

PROGRAM EVOLUTION

Processes of Software Change

All those involved in the design, implementation and use of computer-based systems will be aware of the continuous need for fault repair, enhancement and adaptation to changes in the computer's operational environment, or to unforeseen circumstances. They will recognize the truth in the statement: there is no such thing as a 'finished' computer program.

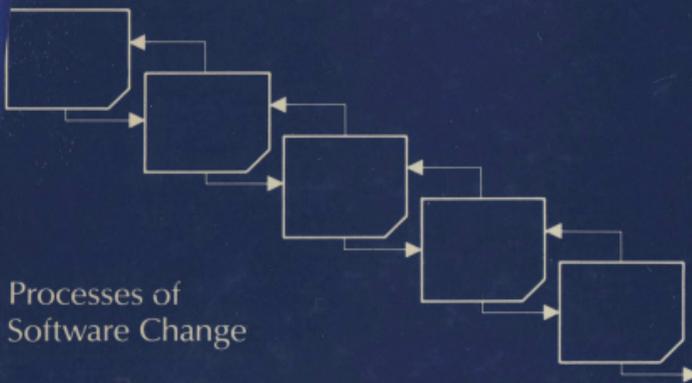
The authors were the first to recognize the phenomenon of program evolution: they pioneered its study, and the development of a software engineering discipline and a coherent programming process. In this book they bring together for the first time papers written by themselves and colleagues that provide complete and accessible coverage of the subject.

This book is required reading for all those with executive management or technical involvement in software development or computer usage and is strongly recommended for all those who are concerned with the effective—and beneficial—exploitation of information technology.

M. M. Lehman is Emeritus Professor of Computing Science at Imperial College of Science and Technology, London, UK and chairman of Imperial Software Technology.

L. A. Belady is Vice President, Software Technology, MCC, Austin, Texas and was previously at the T. J. Watson Research Center, IBM.

Lehman PROGRAM EVOLUTION



Processes of Software Change

M. M. Lehman and L. A. Belady

Academic Press

Harcourt Brace Jovanovich, Publishers

LONDON ORLANDO SAN DIEGO NEW YORK
AUSTIN MONTREAL SYDNEY TOKYO TORONTO

Academic Press Inc. (London) Ltd.

24-28 Oval Wood, London NW1 7DA, England

Academic Press, Inc., Orlando, Florida 32887

Academic Press Canada

55 Barber Greene Road, Don Mills, Ontario, Canada M3C 2A1

Academic Press Australia

P.O. Box 300, North Ryde, N.S.W. 2113, Australia

Academic Press Japan, Inc.

Hokaku Building, 3-11-13, Kitabashi, Chiyoda-ku, Tokyo 102, Japan

17th

22674

2007-1-50

Evolution: Processes of

APIC Studies in Data Processing No. 27

Program Evolution

Processes of Software Change

Academic Press Rapid Manuscript Reproduction

*This is volume 27 in A.P.I.C. Studies in Data Processing
General Editors: Fraser Duncan and M. J. R. Shave
A list of recent titles in this series appears at the end of this volume*

Program Evolution

Processes of Software Change

Edited by

M. M. Lehman

*Department of Computing
Imperial College of Science and Technology
London, England*

L. A. Belady

*Software Technology
MCC
Austin, Texas, USA*

1985



ACADEMIC PRESS

Harcourt Brace Jovanovich, Publishers
London Orlando San Diego New York
Austin Montreal Sydney Tokyo Toronto

COPYRIGHT © 1985 BY ACADEMIC PRESS INC. (LONDON) LTD.
ALL RIGHTS RESERVED.
NO PART OF THIS PUBLICATION MAY BE REPRODUCED OR
TRANSMITTED IN ANY FORM OR BY ANY MEANS, ELECTRONIC
OR MECHANICAL, INCLUDING PHOTOCOPY, RECORDING, OR
ANY INFORMATION STORAGE AND RETRIEVAL SYSTEM, WITHOUT
PERMISSION IN WRITING FROM THE PUBLISHER.

ACADEMIC PRESS INC. (LONDON) LTD.
24-28 Oval Road
LONDON NW1 7DX

United States Edition published by
ACADEMIC PRESS, INC.
Orlando, Florida 32887

BRITISH LIBRARY CATALOGUING IN PUBLICATION DATA

Program evolution : processes of software change.—

(APIC studies in data processing)

1. Programming (Electronic computers)—History

I. Lehman, M.M. II. Belady, L.A.

001.64'2 QA76.6

ISBN 0-12-442440-6

LIBRARY OF CONGRESS CATALOGING-IN-PUBLICATION DATA

Main entry under title:

Program evolution.

(APIC studies in data processing)

Includes index.

1. Electronic digital computers—Programming—

Addresses, essays, lectures. 2. Software maintenance—

Addresses, essays, lectures. I. Lehman, M. M.

II. Belady, L. A. (Laszlo A.) Date

III. Series: A.P.I.C. studies in data processing.

QA76.6.P751175 1985 005.1 85-48036

ISBN 0-12-442440-6 (alk. paper)

ISBN 0-12-442441-4 (paperback)

PRINTED IN THE UNITED STATES OF AMERICA

We dedicate this book to our wives Chava and Gizella and our children Machla, Benny, Jonathan, Rafi, Christian, Esti and Petra. Without their continuing interest and support, their patient forbearing of our involvement in our work and our continuous travel, we could never have undertaken the work or achieved the results that have made this work possible and, we hope, worthwhile.

Contents

<i>Preface</i>	xi
1. Introduction	1
2. Program Evolution	9
<i>M. M. Lehman</i>	
3. The Programming Process	39
<i>M. M. Lehman</i>	
4. Natural Selection as Applied to Computers and Programs	85
<i>G. M. Weinberg</i>	
5. Programming System Dynamics or The Metadynamics of Systems in Maintenance and Growth	99
<i>L. A. Belady and M. M. Lehman</i>	
6. An Introduction to Growth Dynamics	123
<i>L. A. Belady and M. M. Lehman</i>	
7. Programs, Cities, Students—Limits to Growth	133
<i>M. M. Lehman</i>	
8. A Model of Large Program Development	165
<i>L. A. Belady and M. M. Lehman</i>	

9. Program Evolution and its Impact on Software Engineering <i>M. M. Lehman and F. N. Parr</i>	201
10. Evolving Parts and Relations—A Model of System Families <i>L. A. Belady and P. M. Merlin</i>	221
11. Human Thought and Action as an Ingredient of System Behaviour <i>M. M. Lehman</i>	237
12. Laws of Program Evolution—Rules and Tools for Programming Management <i>M. M. Lehman</i>	247
13. Staffing Problems in Large Scale Programming <i>L. A. Belady</i>	275
14. The Characteristics of Large Systems <i>L. A. Belady and M. M. Lehman</i>	289
15. On Software Complexity <i>L. A. Belady</i>	331
16. A Mathematical Model for the Evolution of Software <i>C. M. Woodside</i>	339
17. Modifiability of Large Software Systems <i>L. A. Belady</i>	355
18. On Understanding Laws, Evolution and Conservation in the Large-Program Life Cycle <i>M. M. Lehman</i>	375
19. Programs, Life Cycles and Laws of Software Evolution <i>M. M. Lehman</i>	393

20. The Environment of Program Development and Maintenance Programs, Programming and Programming Support	451
<i>M. M. Lehman</i>	
21. Programming Productivity—A Life Cycle Concept	469
<i>M. M. Lehman</i>	
22. The Role of Systems and Software Technology in the Fifth Generation	491
<i>M. M. Lehman</i>	
<i>References</i>	501
<i>Index</i>	523

Preface

Laymen and professionals alike tend to perceive programs as 'mechanisms' that are conceived, designed and then simply constructed, that is, 'written', to solve some problem or to implement an application on a digital computer. It is generally accepted that the program as first visualised and eventually written will not be error free, that it will have to undergo a debugging process before entering serious service. But once bug-free it should be available forever to fulfill its purpose.

The facts are somewhat different. For one thing, with current programming methods there generally does not exist any way in which a program can be shown, known or made to be fault-free. Bugs will continue to surface as long as a program continues to be used, and some of these, at least, will have been in the program from its inception. Moreover, it has been the universal experience that quite apart from the discovery, during commissioning and afterwards, of errors in design or implementation, computer users come up with a continuing demand for performance improvements, functional enhancement and new capabilities. This occurs both before the program has been installed and after it has entered into service. The consequent continuing program maintenance (as it has come to be known) typically absorbs as much as 70% of the total activity expended on the program during its initial development and subsequent service life.

For a long time it was thought that the occurrence of such never-ending maintenance activity was mainly due to lack of foresight on the part of planners, designers, programmers and managers. Factors such as rapidly advancing technology, the relative ineffectiveness of the software engineering process and the demands of an ever widening market place were, however, also recognised.

None of these factors is, however, the prime cause of the problem. The work of the editors of this book and of other people since the late 1970s has shown that all these factors contributed to the development of a 'Software Crisis', the universal experience that software systems are rarely completed on time, contain a seemingly inexhaustible stock of faults and are excessively costly to create and maintain. It is, however, now recognised that the problems stem from a more fundamental source. Evolution is intrinsic to the very nature of computer usage and of the associated programs (LEH80), that is, programs that are used and

that exceed some minimal capability. As a consequence, programs must be continuously adapted. They evolve in a manner that is reminiscent of the evolution of biological organisms and of social groupings (LEH80, 82b).

The evolutionary pressures on programs arise in several different ways. Evolution first appears during the development process, the human activity, that transforms a computer application concept into an operational system. The concepts, algorithms and techniques that are to be used to implement the program evolve as the design proceeds, as insight into the problem to be solved and understanding of methods for its solution are gradually developed. Evolution is also present in the continuing process that maintains system and cost-effectiveness, adapting it to the needs of a changing environment through the periodic release of modified versions of the code and documentation. Finally, the entire system evolves. It is re-defined, re-designed, re-implemented and replaced as it becomes too complex to maintain, or out of step with evolving application needs and implementation technologies. It is continually being adapted to the continuously changing environment.

By reproducing under one cover some of the key publications in this field as produced over their gestation period, this book traces the gradual evolution of the ideas and insights summarised above and of associated technologies. This historical approach will facilitate the achievement of real understanding of the concepts and issues that are revealed. One does not often have the opportunity to document the history of a technology whilst it is in the making. This book seeks to achieve this.

The book, however, is not aimed primarily at the historian. The collection is important for all who develop or use software. It should also be of interest to the general reader, who, as he follows the book, will be able to achieve the understanding reached by the editors over an extended period as they developed the subject.

The articles constituting this book have mostly been available for a number of years. We hope, nevertheless, that their collection in a single volume will encourage research and development in an area of computing science that is likely to prove of increasing importance as mankind becomes more and more dependant on correct and up-to-date software for operation and survival. As the world relies ever more on computers it becomes vital that all those concerned with computer-based systems, their design, construction, operation, exploitation or management, fully understand the issues raised, the dangers that arise from failing to take appropriate action and the opportunities offered by software engineering. We hope that the readers of this volume will achieve these insights, discover at least some of the answers and follow up appropriate pointers to fundamental topics in this emerging discipline.

Issues discussed are presented almost exclusively in the context of software. They are, however, also likely to prove relevant, following change of terminology, to other artificial systems—even, with different time scale, to biological, social or economic systems. Such wider significance must be further investigated.

Increasing societal dependence on computers and the need for the software that controls them to evolve in response to environmental changes and opportunities make extension and generalisation of these concepts an urgent necessity.

In assembling these articles we have restricted harmonisation of content and style to a unification of the reference listings originally printed at the end of each. We have not sought to eliminate repetition, avoid redundancy or to remove inconsistencies or contradiction between papers written at quite different times. To have done so would have obscured the progress in our underlying understanding and concept formation. At most we have added the occasional footnote to draw the reader's attention to some significant change. Such editorial comments, made in 1984 and 1985, are distinguished by '(eds)' from original footnotes, indicated by '(orig)'.

The final comment on our editorial policy relates to the acknowledgement section at the end of each article. We have been associated with many colleagues over the years as the subject matter has developed. Their individual and collective contributions and their support have contributed significantly to the progress made. They continue to deserve our grateful acknowledgement. The sections have therefore been retained intact. It is appropriate at this point to add our grateful and sincere thanks to Mrs. Jane Spurr for her patient typing, willing amendment, layout planning and re-creation of plots and diagrams to provide uniformity for the manuscript.

This book provides an historical record of developing insight and understanding. As insight and understanding increase, viewpoints and interpretations change. The reader who follows the same intellectual evolution, albeit in much less time than it has taken us, will surely benefit from the experience.

May we be permitted one final introductory remark. Preparation of this book has caused us to re-read articles not looked at for many years. We cannot refrain from expressing our surprise in finding so much of the material at least as relevant today as it was at the time of writing. How little has changed since each of the articles was first published. There clearly has been progress. The informed reader will note that, for example, there is no reference to notions of verification and program proving in the earlier articles. The emergence and wide acceptance of such concepts exemplifies the progress that has been made. But, in general, the observations reflected here and made in a period extending over more than a decade continue to apply and to deserve urgent attention. We mention this, not in a spirit of 'We told you so', but to urge the reader to take the entire book seriously and not to treat it as of only historic interest. It is to be hoped that republication of the material will, in conjunction with the many other initiatives currently underway, help finally change the situation once and for all.

L. A. BELADY
M. M. LEHMAN

CHAPTER 1

INTRODUCTORY REVIEW

In late 1968, one of the editors of this book (MML) was asked by Dr A Anderson, then Director of IBM's Research Division to "undertake a study of programming in IBM and to propose research projects that could seek ways to improve the Corporation's capability in that area". The immediate trigger for this study was an internal Bell Telephone Laboratories report. This had indicated that introduction of the IBM TSS/360 system as an interactive programming support system - to use modern terminology - had produced a threefold improvement in programmer productivity in the Electronic Switching Systems division of the Laboratories.

Lehman accepted the assignment. His immediately preceding experience in Project IMP [LEH66], had already resulted in a direct, personal interest in the methodology and effectiveness of program design and programming and provided strong motivation. It probably also influenced the direction and outcome of the subsequent study.

Project IMP, an attempt to develop a large multi and parallel processing system had been initiated in 1964, passing subsequently through three distinctive phases in a four year history. The first sought to develop a multiprocessor hardware system and to investigate its potential performance [LEH68b]. At the end of the first year answers had been found to many of the questions that faced the design team and apparently adequate solutions to many of the identified problems. The team became convinced, however, that they had attacked the wrong problem. The main problem was not the achievement of a satisfactory design for the hardware configuration and elements but how to control and fully exploit the workload, input, output and system resources during system operation. One should establish design criteria and seriously begin to development of hardware facilities only *after* conception and design of an executive system that provided the system management strategies and mechanisms.

Phase two of the project was therefore initiated. This was to study executive needs and strategies; to derive and develop the design of an executive system. Work on this aspect of the project proceeded for a further year by which time a preliminary architecture and design for the IMP Executive had been prepared. However once again the group

felt that only the relatively simple problem had been solved. The *real* problem lay not in the design of a specific system but in the *methods* and *methodology* of design. Software and system designs were unlikely ever to be optimum or complete. System application and system potential *evolve* (though that term was not used). The principle problems were therefore how to *approach* system specification and design; how one might limit the time or number of iterations required to achieve a satisfactory system; how, in a climate of rapidly advancing technology, one might teach design and transfer experience from one system to the next, from one team to the next, to facilitate subsequent adaptation of the system to changing needs and potential. The prime need was for the identification or development of design methods and methodology.

So Project IMP - phase three, a study of system design and programming methodology was initiated. It soon led to a number of reports and publications including a very fundamental contribution [ZUR67] that went largely unrecognised at the time. The time was clearly not ripe for methodological studies. The work of the group was neither recognised nor appreciated, basically it was not understood, and project IMP was disbanded.

The study of programming that followed made no apparent impact within IBM. It had focussed mainly on IBM internal practices, achievements and problems but also included observation of experience outside the corporation. Its findings were summarised in a confidential report [LEH69], in a series of presentations at various IBM locations and in a proposal to the IBM Director of Research for several research projects that would address some of the problem areas identified and explore potential solutions. And that was that.

This is not the place to speculate on the failure of the study to generate action within IBM, or to ask whether in a different environment there might have been a more positive response, or even to speculate in detail on the relevance or significance, then and now, of the conclusions presented. In the preface we have already said that in our view most of our observations are as relevant today as when first made. The reader may wish to make his own judgement after reading the, now declassified, report republished for the first time as Chapter 3 of this book. In doing so he should bear in mind that the report was generated before the Garmisch report [NAU69] had become widely known, before establishment of the

IFIP working group WG 2.3 on programming methodology and long before the concepts of Structured Programming, Chief Programmer Teams, Software Engineering and other universal panacea, had become popular.

Completion of the study may not have made an impact within IBM or on the wider community. It did, however, have significant consequences for the author and, subsequently, on a small number of colleagues. In particular, it yielded observations which became the seed from which the study of program evolution and the programming process developed. The report had included a brief analysis of the programmatic characteristics of OS/360, then in its 16th release. The results summarised in its section 1.3 (see ch.3), led to the recognition of phenomena whose analysis led slowly but surely to the concepts of Program Evolution Dynamics. The intent of this book is to trace the development of both the interpretation of the observed phenomena and of the concepts.

A year or so before Lehman undertook the study referred to in the previous paragraphs, Gerald Weinberg had written a paper which was, however, not published till 1970 [WEI70] and then in a relatively obscure journal. The present editors did not see the paper till many years later, so that it did not influence their work. Nevertheless it is, probably, the first paper to seriously discuss evolution as applying to *individual* artificial systems. As such it clearly deserves a place in this volume and is reproduced as chapter 4. We stress evolution of individual systems because Simon's Compton lectures (1968) published in his 'Sciences of the Artificial' [SIM69] did consider evolution as it applies to *successive generations* of artificial systems. He does not appear to have considered evolution of operational systems as treated by Weinberg and ourselves.

After completion of the "Programming Process" report, its author suggested to L A Belady that one might be able to model some of the observations, by their analysis advance basic understanding of the process and apply the resultant insight to achieve improved processes. Subsequent results undoubtably confirmed this conviction that further investigation was both required and justified. Its pursuit produced a productive and continuing partnership that has already extended over one and a half decades; a partnership that first recognised the intrinsic evolutionary nature of software and that subsequently conceived and developed many of the interpretations and implications of software life cycle phenomena that have only recently become more widely

recognised and accepted. In due course that study led to the publication of a first report [BEL71].

That report, included here as Chapter 5, is not only of historical interest. The phenomenological observations made are still highly relevant even with today's much advanced technology. In fact, the developments of VLSI and the microprocessor, and the consequent increase in demand for reliable and effective VLSI designs and for ever increasing numbers of programs, will again bring into the foreground, many of the issues and problems identified in the report; issues which some (mistakenly) consider to have been largely solved by recent advances in programming methodology and programming languages. The emergence and widespread application of very large data bases, of non procedural languages (of various classes) and of artificial intelligence techniques as applied, for example, in intelligent knowledge based systems (IKBS) will result in a repetition of the problems and history of the past twenty years, albeit in a new form, unless principles and lessons that could and should have been learned from the software experience of the past are absorbed and applied [LEH82b].

The macro and micro models of the programming process that are discussed in this report are also still relevant. Their investigation and extension has, however, been neglected in recent years. They are referred to again in our subsequent publications but no further significant progress can be reported. The very real progress that has been made in strengthening, extending and interpreting the original observations suggests, however, that the time may now be ripe to follow up of some of this early work.

Chapter 6 is a reprint of a paper presented late in 1971 at a conference on Statistical Computer Performance Evaluation [BEL72]. It adds observations and conclusions additional to those reported in chapter 5 and reflects definite, if small, progress in some of the notions. It is included here because it was the first public (non IBM) exposure of Evolution Dynamics.

Early in 1972 the close partnership between the two editors was relaxed when one (MML) was appointed to the Chair of Computing Science at London University's Imperial College of Science and Technology. Long distance collaboration continued however, and intensified when the other (LAB) came to London on a Science Research Council Senior Visiting Fellowship. It was during this period, that the concept of

continuing evolution was first verbalised; when the term Program Growth Dynamics used until then was replaced by Program Evolution Dynamics.

During this period also, Professor Lehman delivered his Inaugural Lecture choosing as his theme a generalisation of some aspects and concepts of Evolution Dynamics. The text of that lecture [LEH74] is reproduced here as chapter 7. The paper included the first published reference to 'Laws', presenting three in some detail. After a brief discussion of the, then, current notions, concepts and models of program growth, the lecture identified one specific phenomenon, neglect of anti-regressive activity, and explored the consequences of the same human attitudes in economic, sociological and educational activities.

This six month London interlude was the only time during the entire period of the continuing exploration of the software evolution phenomena and its dynamics, that either of the present editors (LAB) was able to devote himself exclusively to the study. Throughout the remainder of the collaboration and to this very day, the study could be given only low priority, in relation to other, assigned, duties. The results of the -all too few - discussions of this period were summarised in a report published in 1975 [BEL75], and in a revised and edited version in 1976 [BEL76]. It is the latter version that is included here as chapter 8 and that represents the first reasonably complete published discussion of the program evolution phenomenon as developed at that time.

The Second International Conference on Software Engineering marked a further milestone in the development of the subject. A paper [LEH76b] presented at that meeting expressed for the first time, the link between evolution, the programming process and software engineering. The paper, reproduced here as chapter 9, includes the seeds of concepts that are developed more clearly in latter chapters of this book. It also includes the first empirical data and data analysis of systems other than IBM's OS/360.

The next two chapters present material that is not in the main stream of the development of Evolution Dynamics as an evolving discipline. They are intellectual offshoots of the main study, with practical implications. Chapter 10 [BEL77b] discusses problems in the organisation and management of software evolution and, in particular, the 'Parts Number' and 'Configuration Management' problems.

Chapter 11 is a more philosophical work [LEH76] taken from the Encyclopedia of Ignorance [DUN77] that, by implication, discusses the very nature of the emerging discipline. It identifies the intrinsic instability of any theory that claims to describe or explain the behaviour of a system that includes thinking people amongst its constituent elements. The paper therefore reports perceptions first mentioned in chapter 9, that led to the definition of E-type programs, as reproduced in chapter 19 [LEH80b].

With chapter 12 we return to the consideration of program evolution phenomena. The paper [LEH78] extended the phenomenological data base both in terms of the number of systems covered and of the amount of data available. These extensions permitted significant advances in the understanding of the nature and the process of evolution; advances reflected in this paper by, for example, the addition of two further laws. The stress on the significance of the process, its structure, content and evolution, as distinct from that of the product of that process, also becomes more apparent in this paper.

The process is largely pursued by people. In the early and mid seventies, methodology - more correctly 'methods' - was already a popular word in the programming fraternity, but the role and importance of tools and of integrated tool-kits was not widely appreciated. Chapter 13 reflects this position with a discussion of manning problems in large scale programming [BEL78b].

In late 1977 a meeting at Brown University discussed 'Research Trends in Software Technology'. The editors' contribution to that meeting is reproduced here in chapter 14 [BEL78]. The paper presents further significant advances in their understanding, with emphasis on the breadth rather than the depth of the topic. The stress on evolution in *large* systems indicates that, even at that stage, they still thought of the phenomenon as one stemming, primarily, from the complexity of the applications that such systems address and of the resulting system. It is therefore very natural that the following chapter 15 addresses that very issue. That paper [BEL79c], however, was in the nature of a survey paper and is included here to provide pointers to other discussions then available. The reader who wishes to obtain an up-to-date picture is referred to a more recent paper [BEN83].

Chapter 16 [W0079a] represents a variation on the main stream developments described in the earlier chapters. It stems from visits to Imperial College by Professor J S Riorden and subsequently by his colleague Professor C M Woodside. Their, then, main research interests lay in control theory and its applications and they expressed a desire to study program evolution from that viewpoint. Their observations and generalisations suggest that further application of control theoretic concepts to the study of program evolution could prove beneficial to an understanding of the evolution of artificial systems in general; a conclusion to be expected since, at the very least, experience based feedback plays a fundamental role in the evolution of such systems.

Chapter 17 [BEL80] develops the insight gained in the Evolution Dynamics studies, to discuss the problem of modifiability. It serves, therefore, as a natural transition path to the final four chapters that reflect a redirection of our investigations from consideration of the *dynamics* of evolution to its *nature*. This redirection is seen in chapter 18 [LEH80b] which discusses the underlying meaning and significance of the five laws. It continues in chapter 19 [LEH80c] with a final summary of the observed consequences of the dynamics, an example of their application and a brief introduction to the nature of programs, the programming process and the software life-cycle.

The next chapter [LEH81] continues this theme, developing, in particular, discussion of the software development and adaptation process and of the consequent desirable properties of programming support environments. This is followed in chapter 21 [LEH81b] by an exploration of the characteristics of the software life cycle, the process of software evolution, and the challenges and opportunities that arise from the development of a software process seeking to master it.

Finally, chapter 22 [LEH82b] attempts an assessment of the societal significance of the notions, concepts and discipline of software engineering, such as those presented in this volume. Mankind relies increasingly on the correct and timely operation of computers. The very fate of mankind and its survival may, in the end, depend on the dynamic characteristics of some program or other, on its having been updated in time. The warnings implicit in this chapter represent an appropriate conclusion to the book.

