Approach. New York: North Holland, Inc., 1981.

450. Walston, C. E. and Felix, C. P. "Method of Programming Measurement and Estimation." IBM Journal Journal, Vol. 16, No. 1, 1977.

451. Walters, G. F. "Application of Metrics to a Software Quality Management Program." Software Quality Management. Edited by J. D. Cooper and M. J. Fisher. New York: Petrocelli Books, Inc., 1979.

452. Wasserman, A. I. "A Top-Down View of Software Engineering." Proceedings of the 1st National Conference on Software Engineering, Washington, D.C., Sept., 1975.

453. Wegner, P., ed. Research Directions in Software Technology. Cambridge, Mass: The M.I.T. Press, 1980.

454. Wegner, P. "Introduction and Overview." Research Directions in Software Technology. Edited by P. Wegner. Cambridge, Mass: The M.I.T. Press, 1980.

455. Weick, K. E. The Social Psychology of Organization. 2nd ed. Reading, Mass: Addison-Wesley Publishing Co., Inc., 1979.

456. Weil, H. B. "Industrial Dynamics & MIS." Managerial Applications of System Dynamics. Edited by E,B, Roberts. Cambridge, Mass: The M.I.T. Press, 1981.

457. Weinberg, G. M. and Schulman, E. L. "Goals and Performance in Computer Programming." Human Factors, Vol. 16, No. 1, 1974, 70-77.

458. Weinberg, G. M. "The Psychology of Improved Programmer Performance." Datamation, (Nov., 1972), 82-85.

459. Weinberg, G. M. Understanding the Professional Programmer. Boston: Little, Brown & Co., Inc., 1982.

460. Weinberg, G. M. The Psychology of Computer Programming. New York: Litton Educational Publishing, Inc., 1971.

461. Weinberg, G. M. "Overstructured Management of Software Engineering." Proceedings of the 6th Int'l Conference Software Engineering, Tokoyo, Sept., 1982.

462. Weinwurm, G. F., ed. "On the Management of Computer Programming. Princeton, New Jersey: Auerbach Publishing, Inc., 1970.

463. Weinwurm, G. F. "Introduction." On Management of Computer Programming. Princeton, New Jersey: Auerbach Publishing, Inc., 1970.

464. Weinwurm, G. F. "The Challenge to the Management Community." On Management of Computer Programming. Princeton, New Jersey: Auerbach Publishing, Inc., 1970.

465. Weiss, D. M. A Comparison of Errors in Different Software Development Environments. Report for the Naval Research Lab. Washington, D.C., (July, 1982).

466. Weiss, D. M. "Evaluating Software Development by Error Analysis." Journal of Systems and Software, Vol. 1, 1979, 57-70.

467. Weiss, H. M. "Project Control." Journal of Systems Management, (May, 1973), 14-17.

468. Wender, P. H. "Vicious and Virtuous Circles: The Role of Deviation Amplifying Feedback in the Origin and Perception of Behavior." Psychiatry, Vol. 31.

469. Wesserman, A. I. "Software Development - There's Got to be a Better Way." Software Engineering. Edited by H. Freeman, et al. New York: Academic Press, Inc., 1980.

470. Whitesides, B. A. "The Hidden Costs of In-House Development." Datamation, (Sept., 1981), 172-175.

471. Whited, J. A. "Management Control Practices for Software Quality." Software Quality Management. Edited by J. D. Cooper and M. J. Fisher. New

York: Petrocelli Books, Inc., 1979.

472. Wilemon, D. L. "Managing Conflict in Temporary Management Systems." Journal of Management Studies, (Oct., 1973).

473. Willoughby, T. C. "Computing Personnel Turnover: A Review of the Literature." Computing Personnel, Vol. 7, No. 1-2, (Autumn, 1977), 11-13.

474. Winrow, "Acquiring Entry-level Programming Management. Edited by J. Hannan. Pennsauken, New Jersey: Auerbach Publishers, Inc., 1982.

475. Wolverton, R. W. "The Cost of Developing Large-Scale Software." IEEE Trans. on Computers, June, 1974.

476. Woodgate, H. S. "Management of Large Scale Computer Program Production." National Computer Conference, 1977.

477. Woodruff, C. K. "Consideration of Selected Personality - Job Satisfaction Constructs Relevant to Project Management in DP Organizations." Proceedings of the 6th Annaul Computer Personnel Research Conference, Miami, Fl, (Aug., 1979).

478. Yates, A. "A Strategy for Computer Programmer Productivity Improvement." Proceedings of the 13th Annual Computer Personnel Research Conference, (June, 91975).

479. Youngs, E. A. "Human Errors in Programming." Int'l Journal of Man-Machine Studies, Vol, 6, 1977, 361-376.

480. Yourdon, E. Managing the System Life Cycle. A Software Development Methodology Overview. New York: Yourdon, Inc., 1982.

481. Yourdon, E. Managing the Structured Techniques. 2nd ed. New York: Yourdon, Inc., 1979.

482. Yourdon, E. Structured Walkthroughs. New York: Yourdon, Inc., 1979.

483. Yourdon, E. "The Second Structured Resolution." Software World, Vol. 12, No. 3.

484. Zelkowitz, M. V., et al. Principles of Software Engineering and Design. Englewood Cliffs, New Jersey: Prentice-Hall, Inc., 1979.

485. Zelkowitz, M. V. "Perspectives on Software

Engineering." Computing Surveys, Vol. 10, No. 2, (June, 1978).

486. Zelkowitz, M.V. "Advances in Software Engineering: Resource Estimation." Advances in Computer Programming Management. Edited by T. A. Rullo. Philadelphia, Pa: Heyden & Sons, Inc., 1980.

487. Zmud, R. W. "Management of Large Software Development Efforts." MIS Quarterly, Vol. 4, No. 2, (June, 1980), 45-56.

488. Zmud, R. W. "The Effectiveness of External Information Channels in Facilitating Innovation Within Software Development Groups." MIS Quarterly, Vol. 7, No. 2, (June, 1983), 43-58.

489. Zolnowski, J. C. and Ting, P. D. "An Insider's Survey on Software Development." Proceedings of the 6th Int'l Conference on Software Engineering, Tokoyo, (Sept., 1982).

APPENDIX:

MODEL DOCUMENTATION

```
* BASE.5 / BASE MODEL: VERSION 5
NOTE
NOTE ***************
NOTE HUMAN RESOURCE MANAGEMENT SUBSYSTEM
NOTE ***************
NOTE
L     WFNEW.K=WFNEW.J+DT*(HIRERT.JK-ASIMRT.JK-NEWTRR.J)
NOTE NEW WORKFORCE (PEOPLE)
N     WFNEW=0
R     HIRERT.KL=MAX(0,WFGAP.K/HIREDY)
NOTE HIRING RATE (PEOPLE/DAY)
C     HIREDY=40
NOTE HIRING DELAY (DAYS)
A     WFGAP.K=WFS.K-TOTWF.K
NOTE WORKFORCE GAP (PEOPLE)
A     NEWTRR.K=MIN(TRNFRT.K,WFNEW.K/DT)
NOTE NEW EMPLOYEES TRANSFER RATE OUT (PEOPLE/DAY)
A     TRNFRT.K=MAX(0,-WFGAP.K/TRNSDY)
NOTE TRANSFER RATE OF PEOPLE OUT OF PROJECT (PEOPLE/DAY)
C     TRNSDY=10
NOTE TIME DELAY TO TRANSFER PEOPLE OUT (DAYS)
R     ASIMRT.KL=WFNEW.K/ASIMDY
NOTE ASSIMILATION RATE OF NEW EMPLOYEES (PEOPLE/DAY)
C     ASIMDY=80
NOTE AVERAGE ASSIMILATION DELAY (DAYS)
A     DMPTRN.K=WFNEW.K*TRPNHR
NOTE DAILY MANPOWER FOR TRAINING (MAN-DAYS/DAY)
C     TRPNHR=0.2
NOTE NUMBER OF TRAINERS PER NEW EMPLOYEE (DIMENSIONLESS)
L     CMTRMD.K=CMTRMD.J+DT*DMPTRN.J
NOTE CUMULATIVE TRAINING MAN-DAYS
N     CMTRMD=0
L     WFEXP.K=WFEXP.J+DT*(ASIMRT.JK-EXPTRR.J-QUITRT.JK)
NOTE EXPERIENCED WORKFORCE (PEOPLE)
N     WFEXP=WFSTRT
NOTE INITIAL VALUE OF EXPERIENCED WORKFORCE LEVEL
A     EXPTRR.K=MIN(WFEXP.K/DT,TRNFRT.K-NEWTRR.K)
NOTE EXPERIENCED EMPLOYEES TRANSFER RATE (PEOPLE/DAY)
R     QUITRT.KL=WFEXP.K/AVEMPT
NOTE EXPERIENCED EMPLOYEES QUIT RATE (PEOPLE/DAY)
C     AVEMPT=673
NOTE AVERAGE EMPLOYMENT TIME (DAYS)
A     FTEXWF.K=WFEXP.K*ADMPPS
NOTE FULL-TIME-EQUIVALENT EXPERIENCED WF (MEN)
A     CELNWH.K=FTEXWF.K*MNHPXS
NOTE CEILING ON NEW HIREES (MEN)
C     MNHPXS=3
NOTE MOST NEW HIREES PER EXPERIENCED STAFF (MEN/MEN)
A     CELTWF.K=CELNWH.K+WFEXP.K
NOTE CEILING ON TOTAL WORKFORCE (PEOPLE)
A     WFS.K=MIN(CELTWF.K,WFNEED.K)
NOTE WF SOUGHT (PEOPLE)
```

```
A     TOTWF.K=WFNEW.K+WFEXP.K
NOTE  TOTAL WF LEVEL (PEOPLE)
A     FTEQWF.K=TOTWF.K*ADMPPS
NOTE  FULL TIME EQUIVALENT WF (EQUIVALENT PEOPLE)
A     FRWFEX.K=WFEXP.K/TOTWF.K
NOTE  FRACTION OF WF THAT IS EXPERIENCED (DIMENSIOLESS)
NOTE
NOTE  ****************
NOTE  SOFTWARE PRODUCTION SUSBSYSTEM
NOTE  ****************
NOTE
NOTE  (A) MANPOWER ALLOCATION SECTOR
NOTE
A     TOTDMP.K=TOTWF.K*ADMPPS
NOTE  TOTAL DAILY MANPOWER (MAN-DAYS/DAY)
C     ADMPPS=1
NOTE  AVERAGE DAILY MANPOWER PER STAFF (DAY/DAY)
L     CUMMD.K=CUMMD.J+DT*TOTDMP.J
NOTE  CUMULATIVE MAN-DAYS EXPENDED (MAN-DAYS)
N     CUMMD=.0001
A     DMPATR.K=TOTDMP.K-DMPTRN.K
NOTE  DAILY MANPOWER AVAILABLE AFTER TRAINING (MAN-DAYS/DAY)
A     AFMPQA.K=PFMPQA.K*(1+ADJQA.K)
NOTE  ACTUAL FRCATION OF MANPOWER FOR QA (DIMENSIONLESS)
N     AFMPQA=PFMPQA
C     QO=0
NOTE  QUALITY OBJECTIVE ... NORMAL QO = 0
A     PFMPQA.K=TABHL(TPFMQA,PJBAWK.K,0,1,.1)*(1+QO/100)
NOTE  PLANNED FRACTION OF MANPOWER FOR QA (DIMENSIONLESS)
T TPFMQA=.15/.15/.15/.15/.15/.15/.15/.15/.15/.15/0
A     ADJQA.K=TABHL(TADJQA,SCHPR.K,0,.5,.1)
NOTE  % ADJUSTMENT IN PFMPQA (%)
T TADJQA=0/-.025/-.15/-.35/-.475/-.5
A     DMPQA.K=MIN((AFMPQA.K*TOTDMP.K),.9*DMPATR.K)
NOTE  DAILY MANPOWER ALLOCATED FOR QA (MAN-DAYS/DAY)
L     CMQAMD.K=CMQAMD.J+DT*DMPQA.J
NOTE  CUMULATIVE QA MAN-DAYS (MAN-DAYS)
N     CMQAMD=0
A     DMPSWP.K=DMPATR.K-DMPQA.K