But that is not the end of this survey. The observant reader will have noticed that in this brief overview no reference has been made to chapter 2. That chapter is, in fact, the most recent [LEH82c] of the publications included in this volume. Having, so far, appeared only in relatively obscure proceedings, its inclusion provides the first wide dissemination of its contents. It truly represents the summation of the evolution dynamics studies; the foundation of theories of Program Evolution and of the programming or software development and evolution processes. The chapter has been deliberately placed at the beginning of the book rather than in its historical sequence at the end, to provide the reader with motivation for following the historical development.

We believe that the notions and concepts presented in this book represent the beginning of a new and systematic approach to the solution of the software engineering problem, by redirecting attention to the total *process* of software development; and by providing the conceptual base and intellectual framework to motivate and facilitate the top-down design of software *processes* and their *integrated* support. Even if the reader proceeds no further than the end of the next chapter, but absorbs and builds on the notions presented there, the book will have served a useful purpose.

CHAPTER 2

PROGRAM EVOLUTION*

1 Program Evolution

1.1 Historical Summary

Program evolution is now widely accepted as a fact of life. The phenomenon was first recognised in the late 1960s as continuing program *growth* [LEH69]. The growth then discussed related to improvement in functional capability. For the sequence of releases of a given system at least, this was assumed to be related to program size as determined by counts of program modules, lines of code or storage requirements.

Collection of relevant data and their interpretation subsequently suggested the concept of *Program Growth Dynamics* [BEL71,72]. Its refinement led to the realisation that observed phenomena should be interpreted as program *evolution*. This represented more than a change of name. It produced, for example, the hypotheses that were later formulated as *Laws of Program Evolution* [LEH74], [BEL76]. Continuing investigation has given rise to the beginnings of a discipline, *Program Evolution Dynamics*; yielding insight [BEL79], [LEH80ab], practical tools and management guidelines [LEP76], [LEH78,80b] and most recently a new view of the programming process itself [LEH81a,b].

It must now be accepted that evolution is, ultimately, not due to shortcomings in current programming processes. It is intrinsic to the very nature of computer usage; computing applications and the systems that implement them. This perception has led to the *SPE* program classification [LEH80b] and thence to the concept of *continuous programming processes* supported by *vertically integrated support environments*.

1.2 The SPE Classification

In the *SPE* classification an S-type program or system is *defined* as one for which the *only* criterion of success in its

Paper presented at Symposium on Empirical Foundations of Computer and Information Sciences, 1982, Japan Information Center of Science and Technology, published in J. Info Proc and Management, 1984, Pergamon Press.

creation is equivalence, in some sense, to a *specification*. The P-type is not considered here. An E-type is one *embedded* in its operational environment, implementing an application *in* that environment, as suggested by figure 1.

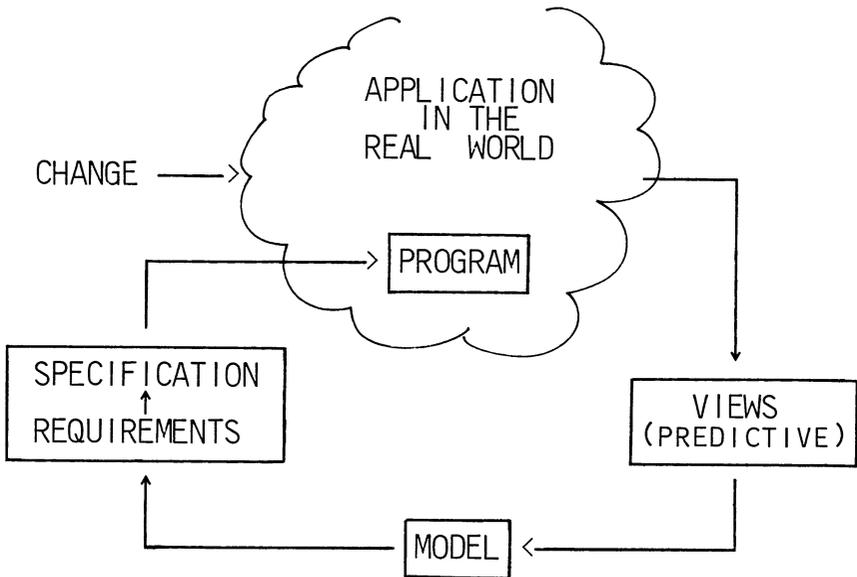


Figure 1 E-Type Programs

E-type systems have no intrinsic boundaries. The programs that implement them cannot have permanent and demonstrably *satisfactory* specifications since the *variety* of features that can be built into such systems is unlimited, meaningful permutations unbounded. Selection of features must take into account the perceived need, inter-related properties such as performance and cost and the nature of the operational environment once the system is installed. Moreover the very act of system *installation* changes that environment. And as operational experience is gained, perception of the problem and of possible solutions continuously advance, whilst exogenous pressures operate at all times to modify the environment, the problem and solution technology still further.

Now the acceptability of an E-type program relates to satisfaction in *usage* and it follows that the level of satisfaction will change with time. The real test of satisfaction occurs *after installation*, on the basis of system usability, performance and adaptability. But satisfaction of initial specifications is, at most, relevant for a limited period whose duration will depend on the foresight of the design team and the rate of change in the operational environment. Clearly, a specification that defines the *ultimate* E-type system cannot be conceived.

1.3 Evolutionary Traits

E-program correctness is determined by user-satisfaction rather than by equivalence to a specification. If it can be demonstrated equivalence is, therefore, merely a means to an end. It indicates a high likelihood of initial satisfaction. Continued satisfaction demands continuing change. The system will have to be adapted to a changing environment, changing needs, developing concepts and advancing technologies. The application and the system should evolve. The human effort to achieve this may, however, be withheld. Thus either the system evolves or its effectiveness and that of the application it supports will, inevitably, decline [LEH74 - first law].

As a system evolves its complexity increases unless specific complexity-control effort is applied [LEH74 - second law]. Complexity growth occurs because managerial guidelines seldom include its control as an objective. Instead they tend to focus on deadlines and on the cost-effectiveness of the *local* process or of the resultant system. No real *cost* is attached to structural deterioration for which the penalty lies in the

future. Complexity growth is therefore in part a consequence of weaknesses in current process-management practice. Advances in the process and in process-management can overcome this.

When, however, system change reflects extension of an application, it generally implies an increase in the complexity of *what is being done* [BEN82]. It leads, inevitably, to increased system complexity; though such increase may be hidden by the use of higher level (*internally* more complex) primitives, VLSI components for example.

The preceding paragraphs have described evolution as a phenomenon of functional and complexity growth. Current programming practice actually *exploits* evolution to achieve satisfactory levels of system attributes. Correctness, reliability, performance, capacity are all achieved *iteratively*. To the extent that their improvement reflects developing human perception and ambition triggered by use of the system or by change in the operational environment, evolutionary development is inescapable. The lack of adequate development technology, supporting engineering science and of evaluation calculi is, however, a strong contributing factor; in fact the ultimate cause. The knowledge, understanding and techniques to permit ab initio design and construction of a system with, at least, an initially satisfactory level of all attributes, simply do not exist. Their development, the emergence of a software engineering discipline, could significantly reduce reliance on iterative evolution of such function and quality factors.

Note that once the essential evolutionary nature of software is appreciated one need not, indeed should not, distinguish between *initial development* of a system and its subsequent enhancement or extension in a *maintenance* process. Software does not, of itself, deteriorate and so need not be 'maintained' in the traditional engineering sense. At most, a system is seen as no longer achieving its full, perhaps newly recognised, potential. Whether one begins with an application concept or with an existing system, all work undertaken to produce a system with more, less or different attributes or characteristics, constitutes evolutionary development. The term *maintenance* is, therefore, inappropriate in the context of software. Use of the term should be abandoned; replaced by *evolution*.

1.4 Process Dependant and Intrinsic Evolution

The above summary of some aspects of program evolution suggests that it is, in part, a consequence of the *process* of programming. Thus its rate may perhaps be reduced through the development and application of more advanced design and implementation technologies. Significant evolutionary pressure also arises, however, from the very nature of computer application and therefore of computing systems and programs [LEH80b]. *Intrinsic* evolution must be accepted as a fact of life.

With current practice, an average of about 70% of the life-time expenditure on a program is incurred after initial intallation. The main cost of a program is not incurred in its creation but in its subsequent evolution. To achieve an appropriate balance between initial product quality, continuing satisfaction and some desired life-time expenditure distribution, requires improved understanding of both intrinsic and process dependent modes of evolution; that is, a clearer insight into the basic nature and dynamics of the act of creation of a computing application through the development *or modification* of system constituents. The remainder of this paper analyses the evolution processes to lay the foundations for the systematic development of a systematic application and program development process; the *software technology process*.

It should be stressed that the analysis to follow is as relevant to VLSI 'hardware' as it is to software. As element numbers per chip increase, the latter will display all the characteristics of complexity, invisibility, evolution and uncertainty that have plagued software for over two decades. The resultant problems add, of course, to those arising from the technology itself. Moreover, as the manufacturing process becomes automated, chip functionality will be defined by its formal inputs as transformed by that process. It may even be a design option whether a chip functional-specification is used to control a manufacturing process or as source code for a program subsequently to be stored in and executed by a simpler chip. VLSI and software technologies can, therefore, be expected to have much in common. Thus in the remainder of this paper 'program' is to be interpreted as including both soft and VLSI implementations.

2 Systems Evolution

It is a truism to assert that all natural and artificial [SIM69] systems evolve. It may therefore be asked why the *dynamics* of program evolution (as distinct from its mechanics) should prove of interest when similar concern has not developed to any significant degree in the study of other systems. The answer to this question is both simple and revealing.

The time scale over which *natural* systems evolve is such that significant change is observable only over many human generations. The natural scientist cannot, therefore, observe change and development *as it occurs*. He operates as an archaeologist and historian, deducing the occurrence, mechanics and nature of evolution from static remains, relics and records of past events.

Artificial systems fall into several classes that vary widely in their characteristics. Consider first *socio-economic* systems such as cities. These too evolve. Noticeable change may occur in a matter of months, but developments that change the structure and character of a city extend over a human generation or more. The sociologist, by and large, does not monitor continuing change and evolution. He deduces it by comparing his observations with historic records. Because the rate of global change is relatively slow, its *dynamics* are at most *deduced*, not experienced.

Engineering artifacts evolve more rapidly. The motor car and the aeroplane, for example, have each seen eight to ten generations in as many decades. New models incorporating minor improvements are released periodically. Modifications may be introduced at any time and even retro-fitted to instances already operational. But the cost of total re-design and re-tooling is such that an essentially new system only appears once in ten years or so. Thus during his career the average aeronautical engineer will, for example, be involved with no more than three or four generations of aircraft. And if he experiences that many, he does so as apprentice, as mature engineer and as senior manager respectively, say. His viewpoint, involvement and responsibility is different for each generation. He experiences and views successive generations as a sequence of static instances, albeit of ever more advanced characteristics.

Finally, consider *programs*. These constitute the 'fruit fly' of artificial systems, undergoing continuing change and rapid evolution. The reasons are manifold [LEH80b]. The frequency and speed with which programs are executed, draws almost immediate attention to any shortcomings or mismatch and to developing or emerging opportunities. This leads to a constant stream of proposals for enhancements. That is, change proposals emerge rapidly because of the intimate coupling between the computing system in execution, operational personnel and the application environment. Once made, proposals for change are too easily accepted since their implementation involves little physical effort. The intellectual effort required is, in general, under-estimated and under-rated. It takes much bitter experience to demonstrate the cost of the effort subsequently required to implement the associated changes.

In practice therefore, it has become generally accepted that software systems evolve through the release of new versions at intervals ranging from less than one month to some two years. Mini-releases containing corrections and minor modifications may also be interspersed between these. Users, salesmen, executives and developers are all exposed to the stream of releases. At every stage of their career they are *actively* exposed to a sequence of system releases. They *experience* system evolution as a *dynamic* process influenced by and in turn influencing the environments in which it exists. The *Program Evolution Dynamics* studies of the last ten years have shown that the dynamics of that process may be modelled; *the models reflecting the discipline that underlies and regulates human society and the effort that implements change.*

3 The Current Programming Process and the Ideal

The software engineering and programming processes as currently practiced have themselves evolved over some three decades. As the state of the art in electronic computing advanced, methods, techniques and tools were conceived, developed, implemented and used to solve *specific* problems as these arose in *specific* environments. Together these form an ever-growing set of process primitives, from which total processes have been created by *ad hoc* association. These serve the needs of each individual production environment; additional elements being created as necessary to achieve local efficiency, effectiveness and cost-effectiveness.

Current programming processes developed during the formative years of a new technology. They had therefore to be assembled by ad-hoc association of such methods, procedures and tools as were available at any given time. Such bottom-up development inevitably leads, for example, to process discontinuities and to local rather than global optimisation. Once an adequate primitive set of methods, techniques, procedures and tools exists, one may, however, *design* a process top-down by decomposition and successive refinement guided by whatever criteria one chooses to adopt. One may, for example, try to synthesize a process using only available primitives. Alternatively one may seek to produce an *ideal* process which is then approached as closely as possible. Primitives may have to be defined and developed to fit needs not otherwise satisfiable, or to achieve significant gains in product quality or in process effectiveness or responsiveness. The latter approach has been adopted here. The paper seeks to identify an ideal process to serve as a base from which practical processes may subsequently be constructed.

4 Levels of Evolution

4.1 The Feedback Controlled Evolutionary System

It has been argued that changes in the operational environment constitute a significant source of evolutionary pressure. In part such changes are due to evolution of the environment itself in response to forces unrelated to the application addressed by the system. In part they are due to experience with system operation which, of itself, suggests corrections and enhancements to, or enlargement of the scope of, the application. That is, installation and operation of a system modifies the operational environment. E-type programs, as they have been termed, therefore include an implicit model of their own operation. They must be designed from a viewpoint of the application universe as it *will be* when the system is installed. The process of specifying and designing them is *essentially* predictive. It must be based on foreseeing the new perceptions of application need and system potential that will develop as a consequence of system usage. However good that *foresight* it must be imprecise [LEH77] and pressure for corrective adaptation will inevitably develop.

Computing systems, however, are not *self*-adaptive. Selection and management of change is the responsibility of managers,

who may well resist pressures and opportunities. But if a system is not adapted to its evolving conceptual and physical environment it becomes ever less satisfactory in the users' eyes. Ultimately it must be abandoned and replaced. Thus whether change is implemented continuously, by successive modification, change upon change upon change, or whether it is revolutionary, an outdated system being entirely replaced, or whether these modes alternate, the system will evolve on the basis of feedback provided, for example, by accumulated learning experience.

The system comprising the application and computing systems in their operational and system-implementation environments constitutes a multi-loop feedback system with both change reinforcing (positive) and change opposing (negative) feedback paths. At worst, the feedback leads to instability; always to continuing pressure for change. The rate of evolution, even though subject to management decision, will depend on the characteristics of the feedback paths. With current practice, four major paths and hence four levels of evolution may be identified. It will be shown that, of these, the two higher levels (slower rates) are largely intrinsic and unavoidable, though effective prognosis and prediction can reduce the rate of change. The two lower levels are largely process dependent. The development of improved, systematic, software engineering practices based on full understanding of why and how computing applications and software evolve, can minimise the evolutionary element.

4.2 Evolution Over Generations

The highest of the four feedback levels, that currently drive program evolution, arises from pressures that reflect changes in the operational and technological environment. The mechanism is similar to that which drives the evolution of socio-economic systems and engineering artifacts. As suggested in section 2 these evolve at a rate expressible in terms of decades or human generations. The increasing use of computers may tend to accelerate the process, but will be counterbalanced by an increasing need for stability and increasing system malleability. Relevant time scales are therefore unlikely to change dramatically. System life-time will continue to span ten to twenty years.

As indicated in section 2, evolution of a system over successive generations also covers successive generations of the personnel associated with that system. The *individual* thus experiences evolution as a *static* phenomenon

recognisable from separate instances of the system. Its dynamics are only observable by the historian. The impact of this mode of evolution on process technology is therefore minimal.

4.3 Evolution Through Successive Releases

During the life-time of each generation, the program *release*, at present, provides a mechanism for the controlled implementation of changes and their transmission to many users. That mechanism has been evolved and refined to a form peculiar to the software industry. Its special nature is due to the fact that software requires intellectual, rather than physical, effort to change. Software releases are therefore created by modification and change to the implementation itself, that is the code and the documentation, rather than by the creation of new instances as is the case for other artificial systems. A release may consist of a single change or of a number of unrelated corrections, enhancements and additions. Whatever the mix, observation suggests that the average work content of a sequence of releases, stabilises to a constant level. This consequence of the feedback nature of system evolution is linked to the effort required for *each* involved individual to regain and retain familiarity with the system [LEH80a - fifth law].

When setting release content and interval objectives, managers can apply alternative strategies [LEH80b]. But increasing societal dependence on computers implies a need for fast response to error reports and design deficiencies. In the early operational life of a system, release intervals of order one month are common. As the system ages, complexity and complexity control effort increase and with average release content constant, the release interval eventually stretches to two or three years. The determinants include the work to be achieved, user resistance to installation of a new system and delays in their mastering and appraising its new characteristics. In any event, the loop-delay in release-based evolution is conveniently expressed in months.

This second level of the current modes of evolution has been extensively studied, measured and modelled [LEH80b]. It is not further considered in the present paper.

4.4 Decimal or Sub-releases

Sub-releases are sometimes interposed between main releases to achieve fast response for the fixing of minor faults or blemishes or to provide urgently desired enhancements. This practice has some impact on the evolution dynamics of the system to which it is applied and to the parameters of the resultant process. The latter is, however, not qualitatively different to one in which releases of this category are not used. Nor is a sub-release sequence ever long enough to define an evolutionary level. Sub-releases may therefore, in general, be treated as part of the release process.

4.5 Developmental Evolution

4.5.1 The Ideal Process

Consider the *process* whereby an isolated change, a sub-release or even the release of a completely new version is taken from conception to eventual operation. Present industrial practice is the outcome of *ad hoc* evolution driven by expanding demand for computer applications, and for the programs to implement them. Its limitations provide the motivation and justification for seeking to unify and advance software technology so as to achieve an economical process that can be *planned* and *controlled*.

At initiation of an E-type development project, the picture of *what* is to be achieved and *how*, is, at best, fuzzy. Specification and design evolve iteratively as a consequence of feedback via various paths. The latter are often *ad hoc* and poorly defined, making analysis difficult if not impossible. To be amenable to analysis, the process should have well defined structure. This may be obtained by first identifying an *Ideal* process; a contiguous and coherent, non-iterative, sequence of orthogonal sub-processes, a set of sub-transformations, necessary and sufficient for the transformation of a computer application concept into an operational system. An analysis in terms of current process concepts, exemplified by figure 2a, that yields such a process, has been given elsewhere [LEH81a,b]. The time required for each of the activities indicated will typically be in the order of weeks and the process is third in the schema proposed in section 2.1.

The programming process, illustrated by figure 2a, is expressed in terms of activities that are widely recognised

and pursued in the current industrial process. More generally, one should view the process as a sequence of linguistic transformations [TUR82] with each model a representation of both the problem to be solved and of the system to be constructed. This is illustrated by figure 2b.

From either viewpoint (2.1) the process may be described as the *transformation* of a computer application *concept* into an *operational system* and its continuing *adaptation* to evolution of the operational environment. This transformation is complex. To achieve a practical process it must be decomposed into a series of sub-transformations. These define a structure of sub-processes, execution of which defines a system implementing the application concept. The system is represented as '*a model of a model of a model of a computer application concept in its operational environment*'. Each of the constituent models represents an abstraction of both the application concept *from* which it derives *and* of the system *to* which it is advancing by a process of reification. The models are *double* abstractions, a fact that has important bearing on the concept of a coherent software process supported by an integrated support environment.

The above reflects the dynamic view of the process. Its static counterpart regards each step of the development process as producing a theory for which the neighbouring steps provide models. In particular, the real world at one extreme and the operational system at the other are each models of the theories provided by the intermediate steps [TUR81].

Note that the term 'ideal' must be understood in the sense of the thermo-dynamicist's 'ideal cycle', in that it is believed not to be attainable in practice. Exogenous change in the operational environment and the consequent pressure for software adaptation is essentially unpredictable and cannot be accomodated without iteration. The following sections will suggest that iteration is also inescapable in specific software development. For example, total understanding of a problem and creative development of the best, in some sense, design for a computing-based solution can, in general, only be achieved iteratively [LEH77]. A linear process with only orthogonal activity cannot be achieved.

2.1 (Eds) For more recent viewpoints see [LEH83], [LEH84], [PSP84], [PSP85].

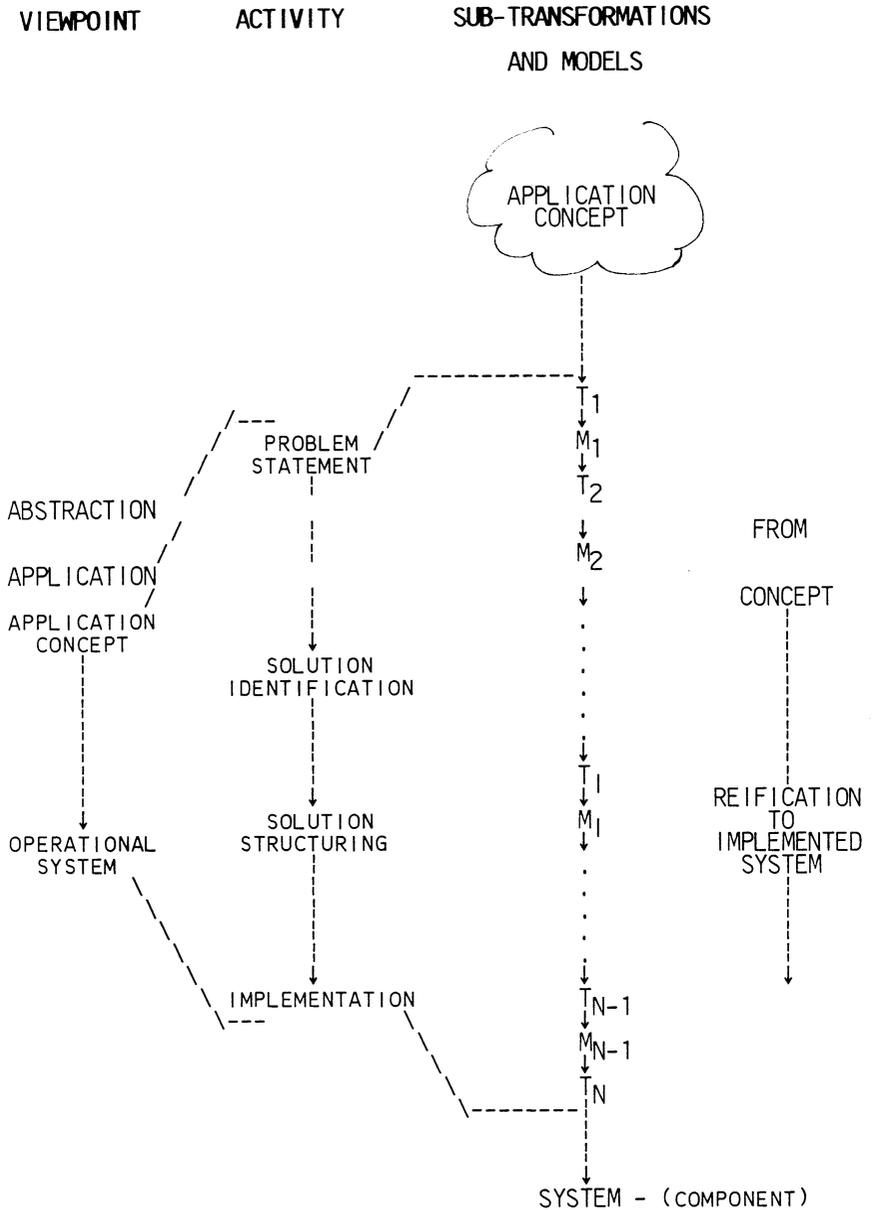


Figure 2b An Abstract Ideal Process

4.5.2 Iteration

Even the linear ideal process suggested by figures 2 is not unique. Any orthogonal set of steps that together cover the necessary and sufficient activity required to produce the target system serves the same purpose. But the sequential process is an idealisation that, in practice, cannot actually be achieved. Thus the detailed role of each step is not significant. It is the structure that provides the starting point for the development of a practical process.

The desire for a programming process that is linear is not new. Such an objective has been implicit in the search since programmed computers were first invented for improved programming methods. The development of high-level languages, for example, could be interpreted in those terms. The trend towards formal specification and the search for techniques that automate the transformation required to convert a program specification into an operational program [DAR79], [KOW79] contains similar implicit objectives.

The specification of any system embedded in an application environment is, however, inherently incomplete. A linear process without evaluation based human decision is, thus at best, only possible when the *structural model* has been defined; that is, when the evolving E-type design has been decomposed into a structure of S-type elements. It has been hypothesised that the latter decomposition is always possible [LEH80b]; in fact a programmer should never be required to commence a *programming* task until elements to be created have been fully specified. Nevertheless, the fact that a selection must be made between meaningful alternatives indicates that, to some extent at least, iteration must be used. In fact, with presently available software process technology, it will occur at three distinct levels in any practical implementation of processes based on the ideal illustrated by figures 2. These levels correspond to the three lowest of the levels of evolution identified above. From the outlines that follow it will be seen that they reflect the organisational feedback paths involved in the various activities.