NOTE  DAILY MANPOWER FOR SOFTWARE PRODUCTION (MAN-DAYS/DAY)
A     DESECR.K=DTCERR.K/DESRWD
NOTE  DESIRED ERROR CORRECTION RATE (ERRORS/DAY)
N     DESECR=0
C     DESRWD=15
NOTE  DESIRED REWORK DELAY (DAYS)
A     DMPRW.K=MIN((DESECR.K*PRWMPE.K),DMPSWP.K)
NOTE  DAILY MANPOWER ALLOCATED FOR REWORK (MAN-DAYS/DAY)
N     DMPRW=0
L     PRWMPE.K=PRWMPE.J+(DT/TARMPE)(RWMPPE.J-PRWMPE.J)
NOTE  PERCEIVED REWORK MANPOWER NEEDED PER ERROR (MAN-DAYS/ERROR)
N     PRWMPE=.5
C     TARMPE=10
NOTE  TIME TO ADJUST PRWMPE (DAYS)
L     CMRWMD.K=CMRWMD.J+DT*DMPRW.J
```

```
NOTE CUMULATIVE REWORK MAN-DAYS (MAN-DAYS)
N     CMRWMD=0
A     DMPDVT.K=DMPSWP.K-DMPRW.K
NOTE DAILY MANPOWER FOR DEVELOPMENT/TESTING (MAN-DAYS/DAYS)
L     CMDVMD.K=CMDVMD.J+DT*DMPDVT.J*(1-FREFTS.K)
NOTE CUMULATVE DEVELOPMENT MAN-DAYS (MAN-DAYS)
N     CMDVMD=0
NOTE
NOTE (B) SOFTWARE DEVELOPMENT SECTOR
NOTE
R     SDVRT.KL=MIN((DMPSDV.K*SDVPRD.K),TSKPRM.K/DT)
NOTE SOFTWARE DEVELOPMENT RATE (TASKS/DAY)
N     SDVRT=0
A     DMPSDV.K=DMPDVT.K*(1-FREFTS.K)
NOTE DAILY MANPOWER FOR SOFTWARE DEVELOPMENT (MAN-DAYS/DAY)
A     FREFTS.K=TABHL(TFEFTS,TSKPRM.K/PJBSZ.K,0,.2,.04)
NOTE FRACTION OF EFFORT FOR SYSTEM TESTING (DIMENSIONLESS)
T TFEFTS=1/.5/.28/.15/.05/0
A     SDVPRD.K=POTPRD.K*MPDMCL.K
NOTE SOFTWARE DEVELOPMENT PRODUCTIVITY (TASKS/MAN-DAY)
A     POTPRD.K=ANPPRD.K*MPPTPD.K
NOTE POTENTIAL PRODUCTIVTY (TASKS/MAN-DAY)
A     ANPPRD.K=FRWFEX.K*NPWPEX+(1-FRWFEX.K)*NPWPNE
NOTE AVERAGE NOMINAL POTENTIAL PRODUCTIVITY (TASKS/MAN-DAY)
C     NPWPEX=1
NOTE NOMINAL POTENTIAL PRODUCTIVITY OF EXP EMPLOYEE (TSK/M-D)
C     NPWPNE=0.5
NOTE NOMINAL POTENTIAL PROD OF NEW EMPL. (TSK/M-D)
A     MPPTPD.K=TABHL(TMPTPD,PJBAWK.K,0,1,.1)
NOTE MULTIPLIER TO POTENTIAL PRODUCTIVITY DUE TO LEARNING (DIMENSIONLESS)
T TMPTPD=1/1.0125/1.0325/1.055/1.09/1.15/1.2/1.22/1.245/1.25/1.25
A     MPDMCL.K=AFMDPJ.K*(1-COMMOH.K)
NOTE MULTIPLIER TO PRODUCTIVITY DUE TO MOTIVATION & COMM LOSSES (DIMENSIONLESS
)
A     COMMOH.K=TABHL(TCOMOH,TOTWF.K,0,30,5)
NOTE COMMUNICATION OVERHEAD (DIMENSIONLESS)
T TCOMOH=0/.015/.06/.135/.24/.375/.54
C     NFMDPJ=.6
NOTE NOMINAL FRACTION OF A MAN-DAY ON PROJECT (DIMENSIONLESS)
L     AFMDPJ.K=AFMDPJ.J+DT*WRADJR.JK
NOTE ACTUAL FRACTION OF A MAN-DAY ON PROJECT (DIMENSIONLESS)
N     AFMDPJ=NFMDPJ
R     WRADJR.KL=(WKRTS.K-AFMDPJ.K)/WKRADY.K
NOTE WORK RATE ADJUSTMENT RATE (1/DAY)
A     WKRADY.K=NWRADY.K*EWKRTS.K
NOTE WORK RATE ADJUSTMENT DELAY (DAYS)
A     NWRADY.K=TABHL(TNWRAD,TIMERM.K,0,30,5)
NOTE NORMAL WORK RATE ADJUSTMENT DELAY (DAYS)
T TNWRAD=2/3.5/5/6.5/8/9.5/10
A     EWKRTS.K=CLIP(1,.75,WKRTS.K,AFMDPJ.K)
NOTE EFFECT OF WORK RATE SOUGHT (DIMENSIONLESS)
A     WKRTS.K=(1+PBWKRS.K)*NFMDPJ
NOTE WORK RATE SOUGHT (DIMENSIONLESS)
N     MAXMHR=1
```

```
NOTE MAXIMUM BOOST IN MAN-HOURS (DIMENSIONLESS)
A     PBWKRS.K=CLIP((MDHDL.K/(FTEQWF.K*(OVWDTH.K+.0001))),
X     (MDHDL.K/(TMDPSN.K-MDHDL.K+.0001)),PMDSHR.K,0)
NOTE % BOOST IN WORK RATE SOUGHT (%)
A     MDHDL.K=CLIP(MIN(MAXSHR.K,PMDSHR.K),-EXSABS.K,PMDSHR.K,0)*CTRLSW
NOTE MAN-DAYS THAT WILL BE HANDLED OR ABSORBED (MAN-DAYS)
C     CTRLSW=1
NOTE CONTROL SWITCH ... ALLOWS US TO TEST POLICY OF NO OVERWORK (0 OR 1)
A     EXSABS.K=MAX(0,(
X     TABHL(TEXABS,TMDPSN.K/MDRM.K,0,1,.1)*MDRM.K-TMDPSN.K))