Section 4.3 discussed system evolution via the release mechanism. The pressures that drive this process, and the justification of any changes undertaken, arise from continuing information exchange amongst technical personnel, from similar exchanges with users and from exogenous change. This may be reflected in a model derived from that of figure

2a, by means of a feedback connection over the entire process as shown in figure 3 by the outer loop. The information fed back reflects the experience and insight accumulated during development, implementation, installation and *usage* of the system.

No calculus is readily available to support linear progression from model to model over the steps that derive the succession of models that collectively capture and embody the output of the software process in the form of evolving design detail. Nor is there a calculus that permits forward evaluation of a design over that process. *Validation* of design decisions at each step, based on assumptions about the remaining process and about the primitives available to implementors, is ad hoc [LEH82]. In the absence of adequate methods, errors, weaknesses and omissions are uncovered only as the process proceeds. Thus every now and again earlier decisions must be reviewed; the models that embody them revised. Such review is currently, and in general, casual little attempt being made to update any but the most recent documentation. In the future, increasing emphasis will have to be placed on explicit review of all models affected by a change, by explicit backtracking over the feedback paths encompassing one or more steps, as indicated in figure 3.

The third level of iteration *occurs* in that portion of each sub-transformation concerned with design of the new features to be added to the model at each stage. It arises because no analytical design method is available. An iterative approach based on intuitive trial and evaluation must therefore be used. This is discussed further in section 4.8.

The above discussion has suggested that iterative design and implementation (the evolutionary programming process based, at least in part, on trial and error) is the consequence of inadequate design theory. An adequate theory is essential if a systematic technology to cover the total software process is ever to be achieved. There exists, however, a fundamental dilemma that complicates the development of such a theory and may well frustrate it. It stems from the fact that system design is two dimensional; creative design must be explored in, at least two directions.

On the basis of figures 2 and 3, one may hypothesise processes in which each model in the sequence is completed before proceeding to the next. A specification, for example, can be considered as being developed by a question and answer process that produces a tree-like structure. When the tree

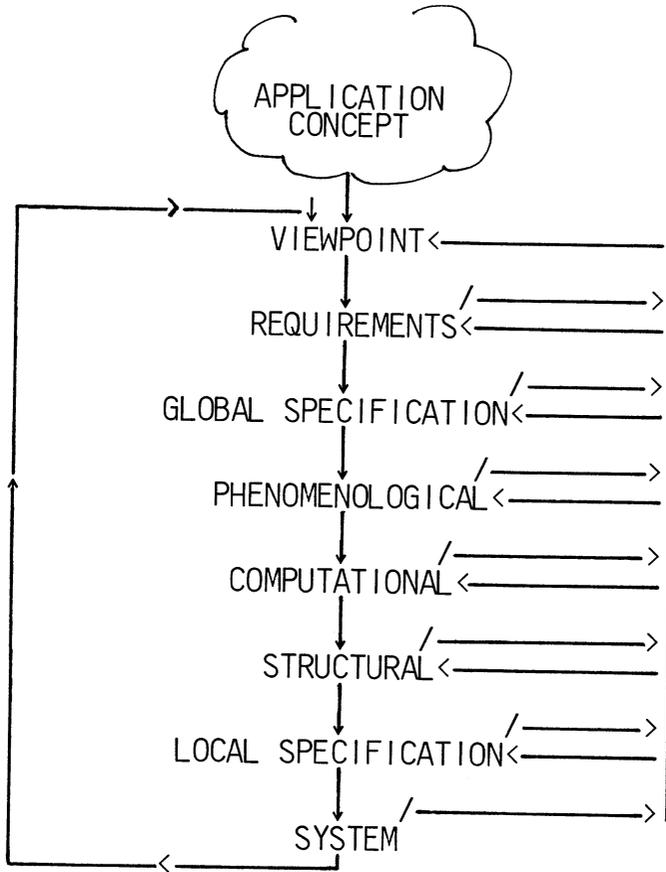


Figure 3 Iteration on the Ideal Process

has been developed down to leaves about which all questions have been directly and uniquely answered, specifically or by 'don't care', the specification is complete and provides the input to the next transformation. Were such a process possible at each stage of the overall process, if the human questioners and decision takers were all-wise, then this procedure could define a sequential macro-process, but with iterative loops for tree development.

Alternatively one may visualise a procedure that limits the descent at each step to one level of refinement; it being recognised that the optimum decomposition may well be a function of decisions still to be taken in future steps. One proceeds, therefore, along several steps of the transformation sequence, before returning to advance the design further by additional decomposition.

4.5.3 A Practical Process

At the present time there does not exist a method of programming based exclusively on one or other of these alternative approaches. Whether either can yield such a method remains an open question [LEH77]. Strong grounds exist, however, for believing that they do not; that a non-iterative, linear, process cannot be achieved. Any practical process must represent a compromise between the two extreme methods, a development from the abstract ideal including, at the very least, several levels of iteration.

Which direction to pursue, and how far to proceed in the refinement process at any given stage of the development, is probably one of the most difficult and critical decisions of the software design process. The software engineer must decide when to backtrack to which model in the sequence of process models to return and which of many alternative design paths to explore. The decisions taken determine the future course of the process. *Process design* is thus itself a critical process activity, an activity that cannot be concentrated in an initial, or any, process step. It must be accepted as an ongoing consideration distributed over the entire life of the software.

In summary, the total programming process is inherently an iteration of iterative steps. Their execution implements evolutionary development, progress towards the final goal by refinement and by resolution of imprecision or incompleteness in original concepts and in individual design and implementation decisions, by gradual selection from the

variety of options that are open to the designer; by exploitation of system permissiveness [MAI84]. As progress is made through the process, the full set of models which collectively constitute the system model must be maintained consistent each in itself and with one another [BEN82].

4.6 The Role of the Ideal-Process Concept in a Theory of Program Evolution

Despite the fact that the 'ideal process' is almost certainly not attainable in practice, the search for an integrated, maximally mechanised (tool supported), software development process can still benefit from the concept; identification of its structure, components and properties. The concept constitutes a useful abstraction to aid formulation of a *theory of program evolution*. Such a theory is regarded as essential as a precursor to the establishment of a *coherent process* for software development. The concept of a coherent process is, in its turn, vital if the much used term of 'integrated programming support environment' is to assume real meaning; if such systems are to be constructed. It should be noted that the term 'development' is here used in its fullest sense to include continuing evolution of the software to adapt it to the needs and opportunities of a changing operational environment. Programs must not only be good in the first place, they must be adapted to remain good despite exogenous changes.

Is it meaningful to seek to develop a theory of program evolution? The fact that programming processes 'yield development via a series of changed steps', satisfies both the Oxford [OXF33] and Webster [WEB59] definitions of 'evolution'. But is this all? Can similarities with other evolutionary phenomena be identified and prove helpful in achieving understanding of that process or conversely of the evolution of other forms of complex systems?

Consider the steps of an E-type application development. These involve selection between alternatives, *natural selection* with *survival of the best*. The process relies heavily on human perception for injection of the consequences of exogenous change analogous to *mutation*. *Adaptation* to environmental changes plays a key role. These facts suggest that software evolution may have much in common with that occurring in other artificial and in biological systems. The significant difference may lie, primarily, in its reliance on iteration rather than on parallel development and hence on the rate of evolution.

In any event one may conclude that the overall theoretical structure, regarded as key to significant advances in the emergence of a discipline of software engineering, may be developed from the postulate of an 'ideal process' and its instantiation as in figures 2.

Its practical approximations will be based on iteration. However, with advances in technology, the dependence on iterative development for the lower levels of the total process will decrease as analytic design and validation techniques are developed for guiding and controlling the selection process. Examination of the process step, lowest in the levels, of the evolutionary hierarchy will further clarify this issue and provide additional insight into program evolution.

4.7 The Step Paradigm (2.2)

4.7.1 Its Core

A recent publication [LEH81b] presented a paradigm describing the activity required in each of the steps that together realise the release development process. After a brief discussion of the paradigm, the present paper will isolate its evolutionary component to determine the degree to which such evolution is intrinsic or technology dependent.

The core activity of an elementary step is illustrated by figure 4. At the highest level of abstraction it represents a *transformation* of an *Input* (model) into an *output* (model). The transformation may implement changes in representation but is primarily directed at achieving some refinement to advance transition to the object system. The process of notational change, restructuring and refining of the input model, in ways to be discussed in section 4.7.3, is termed *design*.

Input and output models also provide a means of communication between designers and between them and their clients. The models must therefore be accessible in a structure and notation that makes them comprehensible to humans, who have to base decisions on their understanding and appreciation of them. A suitable representation for human comprehension may, however, not be the most appropriate for optimum decomposition and refinement. It is therefore appropriate to

2.2 (Eds) For more recent work that describes the canonical step paradigm, see [LEH83].

present the first level description of the core of the paradigm as a sequence of three sub-steps. The first transforms a communication oriented representation to an internal, manipulation oriented, form. The third, if needed, produces a form appropriate for the output interface. In between there is the *Design* step.

Design is achieved by the application of human judgement and decision, on the basis of defined immediate objectives and long-range goals. It must consider all potential inputs to and the desired output from the current step and the total process, the constructs or primitive elements available for the current step, the nature and power of the remaining process and the primitives available to it.

Structuring at each step facilitates intellectual mastery of the total complex. If interfaces and interconnections between the identified parts can be completely specified, it also permits division of further design activity, amongst participants or groups, for that step or for the remaining process. The potential activity split is indicated in figure 4 by the dotted lines out of the design box.

The preceding discussion has indicated the *role* of the design step. Its own design, to produce a practical process in a specific context for example, requires systematic decomposition, structuring and refinement of the basic concept; that expressed above for example. A preliminary analysis aimed at determining a lower level paradigm, is outlined in section 4.7.3. Discussion of the step paradigm must, however, first be completed.

4.7.2 The Complete Paradigm (2.2)

In the absence of precise design calculi, the activity outlined by figure 4 must be supported by activities that address questions such as, 'are we building the system right?', and 'are we building the right system?' [BOE81]. Ideally, each step of the design process must, in the most general sense, be validated. The complete first level description of the step paradigm, illustrated by figure 5, indicates how this might be achieved.

Each step-transformation produces a model that, if satisfactory, represents the input to the next step of the development process. What is 'satisfactory' in this context? The input and output transformations are purely representational in nature. They involve no changes arising

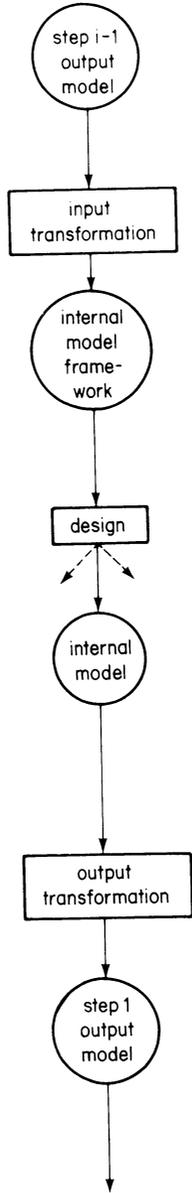


Figure 4

The Step Paradigm Core

from design decisions. Hence the requirement is equivalence, in some sense, between each pair of inputs and outputs.

Demonstration of such equivalence is, here, termed *Vertical Verification*. One could achieve it by *proving* equivalence, by a demonstration that each constituent sub-step of the transformation is correct (*constructive correctness* [DIJ68]) or by demonstrating once and for all that the *transformer is correct*, all in the context of the current step and its primitives.

On completing a design step (2.3) it must be shown that the resultant model is, itself, consistent and that it is complete in relation to the features that were to have been added or the problems that were to have been resolved in the current step. Such a demonstration is, here, termed *Horizontal Verification*.

When an acceptable model, in this sense, has been achieved it should be determined whether that model, *as an intermediate step in the total design process*, is *likely* to lead to an acceptable, or even optimum, operational system. In the absence of appropriate calculii, this judgement is imprecise, but should be based on an assessment of the current model, on where additional detail is required and how it might be developed, on implications of precursor models as to further generalisations or features that must be achieved, on the capability of the remaining process and on the primitives available for implementation of the remaining steps and the final system. The process of developing this assessment is termed *Validation*. In the absence of methodology that leads to a satisfactory design in a single step, the design step becomes iterative. Each iteration requires validation to determine completion of the sub-process. Horizontal verification is, of course, also desirable, obligatory in fact, in each iteration.

The above discussion has outlined the direct, first-level, elements of a paradigm covering the many steps of a practical software process. Though the issue cannot be explored here, it will be self-evident that the process as described, demands simple access to extensive records that contain the state and histories of the various models and of the processes that produced them. Perceptions and decisions that underlie design decisions, and the reasoning that produced

2.3 (Eds) *Horizontal verification will, of course, normally precede vertical verification.*

them, must also be preserved. Finally information relevant to the planning, management and evaluation of the entire process and its relationships to the environments in which it is executed, must also be recorded. The whole of this requirement is indicated in Figure 5 by the lines converging on 'Repository'.

4.7.3 A Preliminary Design Sub-step Paradigm

The present paper is intended to address the issue of evolution in the programming process and to develop, at least, the outlines of a theory of program evolution. It is therefore appropriate to pursue further refinement of the step paradigm only to the extent that identification of further detail assists its development. The specific objective must be to pin down and to clarify more precisely the origin of true evolution, as distinct from technology dependent evolution, in the design process.

The discussion of earlier sections presented the process of program design or development as a sequence of transformations. This view is particularly appropriate when considering the mechanistic aspects of the process. However, in so far as the transformations include a creative element that requires human involvement, it is more appropriate to describe it as one of refinement, as in section 4.7.1. The development of a lower level design paradigm may therefore be based on decomposition and refinement of the refinement *process*.

In his original paper [WIR71], Wirth defined refinement as the *addition of detail*. In his introduction he states that '...the program is refined in a sequence of refinement steps. In each step, one or several instructions of the given program are decomposed into more detailed instructions'. In his conclusion this is expressed as 'In each step a given task is broken up into a number of subtasks'. As Wirth clearly recognised, this process requires human decision. The way in which an element is decomposed will affect, at least some, attributes of the final product of the refinement process.

Wirth also discussed the need to define and *structure* associated data. More generally, structuring is, in fact, an integral part of the refinement process. As detail is added, the internal elemental structure will expand in a way that is dependent both on the original or higher level structure and on the process of refinement. At some stage, it may be advantageous to restructure the emerging element, by

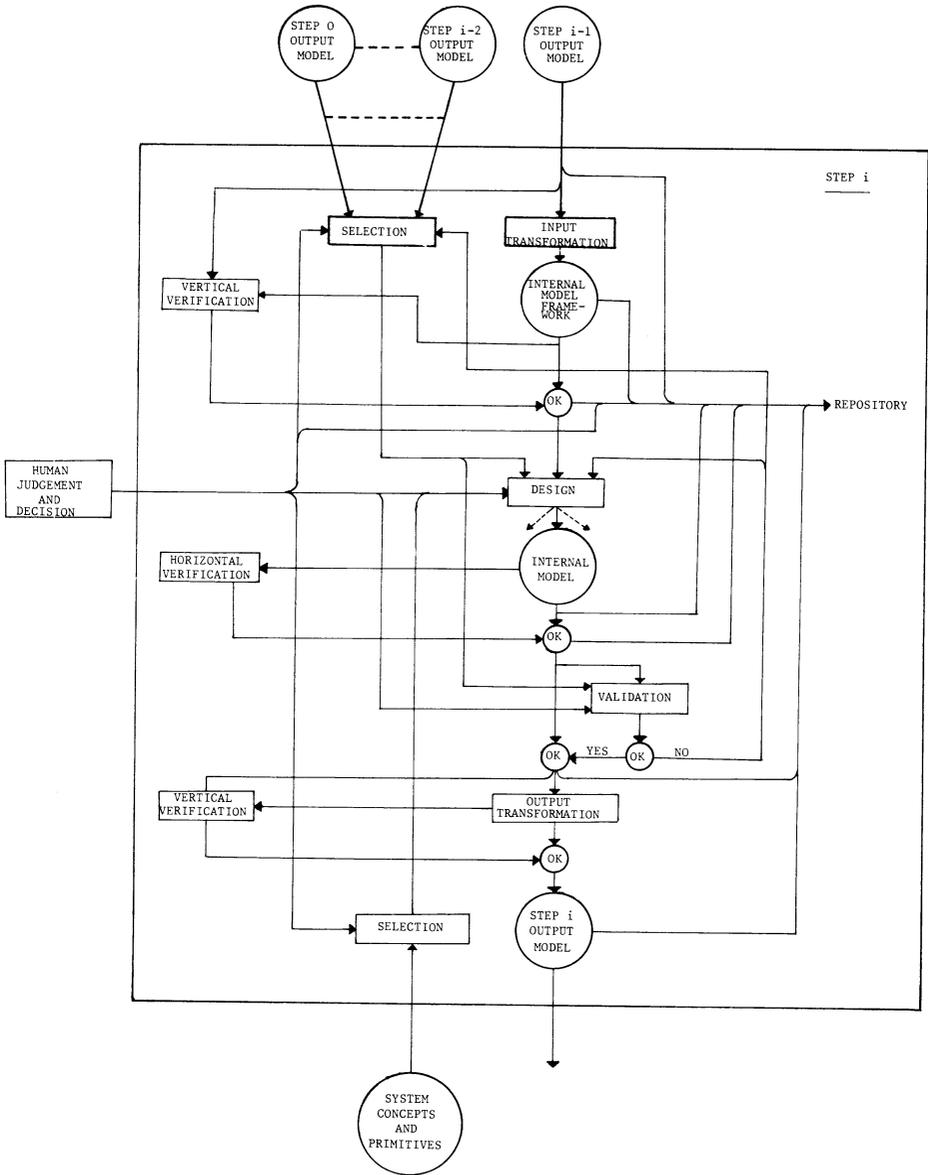


Figure 5

The Full Step Paradigm

associating its primitives in a different pattern without changing its internal semantics. Such restructuring may, for example, improve representational clarity and pave the way for further refinement, though it does not add detail in the Wirthian sense. Whatever the reason for undertaking it, refinement by restructuring is a further element of the design process.

As the process is followed, it will be convenient, occasionally, to focus on an aspect of the emerging design, even to the exclusion of others. Aspects that are then temporarily ignored will subsequently have to be re-introduced. This too is an element of the refinement process.

The above forms of refinement involve evolution because no calculus exists to implement them without iteration. Their evolutionary content is strictly process dependent. However, as refinement proceeds, new insights will inevitably develop into the nature of the application, the properties of possible solutions and potential extensions of both. This developing insight may lead to actual change of one or more of the models, or even to change of the original application concept. It is a mode of refinement that represents evolution in the fullest sense of the term.

4.8 Evolution in the Process Step

The previous section outlined a standard paradigm that describes the stepped activities that, together, achieve the transformation of an application concept into an operational system. The key elements of the constituent steps are the transformations that, jointly, achieve abstraction of the application concept and its reification into an operational system. They reflect the *design* activity that successively refines individual models to achieve the *incremental development* that the step is to provide.

After, at most, a representational transformation of the output of the previous stage, each input model is modified and extended by refinement to add detail, modify structure or impose change. This yields a new model sequence, with consistent and compatible elements, that represents progress towards the target system. The process is guided by the objectives of the current step and by knowledge of the remaining development process and the primitives in terms of which it is to be accomplished. It involves creative thinking; judgements and decisions based on knowledge, understanding, experience and intuition.

It is the *design* sub-step which drives system evolution. Refinement decisions impose verification obligations. They must also include consideration of the remaining process and its primitives, that is evaluated by *validation* procedures. Ideally, these should provide the best assessment possible, at the *current stage*, of the degree of satisfaction that can be expected from the system *likely* to emerge. Because of the imprecise nature of current technology, progress is iterative. That is, there exists, in general, no precise method for selecting the form or content of refinement to be applied. There is no systematic linear technique for selection between alternative structures, algorithms and primitives. In the absence of appropriate methods, or if insight develops in a way that suggests benefit can be derived by a change propagated across the sequence of models, iterative refinement must be applied possibly across several steps.

In summary, where appropriate (formal or analytic) techniques exist, a single application of the design sub-step can produce an acceptable output model. In the absence of such techniques, the design will evolve over several passes through a design and validation loop. Where validation methods are adequate, iteration and lowest level evolution may be confined to the internal step concerned. Where they are insufficiently refined or when *changes* affecting higher level models are introduced so that true evolution occurs, iteration must span several steps or extend evolution over two or more releases.

In any event, evolution clearly features at the step level of the programming process. At this lowest level the real times involved are of the order of days or weeks. It represents a lower level of evolution than that of the release-development process. At both these levels, however, evolution arises from processes tending to one or other of the alternatives identified at the end of section 4.5.2. There is some hope that in the future, development of adequate design and validation techniques will permit refinement of the design process to *reduce* reliance on evolution at the lowest levels. Whether they can ever be made sufficiently precise to confine evolution to the Release Sequence and Generation levels remains to be seen.

5 The Structure of the Evolution Process

The above discussion has briefly introduced a concept of hierarchical evolution and identified natural levels of a process implementing it. Table 1 summarises facts relevant to an associated theory of software evolution.

Table 1 Levels of Evolution

Level	Involvement	Feedback via	Time Unit	Mainly
Step-local Design	Individual	Designer's Perception	Days to Weeks	Process Dependent
Release	Project Group	Subsequent Process Step	Weeks to Months	Process Dependent
Release Sequence	Organisation	Users and Developers	Months to Years	Intrinsic
Generations	Society at Large	Real World Environment	Years to Decades	Intrinsic

Urgency may, occasionally, force an ad hoc system change to be implemented outside the established process. In general, however, system evolution will be constrained, so that change is achieved within processes set up at each level. Table 1 suggests orderly progression, a property that is highly desirable if a related theory is to be palatable.

The pattern has a simple interpretation. It reflects the fact that evolution is achieved by human action in a societal framework. Intervals that represent natural time constants in that framework, in the life and activity of individuals, groups, organisations and society at large, must have an impact on the programming process. The paper demonstrates that they appear naturally from an analysis of that process, both current and abstract. Their appearance is a hopeful sign that that analysis may bear further fruit. What are the immediate implications, particularly on the process and its support?

6 Software Process Support

One of the important conclusions of this study follows from the demonstration that the software process is a phenomenon that can be studied systematically in the context of the environments within which it is pursued. This view has always been implicit in Belady and Lehman's Evolution Dynamics studies. It reflects the tight linkage with the society within and for which computers operate. Many of its properties are a consequence of that relationship. We ignore the relationship at our peril [LEH78,80 - third law].

The notion of software evolution as a partially natural phenomenon leads to the concept of an *ideal coherent process* extending over the system life-cycle. During the last twenty years, ad hoc processes have evolved, assembled from equally ad hoc methods and tools. Each process has been adapted to the development environment in which it is to operate. What is now known and available as a result of this process evolution provides a rich set of primitives. These, in conjunction with the understanding achieved, yield the target implementation primitives for a top-down analysis that can determine a structured process approaching the ideal; and a basis for its practical implementation.

It is now widely realised that an effective programming process must be supported by an adequate set of tools. Such a tool kit must be coherent and integrated. The coherent view of the total programming process based on a theory of program evolution, provides a conceptual framework for the development and implementation of a methodology, a set of compatible methods and an integrated tool kit for their support. Space constraints prevent further exploration here. Preliminary discussion may be found with rapidly increasing frequency in the literature [DOL76], [HUT79], [BUX80], [RID80], [LEH80,81]. There is clearly much to be done. The present paper together with the referenced literature provides the concepts and a systematic and unified base for such an effort. The practical implementation of these concepts could rapidly follow.

7 Acknowledgements

The author has developed the concepts presented over many years in collaboration with a number of colleagues. The continuing and creative association with L A Belady is well known. More recently, Professor W M Turski has acted as a sounding board and constructive critic, contributing insight,

concepts and refinements. Most recently, Dr V Stenning has joined us in regular and productive discussions. The author is also deeply indebted to Drs G Benyon Tinker, P G Harrison and C Potts. Acknowledgement is also due to ERO and its Director of Information Sciences, Mr G M Sokol, for continuing encouragement and support, including that under contract number DAJA-37-80-C0011.

CHAPTER 3

THE PROGRAMMING PROCESS*

1 Growth

1.1 Expenditure

Any view of the IBM (3.1) or of the US programming scene today leaves an overwhelming impression of growth. Thus, for example, in the last decade the IBM System Development Division's (SDD) annual expenditure for programming development has increased more than an order of magnitude. Up to 1964/65 SDD programming expenditure was in fact growing exponentially at a rate averaging a doubling every one and a half years. Since that time the rate of increase has been linearised by holding its programming development budget at a fixed percentage of a linearly increasing divisional budget.

Expenditure patterns in other IBM divisions correspond to those of SDD. Figure 1 summarises annual expenditures for DP Market Development and SDD Type 1 programming. It is estimated that the total IBM expenditure on programming is several times more than that of SDD alone.

Projections see SDD programming development budgets continuing to grow, as will the total IBM programming expenditure. Thus even small improvements in the programming process can make substantial contributions to IBM's profitability. On the other hand, failure to control growth can lead to expenditure levels that will strain or even exceed IBM resources.

1.2 Manpower

Level of staffing is mainly controlled by the availability of funds. Thus the growth of programming manpower in IBM has

3.1 (Eds) *This paper was written while the author was an employee of IBM. Hence the constant references to IBM. One must stress that precisely the same comments that appear here, could have equally been made about the programming activities of any of the manufacturers or software houses.*

IBM Research Report RC2722, 1969, reprinted with kind permission of International Business Machines Corporation.