NOTE MAN-DAY EXCESSES THAT WILL BE ABSOBED (MAN-DAYS)
T TEXABS=0/.2/.4/.55/.7/.8/.9/.95/1/1/1
A     MAXSHR.K=(OVWDTH.K*FTEQWF.K*MAXMHR)*WTOVWK.K
NOTE MAXIMUM SHORTAGE IN MAN-DAYS THAT CAN BE HANDLED (MAN-DAYS)
A     WTOVWK.K=CLIP(1,0,TIME.K,BRKDTM.K+RLXTMC.K)
NOTE WILLINGNESS TO OVERWORK (0 OR 1)
L     BRKDTM.K=MAX(BRKDTM.J,SWITCH((TIME.J+DT),0,OVWDTH.K))
NOTE TIME OF LAST EXHAUSTION BREAKDOWN
N     BRKDTM=-1
L     RLXTMC.K=RLXTMC.J*SWITCH(0,1,OVWDTH.K)+DT*
X     CLIP(1,-RLXTMC.J/DT,EXHLEV.K/MXEXHT,.1)
NOTE VARIABLE THAT CONTROLS TIME TO DE-EXHAUST
N     RLXTMC=0
A     OVWDTH.K=NOVWDT.K*MODTEX.K
NOTE OVERWORK DURATION THRESHOLD (DAYS)
A     NOVWDT.K=TABHL(TNOWDT,TIMERM.K,0,50,10)
NOTE NOMINAL OVERWORK DURATION THRESHOLD (DAYS)
T TNOWDT=0/10/20/30/40/50
A     MODTEX.K=TABHL(TMODEX,EXHLEV.K/MXEXHT,0,1,.1)
NOTE EFFECT OF EXHAUSTION ON OVERWORK DURATION THRESHOLD (DIMENSIONLESS)
T TMODEX=1/.9/.8/.7/.6/.5/.4/.3/.2/.1/0
L     EXHLEV.K=EXHLEV.J+DT*(RIEXHL.JK-RDEXHL.JK)
NOTE EXHAUSTION LEVEL (EXHAUST UNITS)
N     EXHLEV=0
R     RIEXHL.KL=TABHL(TRIXHL,(1-AFMDPJ.K)/(1-NFMDPJ),
X     -0.5,1,.1)
NOTE RATE OF INCREASE IN EXHAUSTION LEVEL (EXHAUST UNITS/DAY)
T TRIXHL=2.5/2.2/1.9/1.6/1.3/1.15/.9/.8/.7/.6/.5/.4/.3/.2/0/0
R     RDEXHL.KL=CLIP(EXHLEV.K/EXHDDY,0,0,RIEXHL.JK)
NOTE RATE OF DEPLETION IN EXHAUSTION LEVEL (EXHAUST UNITS/DAY)
C     EXHDDY=20
NOTE EXHAUSTION DEPLETION DELAY TIME (DAYS)
C     MXEXHT=50
NOTE MAXIMUM TOLERABLE EXHAUSTION (EXHAUST UNITS)
NOTE
NOTE (C) QUALITY ASSURANCE AND REWORK SECTOR
NOTE
R     QART.KL=DELAY3(SDVRT.JK,AQADLY)
NOTE FOR QA RATE (TASKS/DAY)
L     TSKWK.K=TSKWK.J+DT*(SDVRT.JK-QART.JK)
NOTE TASKS WORKED (TASKS)
N     TSKWK=0
C     AQADLY=10
NOTE AVERAGE DELAY FOR QA (DAYS)
```

```
L     CUMTQA.K=CUMTQA.J+DT*(QART.JK-TSRATE.JK)
NOTE  CUMULATIVE TASKS QA'ED (TASKS)
N     CUMTQA=0
A     ANERPT.K=MAX(PTDTER.K/(TSKWK.K+.0001),0)
NOTE  AVERAGE # OF ERRORS PER TASK (ERRORS/TASK)
A     QAMPNE.K=NQAMPE.K*(1/MPDMCL.K)*MDEFED.K
NOTE  QA MANPOWER NEEDED TO DETECT AVERAGE ERROR (MAN-DAYS/ERROR)
A     NQAMPE.K=TABHL(TNQAPE,PJBAWK.K,0,1,.1)
NOTE  NOMINAL QA MANPOWER NEEDED TO DETECT AVERAGE ERROR (MAN-DAYS/ERROR)
T TNQAPE=.4/.4/.39/.375/.35/.3/.25/.225/.21/.2/.2
A     MDEFED.K=TABHL(TMDFED,ERRDSY.K,0,10,1)
NOTE  MULTIPLIER TO DETECTION EFFORT DUE TO ERROR DENSITY (DIMENSIONLESS)
T TMDFED=50/36/26/17.5/10/4/1.75/1.2/1/1/1
A     ERRDSY.K=ANERPT.K*1000/DSIPTK
NOTE  ERROR DENSITY (ERRORS/KDSI)
A     PERDRT.K=DMPQA.K/QAMPNE.K
NOTE  POTENTIAL ERROR DETECTION RATE (ERRORS/DAY)
A     ERRDRT.K=MIN(PERDRT.K,PTDTER.K/DT)
NOTE  ERROR DETECTION RATE (ERRORS/DAY)
L     CMERD.K=CMERD.J+DT*ERRDRT.J
NOTE  CUMULATIVE ERRORS DETECTED (ERRORS)
N     CMERD=0
A     PRCTDT.K=100*CMERD.K/(CUMERG.K+.001)
NOTE  PERCENT ERRORS DETECTED (PERCENT)
A     ERRSRT.K=QART.JK*ANERPT.K
NOTE  ERROR ESCAPE RATE (ERRORS/DAY)
L     CMERES.K=CMERES.J+DT*ERRSRT.J
NOTE  CUMULATIVE ERRORS THAT ESCAPED (ERRORS)
N     CMERES=0
L     PTDTER.K=PTDTER.J+DT*(ERRGRT.JK-ERRDRT.J-ERRSRT.J)
NOTE  POTENTIALLY DETECTABLE ERRORS (ERRORS)
N     PTDTER=0
R     ERRGRT.KL=SDVRT.JK*ERRPTK.K
NOTE  ERROR GENERATION RATE (ERRORS/DAY)