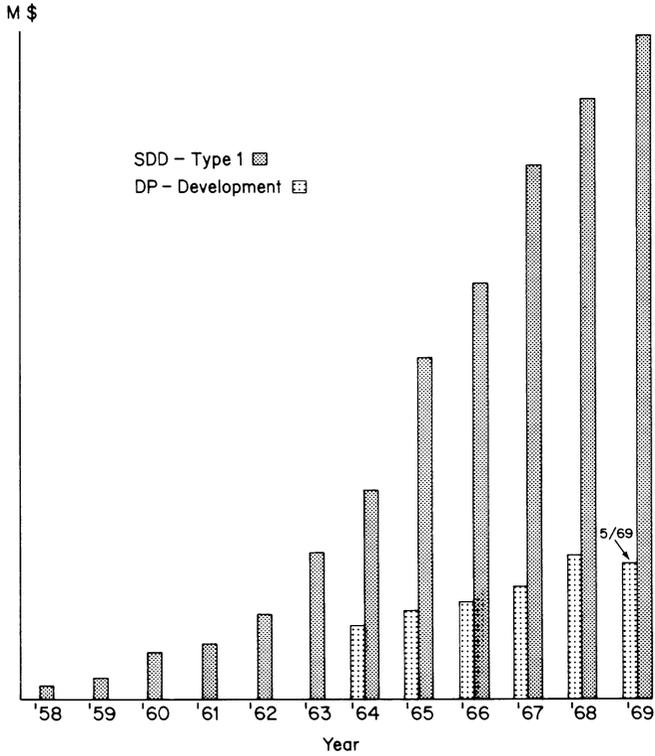


Figure 1 SDD and DP Programming Development Expenditure

followed a pattern similar to that of expenditure growth, as in Figure 2. Estimates of IBM's present programming population vary, as do the predictions of its growth. Clearly this growth represents a major source of increasing expenditure for the Corporation. Since it must be achieved in the face of increasing demand for programmers on a national scale, it will also increasingly create problems of supply, education and training.

The national picture is clouded by an absence of reliable statistics. The US Bureau of Labor has estimated that the year-end 1968 figure was in excess of 100,000 system analysts and 280,000 programmers. Various estimates have assessed the annual increase and training needs of the industry. These are not examined here but the yearly intake and training needs of personnel by the EDP industry is approaching 100,000 people and increasing as in Figure 3. Equally significant is the gap between demand and supply of personnel. A 1968 estimate [BRA68], for example, predicted a shortage of 100,000 programmers by 1970. This people-shortage does not yet appear to have affected IBM-internal recruitment of untrained personnel. Difficulties in the recruitment of experienced programmers are, however, already apparent.

There is also another impact on the Corporation. In a recent survey, [AMM69], one-third of EDP users stated that their number one problem was a shortage of programmers; for more than half, it was one of three major problems. Shortage of programming capability restricts customer satisfaction and therefore impacts IBM sales and expansion. That is these customer shortages must be regarded as an IBM shortage.

There may be no need for alarm at the present time within IBM about the supply of manpower. There is certainly no room for complacency. The problems need to be viewed from several points of view. Those particularly relevant to the present report relate to the selection, training and utilisation of programming staff. We return to these questions again in discussing productivity and education.

1.3 Systems

The cause of the growth discussed in the preceding sections is not a Parkinson effect. It originates in the development of new applications, demand for new systems and new function, and the support of new hardware. In addition programming personnel have also to deal with clean-up problems of existing systems and products.

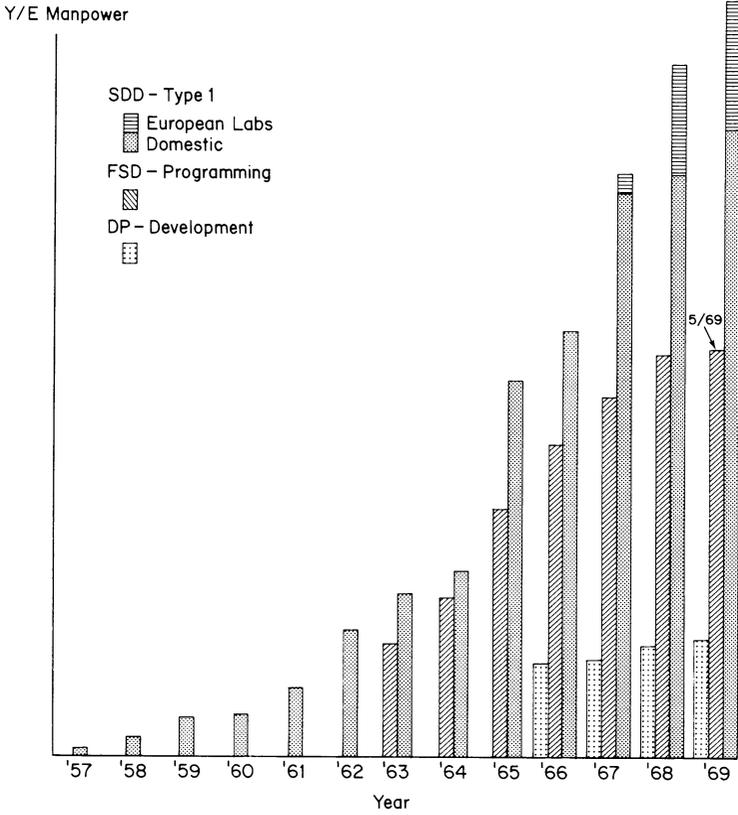


Figure 2 IBM Divisional Programming Manpower Growth

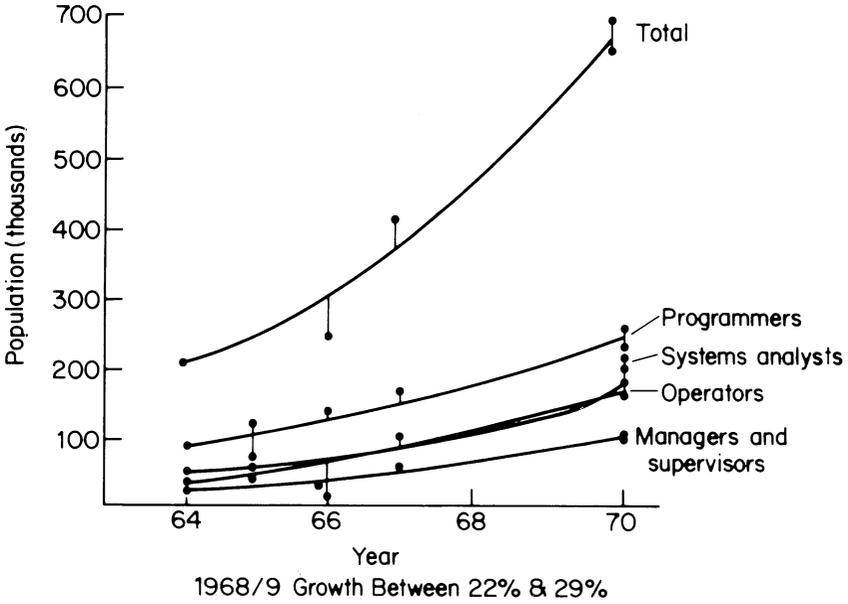


Figure 3 US Programming Population Growth

The overall growth in demand is, of course, in the industry's best interests. It is largely created by aggressive sales efforts, and is a sign of a healthy, expanding industry. The resultant expanding workload has, however, to be handled in the face of decreasing returns from the individual as systems become more and more complex. This complexity is partly due to the increasing size of programming systems, the increase in functional content and capability. Equally it is related to a need for generality, systems that must run on many configurations of the same basic equipment, on a growing range of machines; systems that must be upward compatible over generations of equipment, of languages and of programming systems.

OS/360 represents an example of increasing size and complexity as in Figure 4. Release 1 of March 1966, consisted of 14 components, divided into 1152 modules and 400,359 source statements. Release 16 of September 1968 consisted of 40 components, 3819 modules and some 1,740,364 source statements. Notice that a less than three-fold growth in the number of modules, has required a more than four-fold growth in the number of source statements. Similar growth rates are projected for future releases. Release 20 for example scheduled for shipment December 1970, is expected to contain over 53 components, 4635 modules and 2,200,000 instructions.

The clearest indicator of the rapid growth in the complexity of OS/360, however, is the number of modules that have required some change between successive releases. Increase in size associated with the progressive elimination of bugs, should relate the number of modules handled per release to the number changed in several previous releases. Thus it should remain approximately constant, decreasing as a percentage of the total number of members in the system. Comparison of individual neighbouring releases shows large fluctuations but the trend is clear. Successive releases tend to require ever *more* modification of an ever *larger* system as in Table 1 and Figure 5.

The average number of modules handled per release is increasing rapidly both absolutely and as a fraction of the system. The instruction growth per module handled is decreasing rapidly. There is no a priori reason to suppose that programmers are making more small, module internal, errors that require minor corrections in subsequent releases. Hence these trends are indicators of growing complexity. The

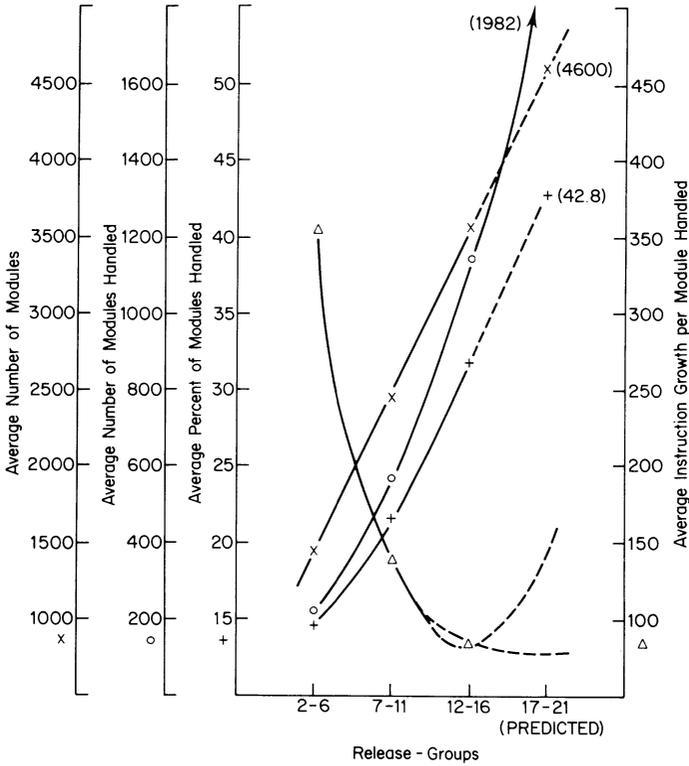


Figure 5 OS/360 Average Growth Rates

Table 1 Growing Complexity of OS/360

		Releases		
		2 thru 6	7 thru 11	12 thru 16
A	No of Modules	1451	2450	3752
V	Instructions/ Modules	406	447	438
E	No of Modules Handled	220	567	1151
R	% of Modules Changed	14.6	22.5	31.9
A	Instruction Growth/Mod Handled	356	141	80
G				
E				

effects of error and of change are spreading ever further through the system. It is this increasing difficulty of change that will soon force the initiation of an OS/360 successor.

By extrapolation it has been estimated that the successor will contain over 30 million lines of code and will cost accordingly. Successors to other programming systems will be required and will grow in similar fashion. These predictions are based on a fundamental assumption that the programming process will continue to be based on the same procedures as in the past. *They suggest that the time has come to develop a new approach to the entire process, to change the way of seeing and doing things (3.2).* It is this possibility that was examined in the present study, that formed the basis of project proposals that were made, and that constitutes the main topic of discussion in the latter sections of this report.

1.4 Cost

Past patterns clearly point to a continued growth in Corporate expenditure on the production of new programming systems. In terms of the present way of programming, we also expect significant increases in the cost of each individual programming system. For present day systems which are, at best, only superficially structured, cost will tend to increase more rapidly than linear - since complexity and hence difficulties of design, implementation, integration and testing all increase rapidly with size. For example the number of potentially interacting modules in an n -module system is $(1/2n(n-1))$. The number of potential interactions and their interdependence is much larger. The designer, the test and the modification procedures must all consider the need or the presence of such interactions.

The number of users of a particular system will tend to grow linearly with the growth of the sales force. Thus according to the present pattern, we may expect the cost of programming products *per user* to increase with time. This occurs at a time when the gradual introduction of LSI technologies has led to predictions of significant decreases in the cost of hardware. It has been suggested that in the face of these trends it will become profitable to implement more software functions in hardware. This argument is, however, misleading.

The difference in cost trends is partly due to differences in the technologies of hardware and software design and production. In particular there is an order of magnitude difference in the mechanisation and tool support provided in the two areas. But the underlying cause of software cost trends is increasing complexity. The problems that are encountered are due to the difficulties of *system* behaviour.

In the absence of any real understanding of complex systems, transferring implementations from soft to hard technologies will merely transform the hardware systems from relatively simple structures into the amorphous assemblies that are found in the programming area. The main problem of large systems is unintentional interaction between components, that require changes to the components for their elimination. Hardware changes in LSI technology are far costlier to implement than software changes. Thus the net result of a change from soft to hard implementation, *that proceeds at a pace more rapid than an improved understanding of system structure and performance* will be to cause still further increases in cost.

1.5 Cost Distribution

The cost of developing, implementing and marketing a programming system is spread over many different activities and components as in Table 2. There is no clear-cut borderline between many of these activities and the precise cost distributions are not known. Implementation and testing, however, each represent major elements in the total process; manpower and machine costs reach equal orders of magnitude.

1.6 Lead Times

The cost and the cost growth of programming systems should be of concern to the Corporation despite an accompanying revenue growth since the large sums involved imply that any savings that can be achieved can make significant contributions to Corporate profitability.

Equally significant, or perhaps even more so, is the total time that it takes to implement and deliver a reliable system. This time too can be expected to increase with size and complexity. If this growth is not controlled, individual customers may prefer and find it profitable, to satisfy their programming requirements by means of tailored or less general purpose systems than those that IBM must produce. Ie, while IBM cannot discontinue its production of general purpose software systems an increasing number of customers will use their own resources or utilise the many software firms now offering services, forcing price increases on IBM software.

The causes of long lead time cannot be accurately ascribed amongst the individual activities. It is clear however that control and improvement demands improvements in all phases of the programming process.

1.7 The Programming Process

The programming process is the total collection of technologies and activities that 'Transform the germ of an idea into a binary program tape' (3.3). At the present time there is little quantitative knowledge how costs, time delays and difficulties are distributed over that process. If improvements are to be made, the first priority must be the collection, assimilation, analysis, correlation and interpretation of data so as to achieve a better under-

The apparent ineffectiveness of rate of code-generation as a measure of productivity may also be viewed from a slightly different point of view. A recent survey of various IBM programming locations has revealed that programmers spread their action over more than twenty classes of activity spending no more than some 25% to 33% of their time 'Programming'. The nature of the survey raises questions about the precise value of this result. There is, however, a strong correlation with Bardain's estimate [BAR64] that a programmer 'Programs' 27% of his time, whereas an engineer 'Engineers' 45% of his time. Thus the survey suggests that the programmer is used as a 'Jack-of-all-trades'. Increased effectiveness of the programming process may well be obtained through more effective utilisation of professionals, aided by support from programming technicians and machines.

In summary it appears that questions of productivity require investigation of three related areas, the effectiveness of the total process itself, the utilisation of human and machine capability and resources in that process, and the productivity of the programmer when he is actually programming. All these questions are discussed briefly later in this report.

1.9 Education

The EDP-education needs of the Corporation and of the country are also growing at an alarming pace. That growth is needed to support recruitment and retraining as new soft and hard systems become available and as tools and support are developed for the programming process.

It has been claimed that IBM trains half a million people a year. Other estimates suggest a training need for over one-hundred thousand raw recruits annually on a national scale. A high percentage of these receive some of their education from IBM. Thus education forms a major activity within IBM, one that appears as an increasing drain on resources. It may, however, also be viewed as possessing profit potential and as offering hope for improved productivity and effectiveness through more discriminating training.

The apparent absence of programming technicians drawn from high school and two-year college graduates is related in part to the form of the programming process and the lack of management experience in the use of this labour source. Equally, however, it may be related to the programmer educational process. That process is discussed in a latter

section. At this point the availability, content and quality of EDP education courses is recognised as a major factor in the effectiveness of the programming process and in supporting the growing market.

1.10 The New World

The preceding discussion has been in the most general terms. The data on which it was based was all derived in the old, pre-unbundling, World. It is, of course too early to measure the effects of the New-World (3.4) environment on growth rates. However, it may well accentuate the effects of growth.

When a customer pays for a product or a service he is likely to attempt an assessment of its value to himself. Some program products will quickly demonstrate and justify their cost. Thus encouraged by program charges to evaluate programs, customers will discover their profitability and will be encouraged to ask for more. That is the New World will tend to increase demand, and hence growth in all the areas we have considered.

Moreover, to the extent that there is a relationship between cost and price and in view of emerging competition, it will be more than ever important to know and control programming costs in IBM. In this sense the New World posture will encourage development of appropriate evaluation and valuation techniques. The resultant pressures can only be beneficial to the effectiveness of the programming process.

1.11 Summary

In the preceding sections we have concluded that there is a high rate of growth in programming demand, manpower, expenditure, complexity and cost. Manpower is already a limiting resource for IBM customers and therefore for IBM.

The growth-control problems relate not only to the productivity of programming personnel but to improving the effectiveness of the entire programming process. Examination of the productivity question in isolation can even be misleading. It could lead to the ineffective transfer of

3.4 (Eds) *the terms unbundling and New-World were in common use in IBM in the late 1960s in relation to the IBM policy decision to charge for software, and its implementation.*

activities not directly related to programming, but essential to the release of a reliable product. Similarly, a tool developed to support the programmer rather than the process, may render him immediately more effective, but may be less than efficient or even harmful when viewed in relation to the total process

Thus, after a brief discussion on the control of growth, this paper will examine the total programming process; the function of structure, of men and of machines within that process. Improvements must bring about reduction in expenditures, cost, human effort, and the lead-time between statement of a requirement and delivery of a reliable program. In essence this must be achieved through increased effectiveness, and interaction for all those involved in the process.

2 Control of Growth

2.1 Form of Growth

The previous section has indicated the unabated growth of the programming area. This growth will continue into the future. In fact, an examination of the programming environment reveals all the ingredients of exponential growth. Thus, for example, new function and new system demand, the maintenance requirement and the impact of complexity all increase as a function of both the number of systems and the number of copies of these systems in the field.

Examination of IBM programming population and expenditure data clearly reveals such exponential growth for the period up to about 1966. Beyond that time, however, further growth tends to be linear. It is the purpose of this section to discuss the mechanisms that have controlled the natural tendency for unabated growth in the past and to propose alternatives for the future.

2.2 Environmental Control

Any increase of expenditure in a particular area of IBM's activity must compete for funds within the totality of IBM's expenditure. Thus, once the rate of growth of expenditure in programming exceeds the rate of budget growth of the environment, such growth can occur only by restricting growth or even decreasing expenditures in other areas.

As a business, with an almost infinite potential market for its product, IBM itself could also grow exponentially. In fact, however, it is controlled in its growth by the national economy and its own financial resources. Thus the Corporation has had to control, to linearise, budgetary allocations to each division. These in turn, have been unable to give free rein to various growth areas. Thus despite long queues of work awaiting the allocations of funds as typified by SDD's 'Outplan' and DP's 'Development Beyond Target', each division has applied a linearisation to its programming development activity.

In SDD, for example, the expenditure on programming development was set at a fixed percentage of the divisional budget some years ago. This rate was considered reasonable in relation to other calls on divisional resources and was not based on an estimate of value. Nor could it be, since reliable forecasting techniques were not available in the software area.

2.3 Open-Loop Control

In summary then, the growth control necessitated by resource limitations has been applied to programming expenditure and manpower growth through the medium of open-loop, budget, control. In the Old World the concept of profitability did not exist. Budget allocations did not and could not reflect the quality, value or profit potential of a program or programming system. Market needs and demands as determined by forecasting activities, were expressed exclusively in terms of impact on hardware sales. The concepts, technologies and tools required to establish software 'value' and hence to control the programming process by means of a self-regulating feedback procedure did not exist. Nor was there any incentive for their development in an environment where business policy regarded investment in software as an overhead cost.

2.4 Closed-Loop Control

The New-World pressures following unbundling create a visible need and the state-of-the-art should now progress, permitting a gradual transition to closed-loop control. Under such control, feedback derived, for example, from the potential 'value' of a program must play a major part in the decision procedure that determines its initiation and management of its development. In this context 'value' is ultimately measured by the direct and indirect profit potential of the program.

It will be related to functional content, implementation cost, performance, flexibility, changeability, reliability and the market needs that exist or can be created.

The incentive comes from the new requirement for controlled costs of programming products. Thus control must cause more selective and profitable development, the abandonment of unsuccessful programming efforts. This represents a major step forward over a control technique dependent primarily on human intuition and viewpoint. These may be objective but equally often will be parochial; coloured by local needs, prejudices and interests.

2.5 Base Control

Applying 'value' feedback will not significantly impact that part of the overall trend to growth caused by justifiable demand and increased complexity. An additional mechanism is therefore required. This lets the *output* of the programming process grow at its natural pace, while limiting excessive growth in expenditure, manpower and cost. It requires a change in the relationship between the input and output of the programming process. In terms of a mathematical model, the alternative to forcibly changing the form of a transfer function by constraints and feedback is to retain the exponential form but to change its coefficients. This may be achieved by changes in the environment and in the level and training of personnel (labour costs), changes in education, training and management (individual and group productivity) and changes in methodology and support (process structure, tooling, and mechanisation).

This potential has been recognised within IBM. The level of expenditure dedicated to programming support, however, falls far below the support level provided in other areas. Thus techniques and tools development in the Components and Manufacturing Divisions are funded at a level and order of magnitude larger in relation to the respective divisional development budgets, than the Corporate programming-support strategy budget in relation to *SDD* programming development expenditure in 1969. In fact Corporate expenditure on programming methodology and support represents a minute fraction of expenditure in the programming area as a whole. This already low relative expenditure is to be still further reduced as in Figure 6.

At this low level of expenditure, the main thrust has, in the past, been directed to the solution of local problems of

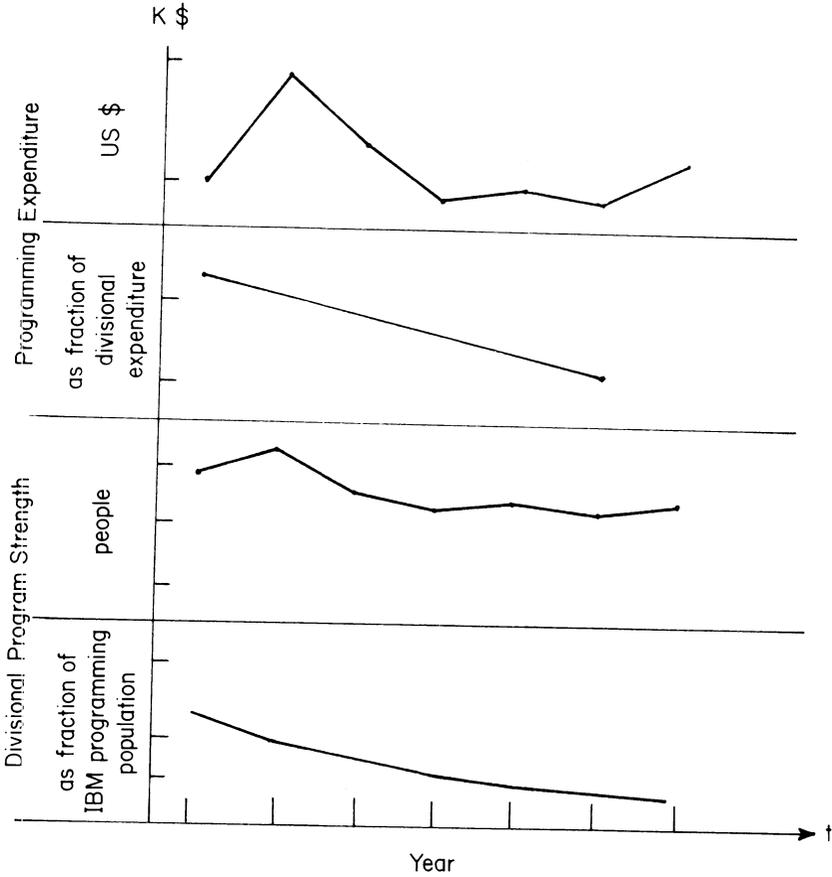


Figure 6 IBM Programming Development Support

immediate urgency. Attempts to structure, mechanise and support the total process, using computer technology wherever appropriate, have not been carried through. No examination has been made of the programming process itself so that modern technologies of data translation, manipulation, storage, display and communication, can be exploited to the fullest extent. (3.5)

The present programming process is a linear sequence of events, some of which have been individually improved. Improving some of these however, need have, and has had, no dramatic impact on the level of activity or rate of growth. Hence external controls have had to be invoked.

2.6 The Present Study

In the preceding sections we have outlined three approaches that individually or in combination can control the programming environment and render the programming process more effective. The first two force the linearisation of a growth potential that is naturally exponential.

At the present time control is essentially open-loop with budget allocations and managerial negotiation the main forces. It would clearly be desirable to replace such procedures by a self-regulating economic mechanism. This requires the development of insight into the programming process and market needs and the development of forecasting, evaluation, measurement, and management procedures, techniques and tools.

These very problems must have been widely discussed over the past few months in connection with the unbundling process. Their solution can make a fundamental contribution to improving the effectiveness and profitability of the programming process. Considerable effort will, however, be required to collect, analyse, correlate and interpret data and to learn how to use the resultant insight to the best advantage.

Ultimately, however, satisfaction of the demand that exists or that can be created for IBM software systems and products, demands modification rather than linearisation of the

3.5 *(Eds) We emphasise once again that this paragraph (that is, the entire paper) was written in 1970. Only now (1984) are these points beginning to be widely understood and accepted.*

exponential growth, despite its explosive trend. Such modification will result from an accelerated introduction of structured mechanisation to the total programming process. In the past the approach to mechanisation has been piecemeal. To determine whether there exists an alternative, what such an alternative might require and what projects and activities might be initiated at the present time, we now choose to discuss the programming process under the five headings of methodology (3.6), tools, languages, management and education.

3 Methodology

3.1 The Basic Need

Methodology is required wherever there is an involvement with the design or control of interacting systems. In the case of software products, systems of people interact over an extended period of time, within a business system, to produce a computer system based on a hardware subsystem. The total system will be judged in the real world by comparison with pre-set objectives and competitive products. The methodology must aim to produce decision taking and optimisation procedures that are system-oriented. This despite their localisation in space, time and people; despite the lack of a solid theoretical base.

An appropriate methodology will also yield all the advantages of structure for the process and for the resultant system. In particular it yields understanding and hence control of the development process itself. Only through such insight can one hope to transfer experience effectively from one system-implementation to the next.

3.2 Extremes in Methodology

3.2.1 Simple Programming Methodology

There is no single methodology of programming. Clearly there is a distinction between the procedure for writing a simple program fulfilling a specific need and that needed to produce a programming systems that is to address the potential requirement of an entire application area.

3.6 (Eds) *The correct word here and in all that follows should be 'Methods'. Methodology is 'the science of methods'.*

The former case is typified by a well-defined problem, limited objectives, an obvious algorithm, self-evident program structure, an appropriate language, and a one-man or small group effort. These conditions are realised because of the nature of the problem and because of limited expected usage and life. Thus, the methodology is straightforward. Select an algorithm for the solution and an appropriate programming language. This will probably be high level and machine independent. If the choice is available, decide whether to proceed interactively using a conversational system or under batch mode. Write, debug and run the program according to the simplest possible flow diagram and using the most readily available compiler.

3.2.2 The Real World Problem-Systems

This simple picture changes with the intrusion of real-world complexities. The objectives of a programming effort that is to serve a general purpose function cannot be uniquely determined. Even when a particular set has been adopted they are not usually formally defined. Over the period of development and implementation they are likely to change (in particular to increase in number and complexity). Thus programmers are essentially aiming at a moving target.

The size of the effort and the lead time that can be permitted are too large for one individual alone to be assigned. The work becomes a team effort which brings with it problem of interfaces, communication, standards and management. Program structure, sub-system assignments, and algorithms will have to be selected. Many of the details will not be pre-planned but will be left to bilateral negotiation or to individual implementors. Since the program is expected to be long-lived, its operating environment and the demands on the program will inevitably change. Thus the program will aim at a general purpose structure. Foresight is, however, always inadequate and thus the program must be changeable and expandable, often by others than the original programmers (3.7).

The system will be required to be efficient in terms of running time and storage space, since it is expected to be large and will hopefully be in frequent use over a long period of time. Thus a decision may be made to code in

3.7 (Eds) Notice here already the first roots of the 1981 SPE classification and of Evolution, concepts that nominate the much later papers.

assembly language, mistakenly in this author's judgement. The language problem is discussed in a latter section. We remark here simply that using an assembly language removes structure, discipline, clarity and readability, compounding the opportunity for error and the difficulty of change.

The problems outlined are really typical of these that arise in the design of any large system. A system assembled from its constituent parts achieves a performance determined both by the performance of individual subsystems and by the interactions between them. Local optimisation of each subsystem cannot ensure system optimisation. In fact 'Murphy's Law' (3.8) appears to guarantee that the final system, when assembled, will be far from optimum.

In general a system is designed by starting with the basic requirements and applying a breakdown or outside-in process that ultimately yields a blueprint in terms of standard components. Once the design has been completed, the implementation, which processes and assembles materials and components in inside-out fashion, can commence. This system design procedure cannot be presently applied to programming systems for two main reasons. First, analysis, measurement and evaluation techniques that permit such a design process to be undertaken are only now beginning to be available. Second, standard components of a sufficiently high-level to make the concept meaningful, are not available. In programming, the design process carries through to the composition of instruction sequences, coalescing the design, implementation, and testing phases.

The sequence of activities that take the original concept of a program down to its final tested and evaluated implementation is thus, by its very nature, continuous. If the process is to be improved it must be analysed and structures as a *whole*. Only then may it be divided into subactivities for the development of specialised support tools, processes and management structure.

3.8 *(Orig) We do not really need to appeal to Murphy's Law. If each subsystem has been individually optimised, by definition it can perform no better than when operating in isolation. Thus interactions can only degrade performance. Many subsystems degrading simultaneously in pseudo-random fashion cannot be expected to produce a near optimised system.*

3.3 Present Methodology

3.3.1 The Phase Review

Any programming methodology now existing has developed in ad hoc fashion over the years. The simple case of Section 3.2.1 in which an informal problem statement is directly translated into, say, a flow diagram and a FORTRAN or COBOL program and then compiled on a machine-independent compiler, will not be further discussed. We merely note the continuing need to develop machine independent, expandable languages and their processors, for ready adaptation to the needs of particular applications.

The methodology of systems programming in the more realistic environment outlined in 3.2.2 is conveniently discussed in relation to SDD's development guide. This linear set of procedures is based on a series of technologies; architecture, design, implementation, test, and so on. In particular the guide defines the details of the Phase Reviews that governs the development, test, announcement and release of quality hardware. The phase-review procedure has however failed to fulfill the same function in the software area.

Part of this failure is undoubtedly due to the intrinsic fusion of the design, implementation and testing processes in programming. Equally, it is due to the absence of techniques and tools that would permit technical evaluation of work completed, in terms of performance and cost forecasts, and accurate assessments of future resource requirement for completion of a particular programming effort.

In the absence of such skills the Reviews cannot lead to meaningful technical judgement of a programming project. Perforce they become a management tool that records past resource investments and determines a compromise between the functional content, core requirement, announcement and release dates, and the amount of resources to be committed to a program at that juncture.

3.3.2 Methodology and Structure

The preceding implies the need to replace existing unstructured technology-oriented programming methodology by an overall total-process-oriented methodology. This is seen as providing a structure to the process. This structure must be designed to guide the programming process and enable it to achieve any desired combination of performance, reliability

and cost for a minimum in human effort and maximum machine support. After all the cost of human effort is on the increase, the supply of human effort is limited and the contrary is true for machines.

By creating an appropriate structure for the process itself, complexity and cost are reduced, and human and machine effectiveness simultaneously increased. These benefits are important in their own right. They can also be expected to lead to significant improvements in system reliability.

There is also a strong relationship between the structure of a process and the structure of the system it produces. The impact of system structure on performance, cost, changeability and reliability is being increasingly recognised. It is too large to be left to chance. The need for restructuring the total manufacturing process follows, if the system structure at all levels of detail is to be a prime attribute and not an uncontrolled consequence of environmental conditions. At the present time and for the simple case of section 3.2.1, program structure is imposed by the language used, any ancillary conventions, the algorithms employed and the compiler. For the larger programming system, any structure that exists is largely a reflection of the management hierarchy that produced it.

3.3.3 Support Activity

In discussing methodology, reference must be made to current programming support activity. Examples of support available or under development are discussed in Section 4.2. In general, under pressures of local priorities and budget restriction, these develop into support for the status quo, serving local needs within the framework of current procedures. In the future, support activities will have to be system and total process-oriented, support pre-planned system structure and provide maximum opportunity for mechanisation.

3.4 Embryonic Methodologies

The previous section has outlined the case for the development of a process-oriented programming methodology. Fundamentally that process may be viewed as *the translation of an application or function concept into a working program* (3.9).

3.9 (Eds) 1984 *Ital.* Note the close correspondence between this definition and the present day view. Current

The urgent need for an integrated set of technologies that will help in the solution of the problems posed by this translation has been widely recognised. The various approaches that have been proposed cannot be detailed here. Table 3 lists some examples, and preliminary study of these exploratory processes provides the conviction that it is meaningful to discuss a total system process.

Management itself has taken a significant first step in the development of a total process by the breakdown of the development process into a series of missions. However, it must be recognised that this by itself is insufficient. The *real* problem lies partly in the formal and meaningful definition of missions and mission interfaces. Even more, the problems lie in their containment and management and in the subsequent breakdown of missions so defined into submissions, sub-submissions and so on. This breakdown process into a tree-like structure can be generalised at higher levels but must also be particularised for each individual system. Thus the mission-concept initiates an outside-in design process; it begins to develop both a structured process and the structured system that it can help to produce. This concept of a tree-like structure for both process and system, arising from an outside-in design procedure underlies most of the other system-oriented design concepts of Table 3.

Another example of an embryonic software development methodology is the multi-level modelling concept developed by Zurcher and Randell of the IBM Research Division [ZUR67]. Their work has already been repeated and expended [PAR67,69]. The basic approach recognises the futility of separating, design, evaluation and documentation processes in software-system design. The design process is structured by an expanding model seeded by a formal definition of the system, which provides a first, executable, functional model. It is tested and further expanded through a sequence of models, that develop an increasing amount of function and an increasing amount of detail as to how that function is to be executed. Ultimately, the model becomes the system.

An even more significant contribution made by the Zurcher-Randell concept was that such a model can simultaneously contain several representations of the same function or set of functions of a system at different levels of abstraction.

terminology would, however, use 'transformation' rather than 'translation'.

Table 3 Embryonics Methodologies

Configuration Management	Airforce/J Aaron	FSD
Mission/submission Concept	W S Humphrey	SSD
CP 67	N Rasmussen	DP
Syntax-oriented Documentation	H Mills	FSD
Architectural Systematics	H P Schlaeppi	RES
Multi-level Modelling	Zurcher & Randell	RES
Sodas	D Parnass	Carnegie
Software Engineering	Garmisch Conference	NATO
NASA Project Engineering	—	—

There are a number of advantages to such multiple representation, and it is they that really demonstrate the major advance represented by the proposed methodology. Suffice it here to say that the multi-level modelling concept once again integrates the various activities that make up the software system development process. Evaluation must proceed step by step with design and documentation and the various stages of design and implementation are replaceable and repeatable.

Outside IBM there has been widespread recognition of the need for a new discipline termed 'Software Engineering' [NAT68], strongly related to systems engineering. The potential for the development of such a discipline exists but awaits intensive, directed research and development effort.

3.5 Further Work in Methodology

The common element of the embryonic methodologies listed in Table 3, is integration of the technologies currently employed in the programming process. They regard a program as a system requiring to be developed from a formal statement of its objectives and show a strong tendency to use modelling and simulation as the main tool for achieving this end.

In fact it is becoming recognised that the programming process needs to become more and more like system design activities in other fields. Thus there is a prima facie case for supporting a study of general design procedures and relating these to the particular requirements of the programming process. In the set of project proposals arising out of the present study it was suggested that an attempt at formal description of the process, for example, could lead to the improvement of the process. In addition many of the listed methodologies and probably others available within and outside IBM, are worthy of further development and exposure in real development projects.

In view of the time period that must elapse before total-process oriented procedures could be introduced it seems important to improve present phase review procedures as applied to programs. In particular as analysis, measurement and simulation techniques for the software area become progressively available, it must become a requirement that the procedure includes not only resource-oriented data and assessment. It must also produce judgements on the technical attributes and expected performance of programming products (3.10). Clearly the initial specification must thus also include performance and test oriented data.

The need for development and standardisation of measurement, analysis, evaluation, testing and forecasting techniques and their associated tools cannot be over stressed. All play a fundamental part both in the feedback control discussed in section 2, and in the development of a meaningful system-oriented methodology. In addition the availability of standard components (or the logically equivalent systems programming language) at a sufficiently high level will be a significant element in the structuring of an effective methodology.

4 Tools

4.1 Tool-Oriented Processes and Process-Oriented Tools

When first introduced computers did not make a significant impact in the commercial world. The real breakthrough came only late in the 1950's when institutions stopped asking, 'Where can we use computers?' and started asking, 'How shall we conduct our business now that computers are available?'

In seeking to automate the programming process the same error has been committed. The approach has been to seek possible applications of computers within the process as now practiced. The Clear-Caster project which initially was an exception, has in the course of time, had to develop in the same fashion due to the day-to-day pressures encountered in the programming effort. Thus the problem of increasing programming effectiveness, through mechanisation and tooling is closely associated with the overall problem methodology. Its solution calls for a review of the process itself so that maximum benefit can be had from the use of computers.

4.2 Present Tools

At present there exist many uncoordinated tools in many different areas. Within IBM the Clear-Caster system maintains the main thrust. It has concerned itself primarily with problems of communication and of programming-data entry, storage, retrieval and display.

Other systems modelling, analysis and evaluation tools are also available or under development [SIM68], [LAC68]. In addition there exist various testing, documentation, management and support systems which can make significant contributions to the process as now practiced. The test case generators under development at Hursley, for example, seek to address the problem posed by the need to exhaustively test every large system.

These many tools support each other functionally. Unfortunately, however, their specification and development has been essentially uncoordinated so that they cannot communicate or pass data to each other. Often the data generated by them cannot even be correlated. Moreover, use of these tools is not universal but depends on the initiative of local or sub-project management.

4.3 Tool Development

4.3.1 Integrated Families of Tools

Specification and development of a coordinated set of tools supporting the total programming process must, as had been seen, consider the structure of that process. The present viewpoint of a long-range goal visualises a growing model, that serves as its own progressive documentation. It is seeded by a formal specification that develops as a result of human and machine manipulation, into an evaluated system,

possibly executing on a model of future hardware, and running under an interactive, conversational system accessed via a communication network.

The present report must of necessity be more modest in its aims. It seeks no more than to provide a preliminary judgement of what such a family could include and on what technologies it would be based.

4.3.2 Interactive Programming

Even without having clearly fixed the methodology, and structure of the total programming process, intuition and practical experience both suggest that a major class of tools arise from the use of a computer to provide or support the functions that the process requires. Moreover, since in one fashion or another, the system will have to communicate with all the participants in the programming process, it is highly probable that an interactive capability will increase effectiveness. Thus the programming tool includes hardware such as alpha-numeric terminals, printers and display units and the software that links them to the computer, provides appropriate function and integrates them into a total support system.

The system will be used by many people sharing the information that represents the developing design and implementation. These people will often be working in and accessing the resultant data-base from widely separated locations and connections in the system must be via a communication network. Thus the programming tool visualised belongs to the family of data-based, communication based, conversational systems recognised widely as *the* systems of the seventies and eighties.

A total-programming-process-oriented system will include facilities for specification, structuring, development, entry, filing and debugging of the program material (3.11). It will provide for execution of program material and for monitoring, measurement and analysis of its performance. Execution of program elements must be possible in isolation

3.11 (Eds) *The reader will notice the total absence here or anywhere in the paper, of any reference to 'verification; or 'correctness proving'. Clearly the author had not encountered the concept anywhere in his study, although Dijkstra's paper on constructive correctness had already been published [DIJ68].*

or linked to other available elements and possibly running under some system other than its intended host. In addition the system must provide, communication between members of the programming team, control by its management, detection of deviation from specifications, conventions or performance objectives, and dissemination of the material within and outside the programming group is provided by the system. That is the machine has become the composing, editing, documentation, display, control and communication centre of the entire activity (3.12).

4.3.3 Languages

The languages used by the programmer during the programming process can be considered as his most fundamental tool. Specification, structural development and coding of program, function and data sequences can all be considerably aided by the provision of appropriate languages.

The topic of language development and its place in the programming process is really the subject for an independent study. Some aspects of the topic are treated in the next section. We note here merely that the provision of tools and support is closely related to the provision of languages. Thus, for example, any computerised tool requires at least a set of commands for its use.

4.3.4 Productivity Gains

The concept of an interactive programming system is, of course, not new. Such systems exist already and have been used. Most of the experimental data that is available [GOL69] refers to one-man efforts in which communication between people and management control are not prime factors. All the published studies appear to indicate that in this environment, variations between programmers are of far more consequence to productivity than any improvement due to the use of an interactive programming system. However these conclusions are not relevant to the more complex systems programming effort. In the latter sphere, the only available numerical data comes from Bell Telephone Laboratory observations on their use of TSS/360. They claim that over a number of projects they have been able to observe gain factors of two to three in lines-of-code generated and debugged by programmers in unit time.

3.12 (Eds) *'Programming Support Environment' in today's terminology.*

The particular gains that BTL claim may be disputed for a number of reasons. We restrict ourselves, however, to two comments. On the debit side one observes that the factor of two or three is disappointing. What is needed is an improvement of one or two orders of magnitude. On the credit side however, we note that BTL have really been using TSS/360 in rather primitive fashion. Thus their results suggest that a considerably greater improvement could be obtained from the development of a system specifically oriented towards the programming process.

4.4 Tools - Conclusion

In the long run there is much to be gained from the development of a process-oriented programming system which supports and is supported by a central processor, filing facilities, terminals, communication and display hardware. For the moment use can also be made of existing programming systems such as Clear-Caster, TSS/360 or APL/360. Any such efforts should be monitored and measured so as to obtain much more data about both effectiveness and about the ways that people use such systems.

In addition to a determination of the direct impact of interactive programming on productivity, it is of considerable interest to qualify and size its indirect effect. For example, to what extent does use of an interactive programming system lead to a tendency to replace analysis by experiment? Does it cause the programmer to plunge into composition and debugging rather than to adopt a more carefully planned approach to these activities?

Equally significant is the effect, both positive and negative, of time and of familiarity. As the novelty of the terminal wears off how does the usage pattern and utility change? Observation, measurement and analysis of human behaviour in the programming process will enable determination of these factors. Feedback can then be applied to improve both the support tools and the process.

A major function of the interactive system is the capability it gives for communication and management control. Thus any controlled experiment which would attempt to measure the effectiveness of programming support systems, must be carried out on a large scale. That is it must be large enough to ensure a requirement for communication at, at least, two levels of management control. Such an experiment could, for example, seek to implement the same system twice in two

different groups each consisting perhaps of 30 to 50 people. One group would employ traditional batch method of computer usage, the other would access the computer interactively and have functional support available.

We note here that there can be no doubt that machines will be increasingly used in the programming process. The question that really needs to be decided at the present time is whether the support system is merely a management information system used in batch processing mode with remote job entry. The alternative, of course, is to let the individual programmer spend all or most of his time at a computer terminal.

The answers to some of these preliminary questions will determine the magnitude and nature of the resources that could and should be applied to the development of a total-process-oriented programming support system.

5 Language

5.1 The Aspects Discussed

This section addresses briefly three further aspects of the relationship between the programming process and the languages used in that process. It discusses in general fashion the language-level at which system-programmers can work and the formal specification of programming systems. It also comments on the semantic wealth of most formal languages and the impact this has on the structure and activities of the programmer population.

The merits and demerits of specific languages and the relationship between programming, command and job control languages are not considered here. Nor do we discuss the single versus multiple language concept or the properties of expandability and changeability. It is clear, however, that in a rapidly changing and growing environment the latter must be attributes of any formal language, much as they are attributes of all natural languages.

5.2 Language Level

5.2.1 Present Practice and Experience

Up to the present time most systems programming has been done in assembly language, though the use of BSL [MEL67] has begun to spread in IBM. This is contrary to the trend in

applications programming where FORTRAN, COBOL and, to a lesser extent, PL/1 have effectively taken over the field. Outside of IBM some high-level systems programming experience has been obtained both in universities and in industry. Outstanding among those are Multics at MIT using PL/1, and the operating systems for the B5500 and B8500 computer. The latter were entirely based on the use of a Burroughs derivative of ALGOL.

Success in the MIT experiment was limited largely because PL/1 was selected at a time when appropriate processors were not yet available. The participants in the Multics experiment are nevertheless convinced that any future system would again use PL/1 [COR69].

The Burroughs experience in the use of ALGOL for systems programming was an unqualified success. Some of this may be attributable to the fact that their hardware architecture was related to the structure of ALGOL. On the other hand, that argument may be reversed. Accepting the desirability and inevitability of using high level languages for systems programming suggests changes in hardware architecture and design so as to expedite those high level instructions or instruction sequences that are frequently executed. This is a trend which is complementary to and in accordance with the trend toward systems-programming languages, standard components and LSI technology.

5.2.2 Advantages and Disadvantages

Opposition to the use of high-level languages has been based on the principle of conservation of execution space/time. It has been argued that programs written in other than assembly language require more core-space and run more slowly. But storage and hardware costs are going down, machine speeds are going up, and the possibilities of hardware implementations of system function has become very real. In addition skilled human resources are becoming scarcer and more expensive. Thus this argument appears fatuous.

Table 4 presents some potential advantages to be expected from the use of higher level languages. Many of these have been demonstrated in practice.

Table 4 Advantages of High-level Systems-programming
(see note 3.10)

PRODUCTIVITY	Rate of code generation substantially independent of language level.
SIMPLICITY OF LEARNING	At least of usable subset
IMPOSITION OF STRUCTURE	
CONFORMITY OF STYLE	Through language conventions
READABILITY	Conciseness, clarity, standardisation of style.
DEBUGGABILITY	Through structure and readability
ERROR GENERATION	Whole classes of error removed
ERROR DETECTION	During preprocessing
CHANGEABILITY	Readability, spreading change, structure.
HARDWARE INDEPENDENCE	In-range compatibility
ARCHITECTURAL INDEPENDENCE	Generation compatibility
SELF DOCUMENTATION	Through increased readability
RELIABILITY	Consequence of all the above.

5.2.3 BSL

Within IBM various independent efforts have sought to develop and introduce high level or systems-programming-oriented languages. In particular BSL [MEL67], has now been in use for some two years in a number of groups, where it has been enthusiastically received and effectively used by those who have made the effort to learn and use it. Thus the language could represent a significant step forward for IBM. Conceptually intended as a systems programming derivative of PL/1, it has, however, developed more into a superset of assembly language with PL/1-like Macros. In particular the language facilitates the descent into assembly language and thereby obviates many of its potential advantages.

BSL has not yet been widely accepted within IBM. The principle objections are seen as essentially invalid at a time when human resources, the lead time of program production, and the growth of expenditure, costs and programming load are critical problems of IBM programming activity. Thus as a first step it would seem that the use of BSL as the main system programming language should be made mandatory for all programming activity as quickly as adequate support can be provided and as quickly as the necessary educational training facilities make its general adoption feasible.

5.2.4 Advanced Systems-Programming Language

BSL is not the be-all and end-all of systems programming languages. There is room for the development of machine independent languages at a higher level than BSL, possibly in association with developments in machine architecture. Such languages are being developed in various places. For example, seven such languages are being developed in Japan [KEN69]. Six of these are augmented-subsets of PL/1, some with, some without, the facility for descent into assembly language segments.

Once again there exists the alternatives of a specific systems-oriented language, or the use of a general language like PL/1. As a first step in this determination more insight is required into the content and structure of operating systems so as to determine what the oft recurring elements are, how they are related, and their interconnection and communication patterns. Thus one needs to analyse existing programming systems functionally and structurally. This can help determine the semantic and syntactic facilities

that are required of a systems-programming language and their functional and primitive content.

The isomorphism that appears between programming languages, standard components and hardware mechanisms has already been mentioned. It is in fact merely a reflection of the design choice which has always been recognised as being possible in principle, and which, in this age of LSI, is becoming a practical reality. This makes implementation of almost any function feasible, though not necessarily economical, in software, hardware or in some intermediate firmware technology. Hardware realisation is essentially interpretative. Thus the isomorphism yields the further insight that software realisation may be executed interpretively or by means of a compilation activity.

5.3 Linguistic Wealth

From generation of a functional concept through to the production of machine-controlling code, the programming process consists of a sequence of text compositions, interpretations and translations. A design decision is taken whenever or wherever in this process, a choice is made between alternatives. This choice may be made explicitly by a human designer, but can equally be implicit in, for example, the selection of a compiler. Such alternatives occur in the selection of languages, primitives, names, structures, sequences, algorithms and so on which, in various combinations, may dynamically achieve superficially identical functional capability confirming to the stated objectives. Cost-performance-wise, and even functionally, the resultant implementations may, however, be very different, for example, in the interpretation of unstated or ambiguous objectives.

Richness is a feature of present languages. They all possess many synonyms, near synonyms and synonymic syntactic structures. It is the wealth in this and the algorithmic area that makes automation of the software design process so difficult. Equally it turns each participant in the process into a designer.

Hence the reduction of linguistic redundancy is a prerequisite to the extension of automation in the programming process. Equally it is required for the development and utilisation of technician-like skills in an area now almost exclusively populated by professionals.

In summary the total programming process is rich and redundant. There are alternative formulations, languages, algorithms, processors, implementations, structures, sequences and expressions. Functional and performance objectives are rarely self evident, unique or completely defined. Thus every programmer becomes a coder and every coder a designer. This is an encumbrance when the prime need is to get an acceptable program running and out into the field. Imposition of linguistic conventions and restrictions is one of the steps that can improve the situation.