A     ERRPTK.K=NERPTK.K*MERGSP.K*MERGWM.K
NOTE  ERRORS PER TASK (ERRORS/TASK)
A     NERPTK.K=NERPK.K*DSIPTK/1000
NOTE  NOMINAL # OF ERRORS COMMITTED PER TASK (ERRORS/TASK)
A     NERPK.K=TABHL(TNERPK,PJBAWK.K,0,1,.2)
NOTE  NOMINAL # OF ERRORS COMMITTED PER KDSI (ERRORS/KDSI)
T TNERPK=25/23.86/21.59/15.9/13.6/12.5
A     MERGSP.K=TABHL(TMEGSP,SCHPR.K,-.4,1,.2)
NOTE  MULTIPLIER TO ERROR GENERATION DUE TO SCHEDULE PRESSURE (DIMENSIONLESS)
T TMEGSP=.9/.94/1/1.05/1.14/1.24/1.36/1.5
A     MERGWM.K=TABHL(TMEGWM,FRWFEX.K,0,1,.2)
NOTE  MULTIPLIER TO ERROR GENERATION DUE TO WORKFORCE MIX (DIMENSIONLESS)
T TMEGWM=2/1.8/1.6/1.4/1.2/1
L     CUMERG.K=CUMERG.J+DT*ERRGRT.JK
NOTE  CUMULATIVE ERRORS GENERATED DIRECTLY DURING WORKING (ERRORS)
N     CUMERG=0
L     DTCERR.K=DTCERR.J+DT*(ERRDRT.J-RWRATE.JK)
NOTE  DETECTED ERRORS (ERRORS)
N     DTCERR=0
R     RWRATE.KL=DMPRW.K/RWMPPE.K
```

```
NOTE REWORK RATE (ERRORS/DAY)
A     RWMPPE.K=NRWMPE.K/MPDMCL.K
NOTE REWORK MANPOWER NEEDED PER ERROR (MAN-DAYS/ERROR)
A     NRWMPE.K=TABHL(TNRWME,PJBAWK.K,0,1,.2)
NOTE NOMINAL REWORK MANPOWER NEEDED PER ERROR (MAN-DAYS/ERROR)
T TNRWME=.6/.575/.5/.4/.325/.3
L     CMRWED.K=CMRWED.J+DT*RWRATE.JK
NOTE CUMULATIVE REWORKED ERRORS DURING DEVELOPMENT (ERRORS)
N     CMRWED=0
NOTE
NOTE (D) SYSTEM TESTING SECTOR
NOTE
L     UDAVER.K=UDAVER.J+DT*(AEGRT.JK+AERGRT.JK-AERRRT.JK-DCRTAE.JK)
NOTE UNDETECTED ACTIVE ERRORS (ERRORS)
N     UDAVER=0
R     AEGRT.KL=(ERRSRT.K+BDFXGR.K)*FRAERR.K
NOTE ACIVE ERRORS GENERATION RATE (ERRORS/DAY)
A     BDFXGR.K=RWRATE.JK*PBADFX
NOTE BAD FIXES GENERATE RATE (ERRORS/DAY)
C     PBADFX=.075
NOTE PERCENT BAD FIXES (FRACTION)
A     FRAERR.K=TABHL(TFRAER,PJBAWK.K,0,1,.1)
NOTE FRACTION OF ESCAPING ERRORS THAT WILL BE ACTIVE (DIMENSIONLESS)
T TFRAER=1/1/1/1/.95/.85/.5/.2/.075/0/0
R     AERGRT.KL=SDVRT.JK*SMOOTH(AERRDS.K,TSAEDS)*MAERED.K
NOTE ACTIVE ERRORS REGENERATION RATE (ERRORS/DAY)
A     MAERED.K=TABHL(TMERED,SMOOTH(AERRDS.K*1000/DSIPTK,TSAEDS),0,100,10)
NOTE MULTIPLIER TO ACTIVE ERROR REGENERATION DUE TO ERROR DENSITY (DIMENSIONLE
SS)
T TMERED=1/1.1/1.2/1.325/1.45/1.6/2/2.5/3.25/4.35/6
C     TSAEDS=40
NOTE TIME TO SMOOTH ACTIVE ERROR DENSITY (AERRDS)   (DAYS)
A     AERRDS.K=UDAVER.K/(CUMTQA.K+.1)
NOTE ACTIVE ERROR DENSITY (ERRORS/TASK)
R     AERRRT.KL=UDAVER.K*AERRFR.K
NOTE ACTIVE ERRORS RETIRING RATE (ERRORS/DAY)
A     AERRFR.K=TABHL(TERMFR,PJBAWK.K,0,1,.1)
NOTE ACTIVE ERRORS RETIRING FRACTION (1/DAYS)
T TERMFR=0/0/0/0/.01/.02/.03/.04/.1/.3/1
R     DCRTAE.KL=MIN(TSRATE.JK*AERRDS.K,UDAVER.K/DT)
NOTE DETECTION/CORRECTION RATE OF ACTIVE ERRORS (ERRORS/DAY)
L     UDPVER.K=UDPVER.J+DT*(PEGRT.JK+AERRRT.JK-DCRTPE.JK)
NOTE UNDETECTED PASSIVE ERRORS (ERRORS)
N     UDPVER=0
R     PEGRT.KL=(ERRSRT.K+BDFXGR.K)*(1-FRAERR.K)
NOTE PASSIVE ERRORS GENERATION RATE (ERRORS/DAY)
R     DCRTPE.KL=MIN(TSRATE.JK*PERRDS.K,UDPVER.K/DT)
NOTE DETECT/CORRECT RATE OF PASSIVE ERRORS (ERRORS/DAY)
L     CMRWET.K=CMRWET.J+DT*(DCRTPE.JK+DCRTAE.JK)
NOTE CUMULATIVE ERRORS REWORKED IN TESTING PHASE (ERRORS)
N     CMRWET=0
A     ALESER.K=UDAVER.K+UDPVER.K+CMRWET.K
NOTE ALL ERRORS THAT ESCAPED AND WERE GENERATED (ERRORS)
A     DMPTST.K=DMPDVT.K*FREFTS.K
```

```
NOTE DAILY MANPOWER FOR TESTING (MAN-DAYS/DAY)