5.4 Formal Specification (3.13)

The third topic to be addressed in the language area is that of formal specification languages. This too is a topic which has been widely discussed over the past few years [COD67] though progress has been disappointing. It would appear that one of the main reasons for the relative failures has been that the specifications so produced were not, in general, machine processable. In other words formal specification was viewed as means for communication between humans rather than as an input to a machine process.

A specification must address function, performance, structure, algorithm, test procedures and test-cases. Development is needed in all these areas. Languages such as PL/1 and APL/360, and techniques such as decision-table functional definition [DRI68] and program algorithm specifications must be evaluated in relation to the specification and initiation of a programming activity and its ultimate objective. In terms of the *total automated process*, *formal specification is the first step* (3.14) and represents an area for research and development activity.

5.5 Languages - Summary

The linguistic area is clearly one that plays a major part in the development of the programming process. Many areas of potential research and development activity have been considered in the course of this study. A common starting point for all areas of development is the creation of program-system-models that aim to reveal the semantic and

3.13 (Eds) *Formal specification is one of the buzz-words of today's search for advance in Information Technology. Yet the need and potential though not its role in verification was recognised over 15 years ago.*

3.14 (Eds) *Italicised* 1984.

syntactic structure and primitive content of existing programming systems at various levels. The analysis would aim to identify primitive elements and syntax for programming systems and languages and future hardware structures, components and their interfaces.

As an interim measure it also appears desirable to define restricted sub-sets of existing languages, for use by high school graduates and non-professional personnel. The objective here is the development of a class of specialist skills identified with programming technicians and support personnel.

There exists also a need to develop manipulatable, machine executable, specification languages that permit the complete, formal, specification of programming systems, performance objectives, test procedures, and test cases.

The processors that compile and otherwise manipulate the programming texts are themselves an intimate part of the area. Thus one must also include the development of translation techniques and the relationship between languages and the hardware on which they run as an area of concern within the present framework. In particular with the ever growing trend away from using machine (assembly) languages for program composition, a prima facie case can be developed for a total re-examination of machine language attributes.

Present machine languages are relics of an age when the main requirement was related to usability. In the future the fundamental qualities of machine languages will relate to the ease and reliability of design and change of both languages and compilers or other processors, the efficiency of execution of the processors, and run-time efficiency. At the other extreme machine language designers must consider implementation technologies such as LSI within a total system environment. The latter makes table-look-up for example, as primitive and vital a function as division, say, and hence a candidate for implementation as a machine instruction.

6 Management

6.1 Present Practice

In view of the magnitude, complexity, duration and changing objectives of large scale programming activities, hierarchical project management plays a vital role. The first

attribute of the present procedure is its linearisation. The sequence includes market justification, architecture, specification, design, implementation, integration, test and so on. Each of the sub-processes has assigned to it a manager or management group which takes over from preceding groups, initiating activity which at that stage becomes their responsibility. Overlapping responsibilities do occur and the need for communication between groups is recognised but, even so, often left to chance contacts.

When a new system is being planned, the first step tends to be the assignment of prime management responsibility to one individual. In the case of a large system, an initial breakdown into a number of partitions or subsystems is followed by the appointment of managers to each of these. This is followed by the allocation and structuring of the total resources to be applied to the project in accordance with the judgement of the management hierarchy that has been created.

In this procedure the emphasis is on the creation of a management system structure and of subsystem interfaces. The content and boundaries of the activities of a group will be controlled by the manager's interpretation and judgement, and as a direct consequence of his negotiation with the management hierarchy and his peers. Moreover as the various phases of a total activity are completed, the management node responsible for each particular phase is dissolved and its personnel reassigned. This removes continuity and assignable responsibility from the process.

The consequences of this procedure are apparent. Communications within a group, and more importantly, between different groups, tend to be random and a matter of chance. Personal relationships between individuals exert a strong influence on final system structure, distribution and content. Optimisation, if any, is local within each group. Thus the system becomes an assembly of its parts, amorphous, redundant and with random, largely invisible, communication. Attempts to debug, improve or enlarge the system become very difficult tending to cause its collapse. Moreover, since reasons for decision are not normally documented and the organisational structure is constantly changing, corrective action following the appearance of weaknesses or faults is a major organisational and technical problem. All decisions will be primarily based on time and space-local considerations.

The preceding analysis may appear pessimistic and exaggerated. Some details may be wrong, others omitted. The history of IBM software projects, the fact that systems have had to be massaged into shape over a series of releases, rather than designed and implemented as a finished project, suggests that in overall effect the analysis is reasonably accurate.

6.2 Dynamic Management Structure

There exists an alternative to the above procedure. Within the framework of the total programming process the responsibility for a total system through all the phases of its specification, production and tests may be given to one management node. Initially it will be given to one individual who initiates a dynamic breakdown procedure that identifies, develops and evaluates system function and implementations at ever greater detail. As the content and potential activity increases, personnel are added. As system structure is created, managers take over the responsibility for subsystems, sub-subsystems and so on.

As each group completes its assignment, continuation of the activity is assigned to the same group or to others more expert in that particular area. The process becomes the driving force with managers allocated to the nodes of a tree-like structure as the possibility and need arises. Thus all human and automation resources, including the managers, are seen as active elements of the process. They are allocated specific tasks and at the appropriate time leave the operation or receive new assignments. Notice that the tree structure grows and shrinks only by changes at its extremities. Thus while individuals will join and leave the project, structural continuity with the past is maintained.

This idealised description of dynamic management remains to be developed and demonstrated in practice. We believe that it can lead to a structured system whose elements and sub-elements are likewise structured. Interfaces can be standardised and communications between the subsystems and sub-subsystems forced into the general structure through the adoption of appropriate conventions. The consequent system may be redundant since any structure requires additional components to shape and support it. This redundancy, however, is the price paid for the advantages of structure, particularly in the areas of continuity, changeability, growth power, teachability and reliability.

This approach to management depends for its implementation on the development of a total-system oriented methodology. A related problem is the development of a superstructure that correlates the management hierarchy of a particular project with other projects going on simultaneously within the same organisation. Direct extension of the dynamic concept appears possible. Clearly, however, the present discussion represents an initial indication of the preferred direction of thrust in developing systems-development management. It does not in itself present a solution to all the problems which will be encountered and resolved through practical experience.

7 Education

The concepts and problems of education permeate this entire report. The first section suggested that shortages of both manpower and educational resources are important consequences in IBM customer offices and, therefore, by IBM. Thus IBM must examine the possibility and consequences of expanding its educational facilities and the reservoir of people from which it draws its supply of programming associated personnel. Equally it requires an examination of student selection mechanisms, to ensure a much higher level of success in basic programming courses. Currently the dropout rate may be as high as 33%. An examination of course content is needed both to achieve a higher percentage of student survival and so that students may in their subsequent activity apply a far higher proportion of that content. The last comment refers to an observation in real life that many programmers use no more than, say 10% of the total capabilities of the systems or languages to which they have been exposed. Moreover their colleagues and management have no way of knowing or controlling which 10% they use.

The practical absence of programming technicians is closely related to these educational problems. The recruiting, education and absorption of people who may not have had the same breadth of education at the college level but who, as demonstrated in practice can acquire programming skills, has a very definite place in the process as now practiced. It will find more application in the more developed process visualised in this report. Reference has been made in the language section to the need to develop subsets of existing languages. This too is implied by the need to recruit, train and successfully employ programming technicians.

This report will not discuss the nature or content of syllabuses, the problem of recruiting and selection, the transition from training to productive activity, continued on-the-job training and education, morale problems, and so on. However a discussion of the programming process would be incomplete without reference to the very fundamental part that education plays both in solving present problems and in future planning and development.

8 Conclusion

The preceding sections have ranged far and wide over the problems and the potential of the programming process. The discussion has been in terms of IBM's internal position and has tended to concentrate on those areas in which, in the author's judgement, the Research Division might make a contribution in the future. It can, however, be generalised for the world at large.

The programming area is clearly one of growth in terms of complexity, the resources it absorbs, the amount and nature of the work that needs to be done, the critical performance judgement applied to products and the increasing reliance that society places on its output. The problem has been brought to the fore by a rapid increase in expenditures, manpower requirements, and in the cost of producing programs. The growth which is potentially exponential in nature has so far been controlled essentially by brute force.

As one indicator of growth, Table 1 in section 1.3 referred to the growth of OS/360. Figure 5 represents a plot of this data extrapolated over the next five releases (3.15). This projection suggests that OS/360 will reach 5000 modules by release 21. Forty-three percent of these modules will be changed, on the average, in each of these five releases so that in going from release 20 to 21, for example, a total of over 2200 modules will have to be handled. More precise prediction is not possible since linear, eyeball and quadratic extrapolation yield different trends. However we may expect this indicator of complexity not to further decrease significantly.

3.15 (Eds) *The reader may wish to compare these 1969 forecasts with actual observations reported subsequently, as reproduced in later chapters and in particular in Chapter 18.*

The solution of the programming problem is seen in the development of a total system-oriented methodology and the associated support technologies and tools. The reality of today's situation is that support activity in the programming area is woefully lacking as in Figure 6, despite the Company's traditional encouragement of such activity in other areas. To some extent this may be due to a lack of conviction that success is around the corner. Equally, however, lack of support is also due to a failure to appreciate the very real need for research and development in this area, the very real potential that exists, and the very real payoffs that will result when the work meets with success.

The present study has resulted in some practical proposals for programming process oriented R and D projects and some recommendations for further action. Their implementation will require active management recognition of the needs, potential and possibilities that now exist.

The impact and profitability of support activities is always harder to appreciate than development work expended on a marketable product. The manager faced with the daily problems of meeting a deadline will always first abandon methodology and systematics. It thus requires a very real and demonstrated conviction by management to achieve an 'industrial revolution' in the programming area.

Moreover, we should comment that any work in the methodology, tool and language area has a direct impact on the market place. We have implied that reduction of costs and of lead-times, and increased functional capabilities are the user-observable consequences of activity in the support areas. More directly IBM must be concerned with evaluation of other organisations' measurements on IBM products and configurations, and the provisions of its own data. It must refine its ability to accurately predict the user observable performance of announced products. Finally, it has already been demonstrated that the tools themselves are marketable, and that these in turn open up large new industry areas for hardware products.

The entire computer-community and the nation as a whole has as deep an interest as IBM, in progress, in systems and programming engineering (3.16). However, the problems that

arise are of individual systems, the concurrent development of several large systems, or the matching of much software with much hardware, IBM's problems are by far the largest. Thus it is only right, as well as being in the Corporation's self interest, that IBM make a major attempt to achieve a breakthrough in the technology to which the Company owes its prosperity. In the long run computers without software are practically useless outside the universities. Only a major breakthrough will permit this company to continue to offer and expand the functional and software support to its hardware products. There are strong indications from the state-of-the-art that the area is ripe for such a breakthrough. What is now required are resources and dedication.

9 Acknowledgements

In the course of the study on which this report is based, I consulted with more than 50 people within IBM and a number of people outside the Corporation. It is a pleasure to acknowledge the unfailing courtesy and eagerness to help experienced in all these many contacts. While I accept entire responsibility for the viewpoints expressed, it is clear that I could not have learned so much about the programming process, nor could I have made my proposals, without the many contacts and suggestions received in the course of my conversations (3.17).

- 3.16 (Eds) *This report was written after the Garmisch conference, but the author was not aware of the conference, had not seen its proceedings [NAT68] and had not heard of the then newly coined term 'Software Engineering'.*
- 3.17 (Eds) *On re-reading this paper 15 years after it first appeared, one can only express regret that it did not even raise a ripple, a modicum of interest at the time of its publication. Its observations and conclusions are as timely and relevant today (1984) as when written in 1969.*

CHAPTER 4

NATURAL SELECTION AS APPLIED TO COMPUTERS AND PROGRAMS* (4.1)

0 Introduction

From time to time, a programmer decides to rerun an old job and finds it will no longer run. Computing centers sometimes discover that the older its computer gets, the more difficult it is to get new jobs to run properly. One person tries to run another's programs and finds that they just do not work in a different installation. All of these well-known difficulties, and many lesser known ones, spring from a single source - Natural Selection taking its unswerving and irresistible course, just as it does in the kingdom of living things. In this paper, I propose to show how Natural Selection produces these undesirable effects and to suggest what can be done to diminish some of them.

1 Conditions for Natural Selection

In 1859 Charles Darwin started a scientific earthquake whose after tremors are still being felt today. In 'On the Origin of Species', he set forth the conditions under which living systems undergo changes which adapt them to an extraordinary diversity of environments. To quote his words [DAR64],

'Owing to this struggle for life, any variation, however slight and from whatever cause preceeding, if it be in any degree profitable to an individual of any species, in its infinitely complex relations to other organic beings and to external nature, will tend to the preservation of that individual, and will generally be inherited by its offspring. The offspring, also, will thus have a better chance of surviving, for, of the many individuals of any species which are periodically born,

4.1 (*orig*) *This article was submitted for publication in 1967, was lost, and rediscovered two years later. The author has stated that he was able to resist the temptation to modify what he said a few years ago. - The Editors.*

but a small number can survive. I have called this principle, *by which each slight variation, if useful, is preserved*, by the term of Natural Selection, in order to mark its relation to man's power of selection' (p 61; emphasis added).

In the tumultuous development of Darwin's ideas in the following century, Natural Selection has been revealed as a phenomena not confined to 'living' systems, but explainable in purely abstract terms. All that is necessary is that a population exists under the following three conditions:

- (1) Its individuals are capable of making reasonably exact copies of themselves.
- (2) A certain amount of inexactitude is present in the copying process.
- (3) An environment exists which selectively favours certain variations.

Requirement (1) is called 'reproduction'; (2) is called 'variation'; (3), 'selection'. All three must be present for Natural Selection to take place; and when all three are present, Natural Selection must take place. The population must increase in 'fitness' - and at a rate which can be determined mathematically if the parameters are known. That variation is a 'chance' process has nothing to do with the inevitability of Natural Selection. As R A Fisher [FIS58] so ably put it:

'The income derived from a Casino by its proprietor may, in one sense, be said to depend upon a succession of favorable chances, although the phrase contains a suggestion of improbability more appropriate to the hopes of the patrons of his establishment. It is easy without any profound logical analysis to perceive the difference between a succession of favorable deviations from the laws of chance, and on the other hand, the continuous and cumulative action of these laws. It is on the latter that the principle of Natural Selection relies' (p 40).

We must note, however, that increase of fitness of a population is not always a 'good' thing for Man. The rat population is constantly increasing its fitness with respect to a largely human environment, and the progressive adaptation of certain bacteria to penicillin and other drugs is an unending source of potential disaster to Man. Thus, although Man participates in Natural Selection of rats and

bacteria, he does not, in a certain sense, 'direct' the process. In order to distinguish this process from the process directed by Man for his own benefit (Selective Breeding or Artificial Selection), Darwin coined the term 'Natural' Selection. About the relative power of two methods, Darwin went on to say:

'But Natural Selection, as we shall hereafter see, is a power incessantly ready for action, and is as immeasurably superior to man's feeble efforts, as the works of Nature are to those of Art' (Darwin, p 61).

We can abstract from the literature on Natural Selection two laws which we can use in their qualitative form to predict certain consequences of the ways we use our computers. The first of these laws is the existence theorem of Natural Selection:

'Given the conditions of Natural Selection, the fitness of the population will increase with time.'

The second law comes under various names, but we shall refer to it as the Law of Evolutionary Potential. Because a population must show variation in order to undergo Natural Selection while at the same time it must reduce variation in order to be well adapted to particular environment, the second law results:

'The more adapted a population becomes to a particular environment, the less adaptable it is when faced with other environments.'

2 The Computer as the Adaptive Population

For our first case, the population undergoing Natural Selection will be the population of components in a single computer. 'Components' can be taken to mean the lowest level units which are subject to replacement. In some cases these might be individual relays, vacuum tubes, resistors, capacitors, or transistors; while in other cases they may be coordinated sets of such parts, as are found in circuit cards and integrated circuits generally.

The first difficulty we face in applying the theory of Natural Selection to this 'population' is that it does not 'reproduce' in the ordinary biological sense of that word. We overcome this difficulty in the following way.

- 1 We pick an arbitrary time interval, T , which will be thought of as the generation time of the population.
- 2 At the end of each interval, T , each element of the population is imagined to 'reproduce'. If T is short, virtually all of the 'offspring' will be identical with their 'parents' - for, indeed, they will be the same component.
- 3 The other way, in which an offspring may differ from its parent, is by 'spontaneous' change in the performance of a component. This change may be entirely undetectable from outside the computer, but we know that such changes are always taking place.
- 4 Selection takes place because not all changes in the components (point 3) are equally detectable by the engineers who are trying to 'maintain' the machine. When a component passes into a state ('produces an offspring') which is both detectable and undesirable, it is replaced ('dies'). Thus, the engineers and their diagnostic programs provide a selective environment which is constantly at work to remove certain types of individuals from the population. The environment is, indeed, selective, because all changes in states of components are not equally detectable by the diagnostic procedures of the engineers.

Viewed in this way, the evolution of the population of components in a computer is truly governed by Natural Selection - 'Natural' because the engineer is not *trying* to favour the errors his diagnostic programs do not detect any more than the doctors are trying to favour certain bacterial varieties over others through the use of penicillin.

Once modelled in this way, the qualitative behaviour of the system is entirely predictable by our first law:

'Fitness of the population will increase with time.'

In this case, however, fitness of the population is measured by the ability to escape the probings of the diagnostic procedures. From one point of view, this result says that the computer will remain fit - insofar as fitness of the whole is related to the fitness of the parts - to perform the diagnostic programs correctly. Our second law of Natural Selection, however, says that the computer will become increasingly adapted to *just* that environment, through the

accumulation of undiscovered states which will not affect the current set of programs but which might affect some other set.

Some specific instances will be useful here. The first case of this type with which I ever became involved is typical of many situations reported to me by my students. A certain petroleum company had been using a computer for approximately four years on one rather complex application - oil royalty accounting. At that time, a group of chemical engineers in their laboratory became interested in using the computer for matrix calculations. After studying the manuals, they wrote their programs, punched their cards, and wired their control panels (as was necessary on this machine). Nobody was too surprised when their programs did not work immediately, for even in those days it was known that programs could have bugs. Eventually, however, the engineers were able to demonstrate that the reason their programs were not working was that certain relays in the computer were not performing according to the specifications in the manual. They were proved correct when, after the customer engineers spent two days bringing the machine up to specifications, their programs ran correctly.

Another example of this type of trouble was the case of an installation (A) which was endeavouring to use a system supplied by another installation (B). As commonly happens in such cases, the new system would not run successfully in the new installation. In most cases like this a modest amount of effort is made to find the trouble, after which the whole project is dropped with some mumbling about 'bugs in the program'. In this case however one programmer was determined to find out explicitly where the trouble was and by much diligent effort eventually discovered a number of *machine errors* which had accumulated in the computer over the years.

It may help clarify matters if two of these errors are examined in detail. The first involved the magnetic tape error routines. At installation B, the common procedure in the case of tape reading errors was to make 20 retrials, while in installation A only 3 retrials were ordinarily made. Over the years, the tape units in A had not been subject to as stringent a diagnostic environment as had B's. Consequently they behaved in unpredictable ways when B's system attempted to backspace and reread them 20 times instead of the accustomed 3. A more careful adjustment of the tape units might have solved this problem, though it was actually done by modifying B's error routines.

Errors in tape units, like errors in relays, are essentially 'mechanical' errors. It might be tempting to imagine that the electronic components of computers are not subject to Natural Selection in the same way, because they are intrinsically more reliable. Unfortunately, Natural Selection is a universal law, and applies whenever the three conditions are met. To be sure, the *rate* of Natural Selection may be altered - as in this case when the rate of variation is reduced - but the process inevitably takes place. Indeed, even though the variation in individual components may be less with electronic components, the situation in general could be well worse since there are many more components. This increase in the number of components not only increases the total amount of variation, but it increases the total amount of variation, but it increases the number of components which are not effectively tested by the diagnostic programs.

The second error found by this enterprising programmer will illustrate that electronic components are equally subject to Natural Selection. In this case, the error involved the failure of the circuit driving one bit of all the words in a particular portion of memory. In all of the programs of Installation A - including the diagnostic programs - this segment of memory was never occupied by anything for which this failure made a noticeable difference. In most instructions, this bit was always zero anyway, and, in fixed point numbers, it was a high order, non-significant zero. The new system, however, happened to have one number for which this bit could be significant and which lay in the erroneous segment of memory. Consequently, whenever that bit of the critical number was supposed to be a one - which was fairly rare - an error resulted.

We are usually not as aware of Natural Selection in computers as we might be, for the troubles it gets us into are often so difficult that we never trace them down. In this last case, for instance, installation A had been experiencing, as most installations do, unexplained difficulties when modifying certain large programs. We can see that an error such as the one bit memory failure could cause inexplicable problems if an instruction in a completely different section of memory were added or deleted, thereby bringing a critical instruction or number into the damaged region. In all likelihood, such an error would not be found directly, but rather eliminated by an equally accidental compensating modification in the course of trying to find it. Such an error remains in the machine but not in the programs.

Through the lifetime of a particular computer, such errors continue to accumulate. As long as no new programs are tried, things may function well. Eventually, however, even slight modifications to existing programs become increasingly difficult to make. Furthermore, newly introduced errors, because there are now so many residual errors to combine with, become increasingly difficult for the engineers to find. Finally, the costs and irritations of using the machines grow to the point where it is simpler to replace with a fresh, unadapted machine - thereby pushing the problems of Natural Selection a few more years into the future.

3 The Program as the Adaptive Population and the Computer as Environment

Whenever two different populations are in interaction so that each forms a part of the environment of the other, they participate in a form of mutual Natural Selection. Bats, for example, develop their hearing best at the frequencies most commonly emitted by the species of moths on which they feed; while the moths develop receptors which are sensitive to the echolocation frequencies of the bats which feed upon them. Nor is it necessary for the relationship to be one of predator and prey; symbiotic and parasitic relationships show precisely the same type of mutual adaptation. Formally, it is easy to see that this must be so, for in such cases, it is entirely a matter of arbitrary choice as to which population is system and which is environment.

With computers, it is probably more conventional to think of the machine as the environment in which the program is run, rather than considering the programs as the environment in which the machine evolves, as we did in the last section. Considering the machine as the environment of the programs, in the conventional way, we get some additional insights.

In this case, we may consider each program as a population of instructions - or microinstructions.

Here, reproduction is taken care of by the successive versions of the programs, and variation is introduced either by intentional or unintentional program modifications. As we have just seen, the computer itself provides an environment in which the programs are developed and which tends to select against program variants which encounter some of the more subtle machine errors present. If the program does not work, we make some changes. If the trouble then goes away, we may

not question further; thus, the program gets even better adapted to the machine on which it is run.

It might seem that no harm can come from this type of Natural Selection because the programs will always be able to run on a machine which does not have an accumulation of errors. An interesting contradiction to this argument is provided by our earlier example of the petroleum company. After the computer was brought up to specifications, the matrix calculations worked perfectly, but the oil royalty programs no longer worked at all! Furthermore, all the king's programmers and all the king's engineers never did succeed in getting that Humpty Dumpty set of programs working again! The eventual result was the replacement of the machine by a new model.

Such extreme cases are not as rare as one might imagine. The most typical situation, reported by many of my students, occurs when some new piece of peripheral equipment is installed on an existing computer. The ensuing difficulties are inevitably attributed to the new equipment, but in those cases where the actual cause is tracked down, it is equally likely to be some error residing in the machine which previous programs had avoided.

4 The Program as the Adaptive Population and the Data as Environment

Programs adapt not only to the computers on which they are run, but to the data which is given to them. I do not mean, of course, the Artificial Selection caused when modifications are made to take care of new data cases which arise, but the Natural Selection which takes place because of the data cases which do not arise.

Although program-to-data adaptation takes place in ordinary data processing programs, the most interesting - and most troublesome - cases occur in programs which use other programs as data. Such programs - compilers, for instance - ordinarily encounter a data environment which is potentially several orders of magnitude more complex than that of ordinary programs. Thus, there are many more unexplored data cases - cases which have never been tried by the program, and many more complex cases - cases which are circumvented rather than analysed and eliminated.

Let us look at some examples. All programmers have had the experience of trying to run a program that once ran and now does not because of a new bug in the operating system. I

recall one occasion in which the interval was six months, and where an error had been introduced into one of the three binary card loading routines available in the system. By the time I encountered the trouble, nobody in the installation could recall that this change had been made. Furthermore, when I had narrowed down the trouble to this particular routine, I was repeatedly advised not to try to find anything wrong with the routine but rather to switch to one of the other two routines. Unfortunately (or fortunately) I could not use the other routines because my data cards were not in acceptable formats; thus, I had to investigate the loading routine itself.

What I found was quite simple. Since the three routines shared some common parts, certain switches were set upon entry to discriminate among them. For the option I was using, one of the switches was never set, and the program thus made a wild transfer of control whenever that entry was used. The existence of such a bug in the system could only mean one of two things: either nobody else had used this routine in six months (this installation claimed to have 3000 users) or those who had used it had been successfully steered away from it by helpful advice.

How many other such bugs were accumulating in this system I have no way to estimate. Programmers who have worked with large systems, however, will recognise the experience of tracing through the wild execution path of one bug and finding one or more other bugs that had never been made manifest before. This kind of experience tends to verify the force of the Natural Selection processes on programs, and long ago led me to formulate the law governing the number of errors, n , remaining in a large program at time t :

'For all t , $n = 1$ '

or, informally, n equals 'one more'.