L    CMTSMD.K=CMTSMD.J+DT*DMPTST.J
NOTE CUMULATIVE TESTING MAN-DAYS (MAN-DAYS)
N    CMTSMD=0
R    TSRATE.KL=MIN(CUMTQA.K/DT,DMPTST.K/TMPNPT.K)
NOTE TESTING RATE (TASKS/DAY)
A    TMPNPT.K=(TSTOVH*DSIPTK/1000+TMPNPE.K*(PERRDS.K+AERRDS.K)
X    )/MPDMCL.K
NOTE TESTING MANPOWER NEEDED PER TASK (MAN-DAYS/TASK)
C    TSTOVH=1
NOTE TESTING EFFORT OVERHEAD (MAN-DAYS/KDSI)
C    TMPNPE=.15
NOTE TESTING MANPOWER NEEDED PER ERROR (MAN-DAY/ERROR)
A    PTKTST.K=CUMTKT.K/PJBSZ.K
NOTE % OF TASKS TESTED (%)
A    PERRDS.K=UDPVER.K/(CUMTQA.K+.0001)
NOTE PASSIVE ERROR DENSITY (ERRORS/TASK)
L    CUMTKT.K=CUMTKT.J+DT*TSRATE.JK
NOTE CUMULATIVE TASKS TESTED (TASKS)
N    CUMTKT=0
A    ALLERR.K=PTDTER.K+DTCERR.K+CMRWED.K+UDAVER.K+
X    UDPVER.K+CMRWET.K
NOTE ALL ERRORS (ERRORS)
A    ALLRWK.K=CMRWED.K+CMRWET.K
NOTE ALL ERRORS REWORKED ... IN DEVELOPMENT AND TESTING (ERRORS)
NOTE
NOTE ****************
NOTE CONTROL SUBSYSTEM
NOTE ****************
NOTE
L    CMTKDV.K=CMTKDV.J+DT*SDVRT.JK
NOTE CUMULATIVE TASKS DEVELOPED (TASKS)
N    CMTKDV=0
A    PJBAWK.K=CMTKDV.K/RJBSZ
NOTE % OF JOB ACTUALLY WORKED (%)
A    PJDPRD.K=TSKPRM.K/(MDPRNT.K+.1)
NOTE PROJECTED DEVELOPMENT PRODUCTIVITY (TASKS/MAN-DAY)
A    MDPRNT.K=MAX(0,MDRM.K-MDPNRW.K-MDPNTS.K)
NOTE MAN DAYS PERCEIVED REMAINING FOR NEW TASKS (MAN-DAYS)
A    MDPNRW.K=DTCERR.K*PRWMPE.K
NOTE MAN DAYS PERCEIVED NEEDED FOR REWORKING ALREADY DETECTED ERRORS (MD)
A    ASSPRD.K=PJDPRD.K*WTPJDP.K+PRDPRD.K*(1-WTPJDP.K)
NOTE ASSUMED PRODUCTIVITY (TASKS/MAN-DAY)
A    PRDPRD.K=CMTKDV.K/(CUMMD.K-CMTSMD.K)
NOTE PERCEIVED DEVELOPMENT PRODUCTIVITY (TASKS/MAN-DAY)
A    WTPJDP.K=MPWDEV.K*MPWREX.K
NOTE WEIGHT TO PROJECTED DEVELOPMENT PRODUCTIVITY (DIMENSIONLESS)
A    MPWDEV.K=TABHL(TMPDEV,PJBPWK.K/100,0,1,.1)
NOTE MULTIPLIER TO PRODUCTIVITY WEIGHT DUE TO DEVELOMENT (DIMENSIONLESS)
T TMPDEV=1/1/1/1/1/1/.975/.9/.75/.5/0
A    MPWREX.K=TABHL(TMPREX,(1-MDPRNT.K/(JBSZMD.K-TSSZMD.K)),
X    0,1,.1)
NOTE MULTIPLIER TO PRODUCTIVITY WEIGHT DUE TO RESOURCE EXPENDITURE (DIMENSIONL
ESS)
```

```
T TMPREX=1/1/1/1/1/1/.975/.9/.75/.5/0
A     MDPNNT.K=TSKPRM.K/ASSPRD.K
NOTE MAN DAYS PERCEIVED STILL NEEDED FOR NEW TASKS (MAN-DAYS)
A     TMDPSN.K=MDPNNT.K+MDPNTS.K+MDPNRW.K
NOTE TOTAL MAN DAYS PERCEIVED STILL NEEDED (MAN-DAYS)
A     MDPNTS.K=TSTPRM.K/PRTPRD.K
NOTE MAN DAYS PERCEIVED STILL NEEDED FOR TESTING (MAN-DAYS)
A     TSTPRM.K=PJBSZ.K-CUMTKT.K
NOTE TASKS REMAINING TO BE TESTED (TASKS)
A     PRTPRD.K=SMOOTH((CLIP(PLTSPD.K,ACTSPD.K,0,CUMTKT.K)),TSTSPD)
NOTE PERCEIVED TESTING PRODUCTIVITY (TASKS/MAN-DAY)
C     TSTSPD=50
NOTE TIME TO SMOOTH TESTING PRODUCTIVITY (DAYS)
A     PLTSPD.K=PJBSZ.K/TSSZMD.K
NOTE PLANNED TESTING PRODUCTIVITY (TASKS/MAN-DAY)
A     ACTSPD.K=CUMTKT.K/(CMTSMD.K+.001)
NOTE ACTUAL TESTING PRODUCTIVITY (TASKS/MAN-DAY)
A     PMDSHR.K=TMDPSN.K-MDRM.K
NOTE PERCEIVED SHORTAGE IN MAN DAYS (MAN-DAYS)
A     SHRRPT.K=PMDSHR.K-MDHDL.K
NOTE SHORTAGE REPORTED (MAN-DAYS)
A     MDRPTN.K=MDRM.K+SHRRPT.K
NOTE MAN DAYS REPORTED STILL NEEDED (MAN-DAYS)
A     SCHPR.K=(TMDPSN.K-MDRM.K)/MDRM.K
NOTE SCHEDULE PRESSURE     (DIMENSIONLESS)
A     PTRPTC.K=SMOOTH((100-(MDRPTN.K/JBSZMD.K)*100),RPTDLY)
NOTE % OF TASKS REPORTED COMPLETE (%)