Compilers, of course, are particularly susceptible to the accumulation of special cases which they cannot handle correctly. I especially recall one assembly language system we were using for writing a large real-time system. Every week we would collect a list of the things that did not seem to be assembling properly and send it to the maintenance crew. Every week we would get back a reply which said in effect that nobody else seemed to be having this particular trouble so it was not worth investigating. They were always helpful, however, in suggesting ways of *avoiding* difficulty -

generally by not using the available language in its full power. Eventually, the accumulation of such cases to be avoided became unbearable, and we undertook the maintenance of the system ourselves. The straw that broke our backs, I recall, was a modification which unwittingly placed a limit on the number of characters of comments at 6×2^{15} . Since we had already far surpassed that number, our system would no longer compile, though 'nobody else seemed to be having that trouble'. We quickly learned that by keeping the size of the system down we could avoid trouble, but it took two months to discover the source of difficulty so that we could really proceed with our work. If we had not persevered, it might have been years before somebody else encountered the trouble; and at that time they would no longer have had the slightest clue about where to look.

5 Speciation

Although Darwin's great work was entitled 'On the Origin of Species', Mayr [MAY63] has pointed out that:

'It is a familiar and often-told story how Darwin succeeded in convincing the world of the occurrence of evolution and how - in natural selection - he found the mechanism that is responsible for evolutionary change and development. It is not nearly so widely recognised that Darwin failed to solve the problem indicated by the title of his work' (p 12).

This is not the place to present the modern view of the origin of species, but it is interesting to us to note that the weight of evidence now points to the splitting of one species into two or more non-breeding parts (usually geographic accident) as the initial event in speciation. The parts then proceed to evolve in their own (somewhat different) environments until they are sufficiently differentiated that they will no longer interbreed even if they are brought together again.

Since we have no process analogous to sexual reproduction in our evolutionary models of computers and their programs, we cannot extend the modern species concept directly. We may, however, utilise the idea of isolating mechanisms, leading to progressively more differentiated populations. Thus, we may predict that different versions of the same system - a FORTRAN operating system, for instance - used in different installations not exchanging programs with one another will become more and more widely separated as time passes. The

separation will be both explicit (through the addition of new features and the deletion of old) and implicit (through the evolutionary mechanisms we have discussed). Thus, the chances that one system will run on the other's computer, or that both systems will compile the same program to do the same things, diminish with time. Rather than moving toward standardisation, then, we are moving toward the state where every computer installation will be an isolated species - unless, that is, some intelligent efforts are expended.

6 Retarding the Rate of Natural Selection

The foregoing arguments and examples have shown how the force of Natural Selection can work against the successful operation of a computer installation. (We do not concern ourselves with advantageous applications.) We cannot eliminate Natural Selection entirely in any of these cases, because we cannot eliminate the necessary conditions which inevitably bring it about. We can, however, do a number of things to slow the rate at which Natural Selection destroys the usefulness of a computer or system of program.

Some of the things we can do are quite obvious. In the case of machine errors, we can reduce the rate of variation by reducing the failure or degradation rate of the components. We can only go so far in this, however, for, as we have seen, this variation is constantly being reintroduced by increasing the number of components in our machines. Furthermore, with more reliable components we get a change in the 'environment', for the maintenance engineers have more difficulty getting experience in finding particular bugs, and diagnostic programs have to be much more complicated just to keep up. Consequently, it becomes more and more tempting to modify the programs so that they will avoid the bugs rather than eliminate them. This amounts to buying present convenience for future disaster - a sort of anti-insurance policy.

Another policy which can be adopted to retard the rate of undesired Natural Selection is to refer to external standards. For instance, rather than using tapes written in the installation to test the tape reading mechanism, tapes from a standard outside source should be used. If this is not done, the installation is in danger of having its tape readers adapt to its tape writers - drifting further and further away from compatibility with any other installation.

The external standards, of course, are not limited to mechanical ones. For compilers, for instance, there should

be a large set of standard programs which they must compile and execute to produce standard results. The mere existence of such standard programs is not enough, however, for as long as they are not the installation's own programs, errors they reveal are not likely to be treated with the proper respect. Getting the day's work done always takes precedence over keeping the 'equipment' in good working order. Only by determined inculcation of certain values can programmers come to the state where they view errors in test programs with the same panic they now use for errors which halt daily production.

Another way in which the progress of Natural Selection may be retarded is by making the diagnostic programming a continuing process, not merely a one-shot job to be done before a machine is *first* delivered. A continuing stream of new diagnostics will have the effect of constantly shifting the environment in which the components have been evolving, ensuring on a dynamic basis that each machine remains rather close to the 'standard' machine in the diagnostic programming shop. Again, however, the maintenance engineers are likely to revert to the old diagnostics if the new ones cannot be made to run without undue difficulty.

The avoidance of diagnostics may be prevented, in some situations, by pushing the diagnostic programming to a microprogrammed level, which cannot be reached by the ordinary problem programs at an installation. In this way, error states of the system with respect to the diagnostic programs are automatically more numerous than error states with respect to the problem programs. Thus many errors may be detected before they reach the problem program level. Such errors may be removed before they become part of the environment to which the problem programs adapt. For instance, if the storage devices of the machine have built-in error-correcting codes, the engineers become aware of component failures before they can affect the programs of the installation. Under such a system, and undetected bit failure in some small section of memory could never be retained.

One modern trend in computer use should have a beneficial effect on reducing the rate of accumulation of deleterious states in both software and hardware, namely, the trend to multiprogramming and multiprocessing. For instance, in a multiprogramming environment using dynamic relocation, a problem program would not always occupy the same locations in memory. Thus a portion of memory would not be likely to

become partly inoperative for a long time without detection by some program or other. The same argument would apply to the use of peripheral components, which would be dynamically assigned by the supervisory program and thus subject to a wider variety of environments in a given period of time.

These observations lead us to one final suggestion for slowing down the rate of Natural Selection. This suggestion has a paradoxical aspect which is familiar enough to biologists, but which might give computer specialists a hard time. Natural Selection causes most difficulty in computer installations because the computer and its programs become adapted to a narrow environment. Thus, when some new thing is finally introduced, the installation is unlikely to be adequately prepared. Indeed, it has most likely been accumulating debilities which will suddenly all become manifest at once - perhaps to the destruction of the installation. The lesson here is the lesson of the Law of Evolutionary Potential.

But the Law of Evolutionary Potential gives us another way of delaying the very death it predicts. By keeping up the variation in the system's environment, we make it less likely that it will get locked onto too narrow an environment. This observation leads us to expect that in those installations supporting the most diverse uses of the computer (not just those with the most users, who may be doing the same type of thing) the buildup of Natural Selection difficulties will be less severe than in those which support only a few relatively stable applications. Furthermore, it even suggests that those frolicsome programmers who sneak in and try insane things with the computer at night are really doing the installation a great service. Perhaps they should be encouraged - if encouragement will not discourage them. Perhaps with computers - as with people - the way to stay young is through play.

CHAPTER 5
PROGRAMMING SYSTEM DYNAMICS
OR
THE META-DYNAMICS OF SYSTEMS
IN MAINTENANCE AND GROWTH*

1 Growth Dynamics

Any large and complex system possesses two separate and distinct dynamics. That most commonly studied is *Activity Dynamics*. In the context of programming systems this represents the behaviour of the programs in *execution*. We shall demonstrate that one may also usefully consider the *Growth Dynamics* of a system, its change with time. For a programming system, this meta-dynamics studies the *body of code* with its statements, variables and structure, and the people and organisation maintaining it. Note that in growth dynamics, the code, which in activity dynamics is viewed as the active component and the source of change, becomes static with the active role being taken over by the (human) environment.

It is common experience that large programming systems continue to need correction and to grow. Repairs are made, existing function is modified, new function defined and added, implementation changed to improve performance and reduce resource requirements. In general, such systems never achieve stability, certain freedom of bugs and of the need for addition or change.

Faults that are uncovered in successive generations fall into various classes. There will be faults or weaknesses that have been in the system since time immemorial but have remained undetected because the particular circumstance required for their activation have not occurred. There are those in the most recent additions in the code. There are bugs in all parts of the system due to changes (or omission of changes) necessitated by repairs or by additions to the code.

There will be errors that remain undetected because they have been hidden (eg a fault in a section of code that could not

be entered as a consequence of a bug that prevented branching to that section of code). Faults will appear when new system configurations are used, new hardware connected or new applications are run. Finally, for each of these classes there will be descendant faults arising from mistakes in the earlier re-definitions, fixes and changes.

The term 'fault' and its various synonyms are used here in their widest sense to include *anything that causes the system to deviate from what is required or desired for acceptable performance*. Faults may be due to a semantic or syntactic errors in the code, misinterpretation of hardware or system specifications, logically inconsistent specifications, poor design or implementation, environmental mismatch, changes in usage or environmental objectives, operational inexperience and so on. Each fault will in general be a cause, at some later date, for corrective effort. Thus the number and rate of appearance of faults and the amount of work required to maintain a system are directly related.

The present paper assumes these facts of life to develop a model of system change, a *macro-model* in the thermodynamic or economic sense. It represents the system as a growing body of code within the human environment that changes it. The model portrays the growth trends of system measures of size and complexity as functions of the effort or work that must go into the system for its maintenance and further development. We shall also outline, but not formalise here, a *micro-model* that is to relate work required for system maintenance to its content, structure and complexity.

The models discussed have already been related to the observed growth statistics of some present systems. They may thus be used as a planning tool and to test the viability of maintenance and development strategies. They also yield considerable insight into the way that the consequences of change spread through the system, so as to ultimately cause an ever increasing amount of work to be required for system maintenance. The maintenance workload appears as one that decreases during early releases but then suddenly increases explosively. Interpretation of the models suggests how this phenomenon may be brought under control. In particular, appropriate *structuring* of the system, its documentation, the project, its management and all communication would greatly enhance maintainability and growth properties and extend the lifetime of large, complex programming systems.

Most of the discussion in this paper does not make any use of concepts specifically related to programming systems. Thus the phenomena discussed and the models developed, when re-interpreted, apply equally to other large systems of people and/or machines, underlying the progressive decay that may be observed in many areas.

2 The Axiom of Distributed Effort

When a system is first implemented, it already contains many of the seeds of future effort (and ultimate decay). As stated in section 1, the need for continuous effort arises in many different ways. Faults will appear and require repair. Re-design or improved implementation will be required to remove operational inefficiency. Shortcomings that make the system difficult to learn or clumsy to use must be overcome.

The need for the necessary corrective action will not appear at once. Intrinsicly it cannot all be detected and corrected in the first period of system development and usage, if only because certain faults shield one another. Others become apparent only as the sophistication of system usage increases with time or as hardware changes. Hence we state that from the original implementation of any system there arises the need for a (partially ordered) sequence of activities that can only be carried out over a long, perhaps infinite, period of time. That is, a software system will age, with new faults appearing, apparently as a function of time.

The resultant activity sequence to repair the faults will, in practice, be partitioned so as to produce a series of releases or system generations. Each release defines the end of a time period during which a manageable amount of corrective activity has been designed, implemented, tested and documented.

We represent the contents of the system and the effort involved in its original implementation (release one) by S_1 and define the corrective action, the amount of work executed on the *first* release during the *first* time interval to achieve release *two*, by:

$$W_{12} = f_{11} \cdot S_1 \quad (1)$$

that is some fraction f_{11} of the original effort. This is indicated in figure 1a and 1b by the ordinate at one. The assumption here is that the amount of activity undertaken is

proportional to the size of the undebugged portion of the system, yielding a geometric distribution of effort. Hence, the amount of work or effort between releases 2 and 3 to correct for the work S_1 originally undertaken in release 1 is:

$$W_{13} = f_{12} (1-f_{11}) S_1 \quad (2)$$

being proportional to the size and content of the system remaining faulty in release 2. Similarly, for release 4, the amount of work due to the original implementation, S_1 , can be represented as:

$$W_{14} = f_{13} (1-f_{12})(1-f_{11}) S_1 \quad (3)$$

and, in general, the work executed in the i th interval is:

$$W_{1i+1} = f_{1i} \sum_{k=1}^{i-1} (1-f_{1k}) S_1 \quad \text{for } i > 1 \quad (4)$$

This process may be pictured as in figures 1a and 1b. The convex down curve of 1b constructed as in figure 1a represents the decreasing amount of work per release resulting from correcting and amending the *original implementation*. The vertical lines indicate the arbitrary release points and the area under the curve measures the effort or work required in the following release interval. The concave down curve of 1b on the other hand, is interpreted as representing the health of the system; the extent to which its contents are fault free and do not require change relative to the *original definition of the system*. Without loss of generality we have, for the moment, assumed that releases are equispaced.

From section 5 it will become apparent that it is reasonable to consider the fraction f_{1i} that determines the work in the i th release period to be constant, that is:

$$f_{1i} = f, \quad \text{for } i \geq 1 \quad (5)$$

Hence, in the i^{th} interval from the i^{th} to the $(i+1)^{\text{st}}$ release, the work to be done due to the upgrading of the *original implementation* S_1 is:

$$W_{1i+1} = f(1-f)^{i-1} S_1 \quad \text{for } i \geq 1 \quad (6)$$

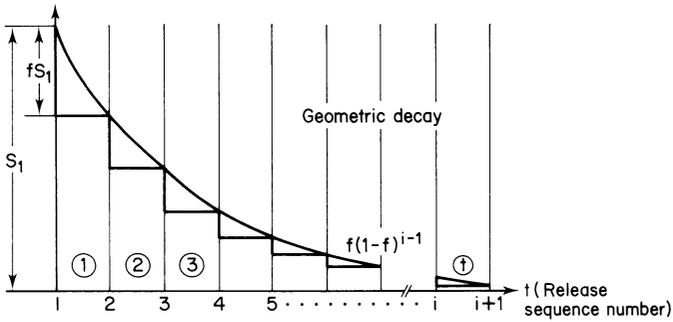


Figure 1a Geometric Decay

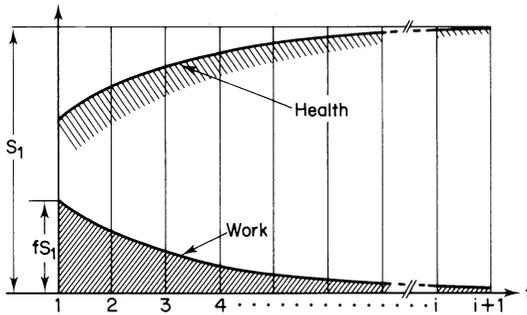


Figure 1b Corrective Activity on Release 1

Recursive decay

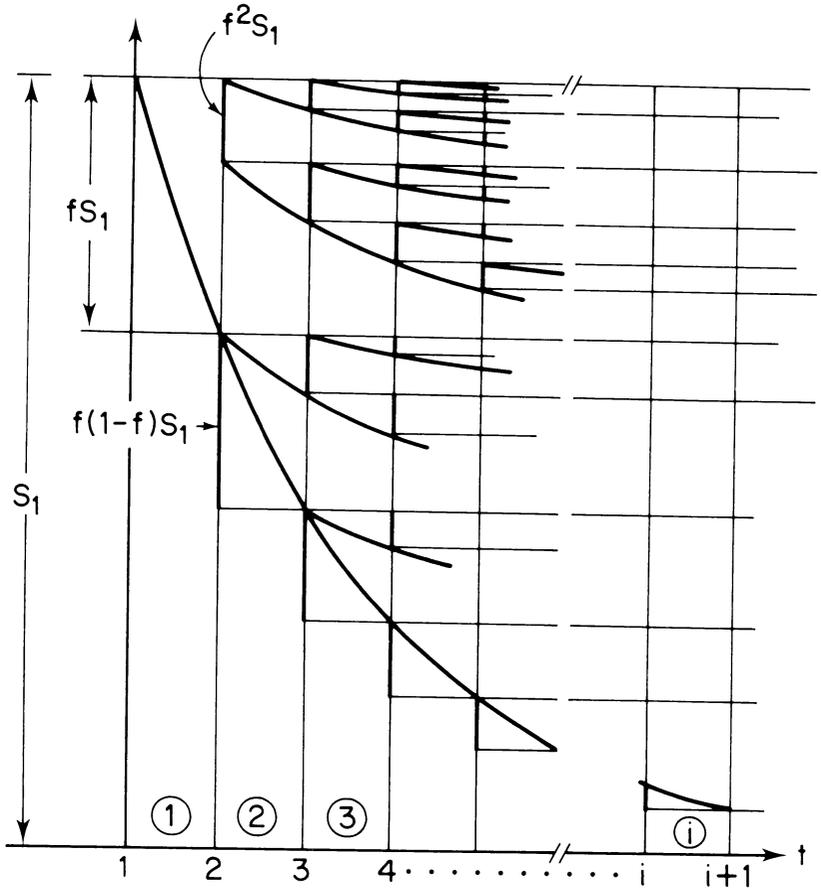


Figure 2 Binary Activity-Tree

Each W_{1i} represents a programming effort of the same nature as the original effort S_1 , though smaller in magnitude. Hence it too contains faults and therefore generates a new sequence of corrective activity. This will be true for all activity on the system. Thus there arises a binary activity tree as in Figure 2. In general in the i^{th} release interval there will be 2^{i-1} independently rooted activities. We define a measure W_i of total activity required in the i^{th} interval to represent the sum of these terms. Applying assumption (5) to (6) and summing we obtain:

$$W_1 = f \sum_{k=0}^{i-1} \binom{i-1}{k} f^k (1-f)^{i-1-k} S_1, \quad 0 < f < 1 \quad (7)$$

$$= f S_1$$

since the value of the binomial sum is 'one' for all values of f as from Table 1.

Figure 3 illustrates the fact that the activity is composed of rapidly increasing number of activities with distributed paths and roots. We shall return in section 6 to consider the effects of the resultant stratification of what has been shown here to comprise a constant level of activity fS_1 .

3 The Effects of Experience

In practice the people implementing and maintaining a system continuously improve their skills and their familiarity with the system. There will therefore, be some gain in efficiency over a series of releases, a reduction in the effort required to complete the repair work selected for each release interval. If, despite the resulting greater productivity, the amount of repair activity (arbitrarily) selected for a given release, is not increased, the sequence of f_{1i} in equation (4) would then be monotonically decreasing. The consequences of experience may, however, equally be imposed on the total work defined by (7) by means of a negative exponential, two parameter, multiplier, so that (7) becomes:

$$W_{1i} = Ae^{-Li} f S_1 \quad (8)$$

as in Figure 4. Values for the parameters will be determined by the average rate at which experience and familiarity with the system is acquired (L), by the degree and any changes in tooling and other automation (A) and by the fact that we are dealing not with a continuum but with discrete activities.

Table 1

Interval (sequence number)	Simple term (equation (6))	Additional terms for release $i (i > 0)$ due to repair efforts in all preceding intervals. (Equation (7)). (# terms) = 2^{i-1}							Sum W_i / S_1
1	f								f
2	$f(1-f)$	f^2							$f[(1-f)+f] = f$
3	$f(1-f)^2$	$f^2(1-f)$	$f^2(1-f)$	f^3					$f[(1-f)+f]^2 = f$
4	$f(1-f)^3$	$f^2(1-f)^2$	$f^2(1-f)^2$	$f^2(1-f)^2$	$f^3(1-f)$	$f^3(1-f)$	$f^3(1-f)$	f^4	$f[(1-f)+f]^3 = f$
.									
.									
.									
i	$f(1-f)^{i-1}$	$f^2(1-f)^{i-2}$. . .					f^i	$f[(1-f)+f]^{i-1}$
.									
.									
.									
									$= f \left[\sum_{k=0}^{i-1} \binom{i-1}{k} f^k (1-f)^{i-1-k} \right]$ $= f$

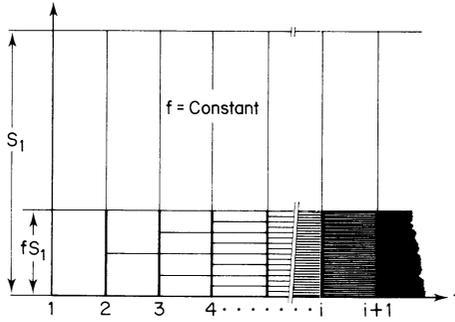


Figure 3 Constant but Stratified Activity

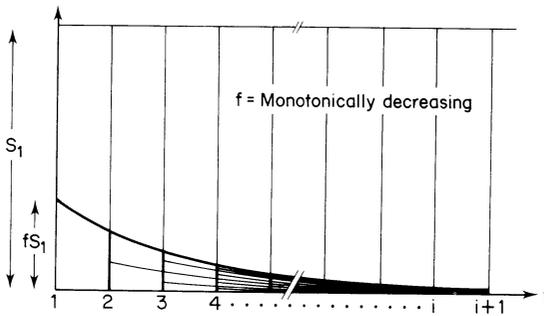


Figure 4 Decay due to Experience

Thus effort fragments falling below some activity threshold disappear, and the number of strata is reduced below 2^{i-1} .

4 New Function

The repair activity considered so far has all been a consequence of the original implementation S_1 and subsequent repairs. In practice one invariably also wishes to add additional capability to the system as it ages. The resultant programming activity is of the same nature as the original and can be represented by a series of curves similar to Figure 4, but of reduced amplitude, superimposed to yield the activity pattern suggested by Figure 5. Thus we now *redefine* W_i to represent the *total* activity in the i^{th} interval including that required for the addition of new function and the repair of all function created in the first i releases. That is:

$$W_i = \sum_{k=1}^i A_k e^{-L_k(i-k)} f S_k \quad (9)$$

It is easily seen that this total activity is stratified into $2^{2(i-1)}$ elements.

5 The Management Environment

In the real world the work requirement outlined in the previous section is performed under manpower and budget constraints. The workload of a succession of releases, including both fixes and addition of new function, is planned to fully and continuously utilise anticipated available resources, human and machine. This effect smooths the sawtooth representation (Figure 5) and produces a work curve which will either stay constant or change to follow the contours defined by budget allocations $B(i)$ (Figure 6).

The time integral of Figure 6 can now be interpreted as the expended effort or cost (of people plus computers) with the area under the curve between ordinate pairs representing the cost (eg in dollars) of the release. Furthermore, the abscissa or time axis does not represent real time, since efforts on various releases (unfortunately) overlap, resources may be moved around to cope with the most urgent tasks irrespective of their release association and releases are intentionally not equi-spaced. Thus we relax the previous assumption of equally spaced releases. In fact, it is well known from practical experience that release dates

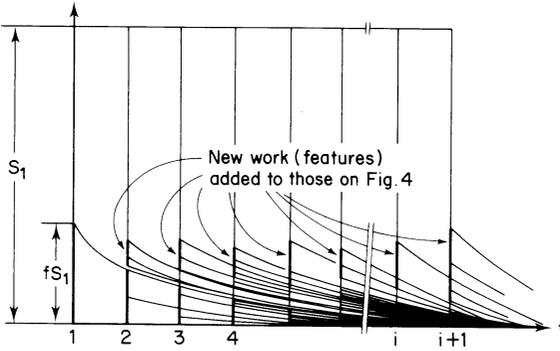


Figure 5 The Addition of New Function

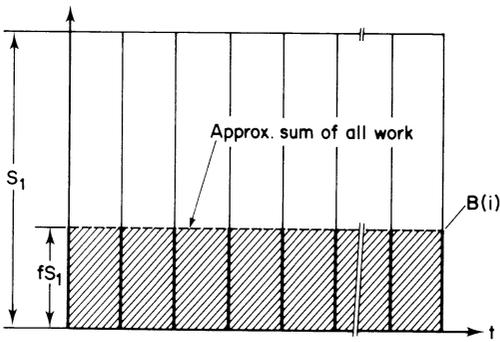


Figure 6 Budget Controlled Work Curve

cannot, in general, be precisely predetermined because of the unpredictability of a rapidly growing work-load.

6 The Effect of Complexity

The previous sections have developed a model of the primary effort required to develop and maintain a system. It is determined by the need to fix faults and provide new function that will enhance the system. In practice the level of effort that can be applied is determined by available manpower and budget appropriations. The latter must cover the cost of people and of their support, including computer costs for testing, for programming support and for documentation. Since a single budget covers the cost of both repair and enhancement, unforeseen growth of the former must lead to retrenchment in the latter.

The total effort required is stratified. The roots and root paths of the various activities undertaken in any release are dispersed in time over previous releases, in space over the system, managerially over many different groups and individuals and geographically over different locations. That is, the generations of *people* that implemented the system and the *records* that document the original design, its subsequent history, its functional content and underlying structure, the nature of each fault, the proposed repair and so on, are also scattered geographically and over the lifetime of the system. Nevertheless, for each separate activity, any relationship to other activities must be noted and acted upon. This *correlation* requires communication between the people executing the various pieces of work *wherever* and *whenever* they do it. In addition, it requires each group to check major portions of the system, its documentation and its plans. That is, it introduces effort additional to that considered so far, to avoid the introduction of new faults due to interference effects between different repair and growth activities. If this cross correlation work is not executed, new sequences of faults and correction activity will be started up due to the fact that changes, required in one activity or area of the system as a consequence of activity executed in another area, are not being carried through.