N     PTRPTC=0
C     RPTDLY=10
NOTE REPORTING DELAY (DAYS)
A     PDEVRC.K=SMOOTH(MAX((100-((MDRPTN.K-MDPNTS.K)/(JBSZMD.K-TSSZMD.K)
X     )*100),PDEVRC.K),RPTDLY)
N     PDEVRC=0
NOTE % DEVELOPMENT PERCEIVED COMPLETE %
L     UNDJTK.K=UNDJTK.J-DT*RTDSTK.JK
NOTE UNDISCOVERED JOB TASKS (TASKS)
N     UNDJTK=RJBSZ-PJBSZ
N     RJBSZ=RJBDSI/DSIPTK
NOTE REAL JOB SIZE IN TASKS (TASKS)
R     RTDSTK.KL=UNDJTK.K*PUTDPD.K/100
NOTE RATE OF DISCOVERING TASKS (TASKS/DAY)
A     PUTDPD.K=TABHL(TPUTDD,PJBPWK.K,0,100,20)
NOTE PERCENT OF UNDISCOVERED TASKS DISCOVERED PER DAY (1/DAY)
T TPUTDD=0/0.4/2.5/5/10/100
A     PJBPWK.K=(CMTKDV.K/PJBSZ.K)*100
NOTE % OF JOB PERCEIVED WORKED (%)
R     RTINCT.KL=DELAY3(RTDSTK.JK,DLINCT)
NOTE RATE OF INCORPORATING DISCOVERED TASKS INTO PROJECT (TASKS/DAY)
L     TKDSCV.K=MAX((TKDSCV.J+DT*(RTDSTK.JK-RTINCT.JK)),0)
NOTE TASKS DISCOVERED (TASKS)
N     TKDSCV=0
C     DLINCT=10
NOTE AVERAGE DELAY IN INCORPORATING DISCOVERED TASKS (DAYS)
L     PJBSZ.K=PJBSZ.J+DT*RTINCT.JK
```

```
NOTE CURRENTLY PERCEIVED JOB SIZE (TASKS)
N    PJBSZ=PJBDSI/DSIPTK
A    TSKPRM.K=PJBSZ.K-CMTKDV.K
NOTE NEW TASKS PERCEIVED REMAINING (TASKS) .
A    PSZDCT.K=TKDSCV.K/ASSPRD.K
NOTE PERCEIVED SIZE OF DISCOVERED TASKS IN MAN DAYS (MAN-DAYS)
A    RSZDCT.K=PSZDCT.K/(MDPRNT.K+.0001)
NOTE RELATIVE SIZE OF DISCOVERED TASKS (DIMENSIONLESS)
A    FADHWO.K=TABHL(TFAHWO,RSZDCT.K/(MSZTWO+.001),0,2,.2)
NOTE FRACTION OF ADDITIONAL TASKS ADDING TO MAN-DAYS
T TFAHWO=0/0/0/0/0/0/.7/.9/.975/1/1
C    MSZTWO=.01
NOTE MAXIMUM RELATIVE SIZE OF ADDITIONS TOLERATED W/O ADDING TO PROJECT'S MAN-
DAYS
R    IRDVDT.KL=(RTINCT.JK/ASSPRD.K)*(FADHWO.K)
NOTE RATE OF INCREASE IN DEVELOPMENT MAN-DAYS DUE TO DISCOVERED TASKS (MD/D)
L    TSSZMD.K=TSSZMD.J+DT*IRTSDT.JK+ARTJBM.K*CLIP(1,0,FREFTS.J,.9)
NOTE PLANNED TESTING SIZE IN MAN-DAYS ... BEFORE WE START TESTING
N    TSSZMD=TSTMD
R    IRTSDT.KL=(RTINCT.JK/PRTPRD.K)*(FADHWO.K)
NOTE RATE OF INCREASE IN TESTING MAN DAYS DUE TO DISCOVERED TASKS (MD/D)
L    JBSZMD.K=JBSZMD.J+DT*(IRDVDT.JK+IRTSDT.JK+ARTJBM.JK)
NOTE TOTAL JOB SIZE IN MAN DAYA (MAN-DAYS)
N    JBSZMD=DEVMD+TSTMD
R    ARTJBM.KL=(MDRPTN.K+CUMMD.K-JBSZMD.K)/DAJBMD.K
NOTE RATE OF ADJUSTING THE JOB SIZE IN MAN-DAYS (MAN-DAYS/DAY)
A    DAJBMD.K=TABHL(TDAJMD,TIMERM.K,0,20,20)
NOTE DELAY IN ADJUSTING JOB'S SIZE IN MAN DAYS (DAYS)
T TDAJMD=.5/3
A    MDRM.K=MAX(.0001,JBSZMD.K-CUMMD.K)
NOTE
NOTE ***************
NOTE PLANNING SUBSYSTEM
NOTE
NOTE ***************
NOTE
NOTE MAN DAYS REMAINING
A    TIMEPR.K=MDRM.K/(WFS.K*ADMPPS)
NOTE TIME PERCEIVED STILL REQUIRED (DAYS)
A    INDCDT.K=TIME.K+TIMEPR.K
NOTE INDICATED COMPLETION DATE
L    SCHCDT.K=SCHCDT.J+DT*(INDCDT.J-SCHCDT.J)/SCHADT.K
NOTE SCHEDULE COMPLETION DATE
N    SCHCDT=TDEV
A    SCHADT.K=TABHL(TSHADT,TIMERM.K,0,5,5)
NOTE SCHEDULE ADJUSTMENT TIME (DAYS)
T TSHADT=.5/5
A    TIMERM.K=MAX(SCHCDT.K-TIME.K,0)
NOTE TIME REMAINING (DAYS)
A    WFINDC.K=(MDRM.K/(TIMERM.K+.001))/ADMPPS
NOTE INDICATED WORKFORCE (PEOPLE)
A    WFNEED.K=MIN((WCWF.K*WFINDC.K+(1-WCWF.K)*TOTWF.K),WFINDC.K)
NOTE WORKFORCE LEVEL NEEDED (PEOPLE)
A    WCWF.K=MAX(WCWF1.K,WCWF2.K)
```

```
NOTE WILLINGNESS TO CHANGE WORKFORCE LEVEL (DIMENSIONLESS)
A     WCWF1.K=TABHL(TWCWF1,TIMERM.K/(HIREDY+ASIMDY),0,3,.3)
NOTE WILLINGNESS TO CHANGE WORKFORCE (1) (DIMENSIONLESS)
T TWCWF1=0/0/.1/.4/.85/1/1/1/1/1/1
A     WCWF2.K=TABHL(TWCWF2,SCHCDT.K/MXTLCD,.86,1,.02)
NOTE WILLINGNESS TO CHANGE WF (2) (DIMENSIONLESS)
T TWCWF2=0/.1/.2/.35/.6/.7/.77/.80
N     MXTLCD=MXSCDX*TDEV
NOTE MAXIMUM TOLERABLE COMLETION DATE (DAYS)
C     MXSCDX=1E6
NOTE MAX SCHEDULE COMPLETION DATE EXTENSION (DIMENSIONLESS)
NOTE
NOTE ****************
NOTE INITIALIZATION
NOTE ****************
NOTE
NOTE THE REAL JOB SIZE = 64,000 DSI
NOTE FROM BOEHM PAGE 90:
NOTE DISTRIBUTION OF EFFORT BY PHASE IS:
NOTE DESIGN (39%), PROGRAMMING (36%), INT TESTING (25%)
NOTE FROM BOEHM PAGE 64-65:
NOTE EFFORT = 2.4*(KDSI)**1.05
NOTE        = 190 MM
NOTE        = 190 * 19 = 3592 MAN-DAYS
NOTE DEVELOPMENT EFFORT = 75 %
NOTE                    = 2695 MAN DAYS
NOTE GROSS DEV PRODUCTIVITY = 64,000/2695 = 24 DSI/MD
NOTE
NOTE SCHEDULE = 2.5 * (MM)**.38
NOTE          = 18 MONTHS
NOTE          = 348 DAYS
NOTE
NOTE AVERAGE STAFF SIZE = 3592/348
NOTE                    = 10
NOTE
NOTE GROSS PRODUCTIVITY INCORPORATES: DEV, FOR QA, & REWORKING
NOTE ASSUMING 25% OF EFFORT GOES INTO QA & REWORKING
NOTE 25% OF 2695 MAN DAYS = 674 MAN DAYS
NOTE DEVELOPMENT PRODUCTIVITY = 64,000/(2695-674)
NOTE                          = 31 DSI/MAN-DAY
NOTE
NOTE ASSUME LOSSES IN PRODUCTIVITY = 50 %
NOTE THEREFORE POTENTIAL PRODUCTIVITY = 31 * 2 = APPROX 60 DSI/MD
NOTE DEFINE 1 TASK = 60 DSI
C     DSIPTK=60
NOTE DSI PER TASK
C     RJBDSI=64000
NOTE REAL JOB SIZE IN DSI
C     UNDEST=0
NOTE TASKS UNDERESTIMATION FRACTION (FRACTION)
N     PJBDSI=RJBDSI*(1-UNDEST)
NOTE PERCEIVED JOB SIZE IN DSI
N     TOTMD=MDSWCH*(((2.4*EXP(1.05*LOGN(PJBDSI/1000)))*19)*(1-UNDESM))
X     +(1-MDSWCH)*TOTMD1
```

```
NOTE TOTAL MAN DAYS
C    UNDESM=0
NOTE MAN-DAYS UNDERESTIMATION FRACTION (FRACTION)
N    DEVMD=DEVPRT*TOTMD
NOTE DEVELOPMENT MAN DAYS
C    MDSWCH=1
NOTE SWITCH 0 OR 1
C    TOTMD1=0
NOTE TOTAL MANDAYS
C    DEVPRT=0.80
NOTE % OF EFFORT ASSUMED NEEDED FOR DEVELOPMENT
N    TSTMD=(1-DEVPRT)*TOTMD
NOTE TESTING MAN DAYS
N    WFSTRT=TEAMSZ*INUDST
NOTE TEAM SIZE AT BEGINNING OF DESIGN (MEN)
C    INUDST=.5
NOTE INITIAL UNDERSTAFFING FACTOR (DIMENSIONLESS)
N    TDEV=SCSWCH*((19*2.5*EXP(0.38*LOGN(TOTMD/19)))*SCHCOM)
X    +(1-SCSWCH)*TDEV1
NOTE TOTAL DEVELOPMENT TIME (DAYS)
C    SCHCOM=1
NOTE SCHEDULE COMPRESSION FACTOR (DIMENSIONLESS)
C    SCSWCH=1
NOTE SWITCH 0 OR 1
C    TDEV1=0
NOTE TIME TO DEVELOP
N    TEAMSZ=(TOTMD/TDEV)/ADMPPS
NOTE
NOTE ****************
NOTE VII. CONTROL STATEMENTS
NOTE ****************
NOTE
SPEC DT=.5,MAXLEN=1000,PLTPER=10
A    LENGTH.K=CLIP(TIME.K,MAXLEN,PTKTST.K,.99)
A    PRTPER.K=LENGTH.K
PRINT TOTMD,DEVMD,TSTMD,TDEV
PRINT TOTWF,CUMMD,CMQAMD,CMRWMD,CMTSMD,CUMERG,CMERES,CMRWET,PRCTDT
PLOT TOTWF=W(0,20)
PLOT PDEVRC=1(0,100)
PLOT PJBSZ=J,CMTKDV=1,CUMTKT=T(0,1500)/CUMMD=C,JBSZMD=D(0,
X    5000)/SCHCDT=S(200,600)/PTRPTC=R,PDEVRC=V(0,100)
PLOT AFMDPJ=F(0,2.4)
PLOT CUMERG=G,CMERD=D,CMERES=S(0,4000)/PRCTDT=P(0,100)
PLOT AFMDPJ=F(0,2.4)/EXHLEV=X,OVWDTH=V(0,100)/MDHDL=H,PMDSHR=P
X    (-500,500)/SHRRPT=1(-500,500)/SCHPR=S(-1,1)/
X    JBSZMD=D,CUMMD=C(0,5000)/SHRRPT=1(-200,200)
```