The extra effort required for this cross-correlation relates *at least* to the level of stratification of total activity. We have recognised $2^{2(i-1)}$ activity-elements in the i^{th} interval. The additional work load will be proportional to the number of interconnections between the various records,

that is approximately $2^{4(i-1)}$ (5.1). Thus the total effort W_i in the i^{th} release interval contains an additional term which we shall refer to as the secondary or exponential term E_i and which may be represented by:

$$E_i = C2^{4(i-1)} - D(i) \quad (10)$$

C represents the average magnitude of the effort required for cross-correlation between two activities. Its value is a function of the average content of an activity, the structure and content of the system and the extent to which tools to automate the correlation activity are available. $D(i)$ represents the fact that individual sequences can in fact decay to zero. More specifically it expresses the extent to which the total activity (particularly the documentation) of the i^{th} release is accurate and complete in terms of the *total state or content of the system* at that time, and not merely a reflection of the changes and additions completed during the preceding release interval.

The preceding derivation has assumed that the activity required in one stratum during one release interval is homogeneous in terms of the function and region of the system to which it relates. In practice activities will consist of *subactivities*, relating to separate fault reports, functional specifications, system components and so on. The micro-model will show how this sub-stratification further increases the magnitude and rate of growth of the secondary exponential term. That is (10) represents a lower bound on the required correlation effort, more generally represented by:

$$E_i = C2^{4Gi-D(i)} \quad (11)$$

where G is related to the average number of substrata in each activity. D has been modified for simplicity. Combining expressions (8) with (11), we obtain the total effort in the i^{th} interval as:

$$W_i = f \sum_{k=1}^i A_k e^{-L_k(i-k)} S_k + C2^{4Gi-D(i)} \quad (12)$$

5.1 (Orig) The number of levels of activity will actually be $2^{2(i-1)}(2^{2(i-1)}-1)$ corresponding to the number of edges in a complete directed graph of 2^{2i-1} nodes.

as a measure of the work required to maintain and grow the system. Assuming that experience gained on one system change applies to all activity, and that we may ignore differences between individual changes and functional additions, (12) reduces to:

$$W_i = Af \sum_{k=1}^i e^{-L(i-k)} S_k + C_2^{4Gi-D(i)} \quad (13)$$

Applying the environmental (management and budget) constraints, (13) reduces to:

$$W_i = W(B(i)) + C_2^{4Gi-D(i)} \quad (14)$$

$D(i)$ may be further expanded to represent the completeness and correctness of documentation (D), its accessibility (A), and the consequences of increasing experience and familiarity with the system (L), yielding:

$$W_i = W(B(i)) + C_2^{4Gi-D(i)A(i)L(i)} \quad (15)$$

$W(B(i))$ represents an appropriate function of the release budget allocation (plus supplementary appropriations) $B(i)$ that covers the cost of all activity except that included in the second (exponential) term.

The term $W(B(i))$ represents the cost of *primary* activity, that is activity undertaken to improve the cost, performance and other ratios of the system. The secondary term E_i on the other hand represents the cost of communication, correlation and *secondary* repairs. That is it accounts for activity that must be undertaken as a consequence of the primary change, because of subsystem interactions, the sharing of variables and so on, but that does *not directly* relate to the functional enhancement of the system.

We observe parenthetically that these two terms, and the behaviour of our model conform closely to a model suggested by Baumol in a recent paper [BAU67]. That is, activity represented by $W(B(i))$ is progressive (in Baumol's sense) in that its cost may be expected to be amortised in cost and performance improvements. The activity represented by E_i on the other hand is non-progressive and appears as a necessary evil. Much of the communication and correlation effort, for example, will lead to the conclusion that certain sections of the system need *not* be changed. On the other hand, if the work is not undertaken, faults will subsequently appear and

lead to all the effort associated with fault detection, analysis and repair. In the following sections we show that despite its tendency to unmitigated growth, productivity improvements are possible for the non-progressive activity. This supports counter arguments by C S Bell and others [BEL67] in their comments on Baumol's hypothesis.

The present theory has been developed only for a system after its first implementation, that is for $i \geq 1$. The special case of $i = 0$ has not been considered through it is clear that a model could also be developed for that case, that is to a system during its initial development. For $i > 0$ the exponential term of [14] and [15] can nevertheless assume, algebraically, any value between infinity and zero.

A perfectly structured system, for example, in which any changes were completely localised would have G equal zero and hence a negative exponent. We assert, however, that G does not remain zero as the system grows. The addition of any function not visualised in the original design will inevitably degenerate structure. Repairs also, will tend to cause deviation from structural regularity since, except under conditions of strictest control, any repair or patch will be made in the simplest and quickest way. No search will be made for a fix that maintains structural integrity.

The parameters $D(i)$ $A(i)$ $L(i)$ really quantify these effects, the accuracy and accessibility of documentation, the consequence of increasing experience and familiarity with the system (ie documentation and its accuracy less vital) and the degree to which separate activities have been coalesced or merged. Thus we may visualise a system in its early life in which the structure is still relatively pure (G small) and documentation is relatively complete. As familiarity with the system increases $D(i)$ may be larger and increasing faster than $G \cdot i$ with the result that the exponent actually becomes more negative and the correlation (maintenance) effort appears to decrease with time.

However, as already observed, G will ultimately increase as does i . It is also common experience that documentation inevitably lags the maintenance effort, that is, it becomes incomplete and/or inconsistent. As the system grows and personnel change, familiarity with the system also decreases. Thus we assert *in practice*, $D(i)$ ultimately decreases relative to $G \cdot i$, the exponent cannot be held negative or small and the exponential term grows rapidly. This corresponds to the algebraic truth that the difference

between two large numbers cannot indefinitely be held absolutely small as the numbers grow. Equally it relates to the observed phenomena that system complexity grows with time while accuracy of documentation and familiarity with the *total* system decrease with time.

That is, the total work planned and undertaken to maintain and expand the system will be supplemented by an amount which is initially small relative to the overall level of activity. Thus it will not be separately identified and the system will give all appearances of settling down, being manageable and stable. However, at some time, the effects of complexity will begin to be felt, ultimately becoming dominant.

The exponential growth in effort which will be required to maintain and grow the system can express itself in a need to apply more and more resources to system maintenance and development. Alternatively, or in general simultaneously, it will be reflected in increasing *difficulty* in making changes, in keeping schedules, in getting items of work completed on time. That is, we may represent the effect as in Figure 7a on a constant time-interval base by a rapidly and exponentially increasing work requirement. This is held within budget limits only by abandoning or delaying the addition of planned function. More often the workload is maintained at an approximately constant level as in Figure 7b, but with releases spread at increasingly large and unpredictable intervals of time. In either case the exponential growth is devastating to plans for further system growth. Sooner or later it will dominate and ultimately destroy system stability.

Note that, in part at least, there exists a choice in the implementation of any change. One may (in theory *and given sufficient documentation*) make an exhaustive check on the entire system to uncover all the relationships and consequent changes arising from any local change that is made. If this is done successfully redundant effort will, as indicated above, be expended. However the effects of the changes have been localised in time. Alternatively, one may make a more cursory determination but must then expect to set up a primary and subsequent secondary fault sequences whose impact (on performance and schedules) will be felt during later releases. Whatever compromise is made between the extreme limiting strategies, the ultimate consequences of complexity, sooner or later, will be an explosive increase in required effort.

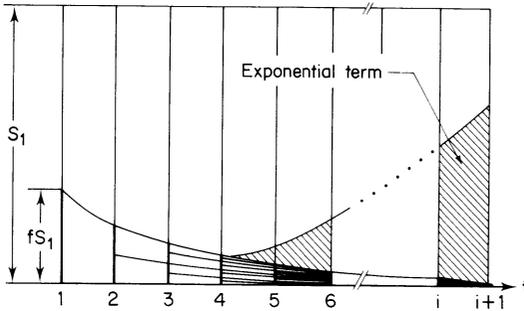


Figure 7 Increasing Complexity

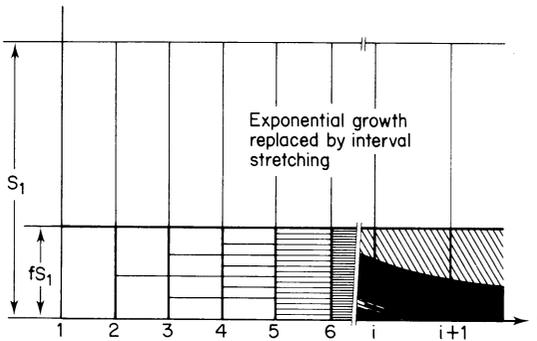


Figure 8 Lengthening Release Intervals Because of Increasing Complexity

7 Deductions from the Macro-model

From the structure of the macro-model (15) one may draw conclusions about certain aspects of programming methodology and system structure. To prevent the effort represented by the exponential term from growing too rapidly the interval between releases should, paradoxically, either be stretched to the maximum or reduced to a minimum. The former strategy holds down the rate of growth of i . By *completing* a given amount of repair and functional enhancement and releasing it to the user in the smallest numbers of releases, one obtains more complete correlation between documentation and code and decreases the probable number of errors released. One may conjecture that software development for the Apollo space program was in fact forced into this mode of operation with releases defined by the individual flights. There, a priori, existed *widely spaced* and *well defined* points in the development program at which the entire system and its documentation had to be stabilised and checked for completeness and consistency. Each flight-release establishes a new, well tested, *base* system from which one may move forward, *almost* ignoring all previous releases.

Alternatively, one may make the release interval so small that the quantum of activity in each is reduced to the level where it can be most completely defined, implemented, documented and distributed. That is $D(i)$ increases as rapidly as $4G_i$ and their *difference* remains small. To be successful this strategy must however, include a high degree of automation to keep documentation, code changes and functional changes in step (5.2). Intuitively one may suppose that it will work best for a system of which only one copy is extant, which is being written in a sufficiently high level language, and is in a well structured form. Thus the code itself can be directly read and therefore compose the primary system documentation; the authoritative source of system definition.

This strategy relies on the difference $4G_i - D(i)A(i)L(i)$ remaining small. Thus one would expect the smooth growth of the system to be disrupted every now and then by instability ($4G_i$ and $D(i)A(i)L(i)$ growth out of step). There is strong corroborative evidence, which must however be further investigated, that this is precisely what has happened in the

5.2 (Eds) *A concept that must have been considered far-fetched when written becomes plausible now that the role and importance of IPSES has been recognised.*

past in at least two major system projects that followed a strategy of almost daily releases.

Small G will also delay the onset of serious exponential growth problems. G is a measure of system size, structure, and complexity, in short, of system entropy. Its detailed analysis must await more complete development of the micro-model to be discussed in the next section. It is however abundantly clear that it may be kept small by system structuring, appropriate partitioning of function and code amongst sub-elements of the system, the definition and control of interfaces between these sub-elements, the complete testing of sub-elements before integration, and reduction of free system-internal communication via global objects. We have already recognised, however, that however well structured a system is in its initial implementations, the structure will inevitably degrade as new function is added. Thus G will tend to increase with time. We note that the problem of initially small G is, in part at least, strongly related to hardware partitioning problem that seeks to distribute logic circuits on chops and chips on cards so as to minimise pin and interconnection requirements.

Finally the coefficient C relates the total activity required to human and machine effort. The magnitude of the exponential term as it relates to total dollar costs, man-days and computer hours, can be reduced through the application of a total system development methodology based on an integrated family of tools for specification, design, documentation, testing, measurement, activity and system growth monitoring and project management [LEH69] (5.3). From the model *it follows directly* (5.4) that, for extended programming projects, the main emphasis for tooling and automation should be in the area of project communication and documentation.

8 The Basis for a Micro-model

In the preceding sections we have developed an intuitively appealing, but nevertheless heuristic, macro-model. Superficially at least, its functional characteristics appear to explain the long term behaviour of some large programming projects. Specifically, exponential growth trends in some

5.3 (Eds) *Another example of foresight. It has taken till now for this concept, now termed an IPSE, to have become generally recognised and accepted.*

5.4 (Eds) *Italics inserted 1984.*

measures of system maintenance effort have recently been observed for a number of systems. The observations and their interpretation will be separately documented.

It is however common experience that not all large programming projects are equally sensitive to these growing pains. To some degree at least this may be due to all aspects of the respective projects; to all of the factors or parameters in our macro-model. But it is generally agreed that specific attributes of the systems themselves play a strong role in the rate of exponential growth. Thus a micro-theory is required that explains the relationship between a system's content and structure and its Growth Dynamics. The present sections outlines such a microtheory, on a phenomenological basis. When expressed analytically the theory should provide underpinning for the macro-model; structure and magnitude for the parameter G of (14).

A programming system may be viewed as a space in which objects such as variables, control blocks, tables, queues and so on are defined. The objects act as communication links between the different elements of the system. Their number and individual structures are a measure of the 'mass' of the system, its resistance to change. This is invariant in a static system, that is one that is not *being changed*.

Any change in the system represents at least a change in the status or definition (meaning) of one existing object or the creation of a new one. A change may also impact the meaning, significance or structure of other objects in the system not directly involved the change. Whenever a change is made locally in some part of the system, it is necessary to locate *all* of the other places in the system in which the value or the form of a related object or its dependents are specified, re-specified or used. Then one may determine whether the change that has already been planned, demands some change in these other regions of the system. To compound the effort required, the procedure for this may have to be iterative. Thus work must be performed on the system to achieve change and its magnitude is related to system mass and to the degree of interconnectivity between the elements of the system. That is, the number and pattern of the communication links (and hence the probability of spreading change or faults) determine the complexity of the system. As they increase the entropy or disorder of the system also increases and changes are increasingly arduous to implement. These concepts reflect the accepted viewpoint that a well structured system, one in which communication is passed via parameters

through defined interfaces, is likely to be more growable and require less effort to maintain than one making extensive use of global or shared variables.

The total functional capability of a static system in a static environment, its 'free energy', is also invariant. It is a direct outcome of the functional relationship between the objects as defined by the system code. The total free energy, its power to do work in or on its environment, will grow if work is performed on the system to add or improve function or to repair faults. It will diminish through the appearance of faults from whatever cause. The work needed to maintain and grow the system is also clearly related to the total function or energy of the system since the number and average number of instances of objects will be related to the total amount of code in the system.

Finally we may consider the (average) probabilities P_{ijk} that the interconnection between regions i and k , as determined by a shared object j , is overlooked or misinterpreted, or that a required change is incorrectly implemented. The number of faults initiated is proportional to these probabilities, and to the number of system-internal interconnections, the internal communication of the system. For a system in which the average number of interconnections per object involved in a change *exceeds* the reciprocal of the probability of error P_{ijk} , any change in the system causes, on the average, $(1 + \epsilon)$ further faults. That is the system has reached a critical mass and may have become unstable. On purely theoretical grounds stability cannot be restored without a fundamental change in methodology. One might argue that because of the discrete nature of the phenomena, an environment could be created in which by halting all further system expansion and by the expenditure of a large amount of effort, the situation could be brought under control. However, we suggest that this is likely to prove an unsound strategy whether from a technical or business point of view.

9 Summary

It has been shown that programming systems change with time. As a consequence of their size and complexity, and that of the organisations that produce them, such systems are always limited in their ultimate growth potential. Ultimately the maintenance effort required *must* increase exponentially. At that stage the system becomes, at best, unprofitable to maintain or expand, and at worst, unstable.

The magnitude of the effort arises from the need for those that implement the changes to be aware of the total internal state of the system at all times, and for numerous implementors working in parallel to communicate with one another. It is also related to the number of shared or global objects in the system or to the extent and topology of communication between them. Thus the latter represents a measure of the disorder or entropy of the system; the degree to which the system and/or its documentation is unstructured. In short, the exponential increase in maintenance and development effort that appears in ageing systems (after they appear to have settled down to comfortable middle-aged maturity) is a consequence of the need for increasing communication, both that internal to the system and that between the people and groups (over spatial and temporal gaps) implementing, maintaining and using the system.

Further development of these models should yield more insight into the phenomena they represent. Thus it will be possible to provide a formal theoretical basis on which to develop project and system structural rationale. This must be done for the system's internal structure, for system communication, for project and programming process structure and for documentation and other tools which are required to implement and control the development and growth of the system. That is development of the models relates directly to the development of a *programming system development methodology*. This should and would structure a system project, so as to extend the viable lifetime of the resultant system and delay the possible onset of mass criticality.

There will, however, always be growth and an economic limit to permissible growth. We conjecture that the exponential term in our model must always exist and ultimately dominate. Thus the objective can only be to delay explosive cost growth and/or the critical mass situation beyond the other economic and technical life-time limits for the system. The present primary requirement must however be to quantify *complexity of growth* measures. Then one will be able to plan precisely for the development, maintenance, growth and ultimate termination of a system, and its replacement by a successor built up from ground level.

Finally it may be observed that the phenomena discussed here are largely independent of the specific function and structure of programming systems. Thus the models will also find more general interpretation in other large system areas. We have already indicated the close relationship between our

model and the macro-economic model of Baumol [BAU67]. We are also able to demonstrate such a relationship in at least one sociological situation relating to the increasing dominance of middle management in industrial organisations [LEH68]. Thus the phenomena discussed here appear to represent a fundamental growth attribute of systems and society.

10 Acknowledgments

We wish to acknowledge the many helpful discussions with our colleagues, too numerous to list individually. Special thanks are however due to Dr D N Streeter for his consistent support and encouragement during the birth and development of these ideas and also to him and to Dr H Freitag and J Pomerene for specific contributions.

11 Afterthought

On rereading this paper in 1984, the editors were struck by the continuing relevance of its models and observations. At the time of its publication the report did not achieve wide exposure nor did the authors receive much reaction, true, in fact, for most of the chapters of this book. The subject matter was simply not perceived to be of real consequence at that time when the principle objective was to get more and more programs written more and more quickly.

Nor did the authors pursue any further the process models they had developed or their implications. On now reviewing the material it does seem that it is worthy of further investigation and development, now that more and more people are recognising the central importance of understanding and controlling the Programming Process [SWP84].

CHAPTER 6

AN INTRODUCTION TO GROWTH DYNAMICS*

1 Project Statistic

The lifetime of any large programming system normally extends over a series of releases. Each such release represents an improvement on its predecessors as regards faults that have been repaired, performance weaknesses that have been removed, functional capability that has been improved and new function that has been added.

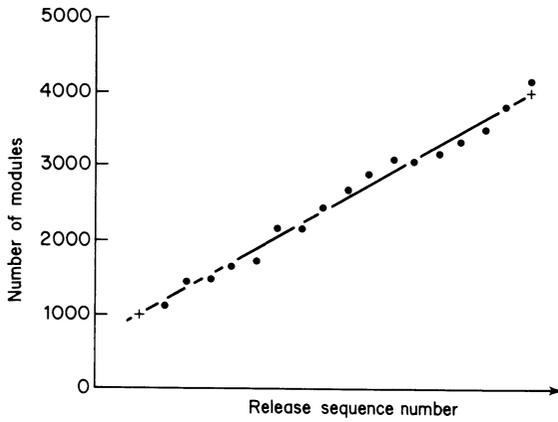
The history of a programming project is, in part, contained in the statistics (if recorded) of these releases. Thus typically there could be data available on the number of instructions and/or components (modules) added, removed and modified during the release, the number and reporting rate of previous repair work, the man-days, machine hours and dollar cost of each release, the time interval between releases and so on.

Examples of such data for a major system are given in figures 1 - 3. Figure 1 is a plot of the number of instructions in the system as a function of release number. It displays clearly the linear growth of the system. In the present paper we shall not analyse the deviations from linearity, except to observe that the ripples on the data are typical of a self-stabilising process with both positive and negative feedback loops (6.1). That is, from a long-range point of view the rate of system growth is self-regulatory, despite the fact that many different causes control the selection of work implemented in each release, with varying budgets, increasing number of users desiring new functions or reporting faults, varying management attitudes towards system enhancement, changing release intervals and improving methodology.

Figure 2 shows, as a further example, a plot of the net instruction growth of the system per release-interval day. The points are fairly widely dispersed as would be expected for any project stretching over several years. Surprisingly,

6.1 (*Orig*) *This observation was suggested to us by G Prada.*

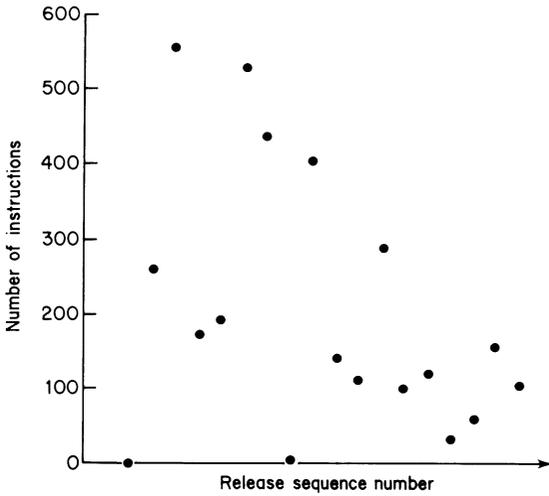
```
H <- CHN[B;4]
A F PLOT (H VS B)WITH(BS LIN H LSQ2 B)VS BS
```



DEC. 14 1971/101

Figure 1. Normalised component growth per release

```
H <- (CHN[B;7]-CHN[B-1;7])+CHN[B;10]  
A FPLOT H VS B
```



DEC. 14 1971/104

Figure 2. Normalised net instruction growth per day.

however, if these same data are smoothed, for example by averaging as in figure 3, the resultant points lie on a very smooth curve. The points of figure 3 can in fact be least-square fitted to a straight line and the logarithm of the points to the exponent of an exponential, with a chance probability of less than 0.1% and 0.01% respectively.

An extensive study of the total project statistics for this system has consistently displayed the same phenomenon. That is, in most cases, the raw data showed large variations from release to release. Smoothing or averaging, however, resulted in points that clearly lie on very smooth curves. In general these curves were of three classes - linear, exponential and bell-shaped. Linear fits appeared for measures reflecting the growth in size of the system. Exponentials appeared to fit well to points related to measures of system complexity. Data which reflected the effort (men, machine-power, cost, etc) going into successive releases appeared to lie on a bell-shaped curve that represented an effort at first decreasing and subsequently increasing continually. This last observation corresponds to the common experience of many projects that a settling-in period during which maintenance becomes progressively easier is followed by increasing maintenance problems that ultimately cause the replacement of the system.

2 Growth Dynamics

The behaviour of smoothed data, as discussed in the previous section, suggests that there is a self-regulating dynamics that underlies the growth of the system. That is, we may consider a system that comprises the body of code (constituting the programming system) together with the programmers that implement change and enhancement, their managers and executives and the user population. The dynamics of this system (the code, the programmer population, the applications, the hardware) as its internal state changes with time is governed by laws that cause it to be self-regulatory. In Simon's terms [SIM69] we may view this total system as an artificial system and search for the laws that govern its behaviour.

3 The Macro-model

In two recent papers [LEH69], [BEL71], the present authors presented studies of the programming process. They showed

that the effort required to take a system from the i^{th} to the $(i+1)^{\text{st}}$ release could be represented by

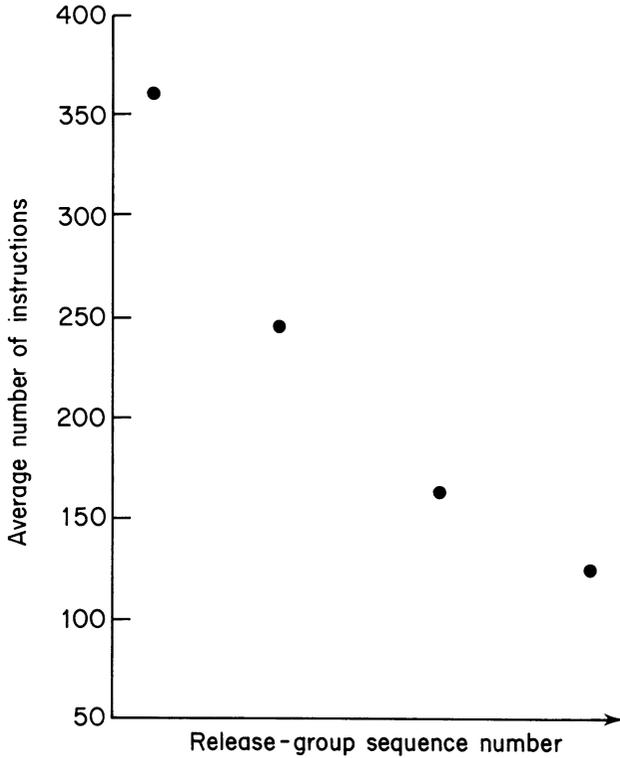
$$W_i = W(B(i)) + C_2 2^{G(i)} i^{-D(i)} A(i) L(i) \quad (1)$$

In this expression $W(B(i))$ is a function of the budget allocated to the i^{th} release. $G(i)$ measures the lack of structure in the system. That is, a programming system perfectly structured would have $G(i)$ equal to zero while for a system totally unstructured (every element connected to every other element) $G(i)$ would approach infinity. The two parameters $D(i)$ and $A(i)$ measure the degree of knowledge that exists and is available about the internal state of the programming system. That is, $D(i)$ represents the completeness and accuracy of both present and archival system documentation, while $A(i)$ measures the accessibility of that documentation. $L(i)$, on the other hand, represents the experience and knowledge about the system, of the programmers working on the system in the i^{th} release. The greater the awareness, the less important is the existence and/or accessibility of the documentation.

We have not at this time exhaustively validated this macro-model, nor are we able to assign numerical values to the parameters. Qualitative and corroborative evidence of the general form of the model is, however, seen in the bell-shaped curves previously mentioned. Figure 4 represents an example from the same system referred to above, measuring components handled per day (smoothed data). The increased productivity indicated during the early days of the release resulted from the dominance of the first term of (1). Thus, during this period, budget and manpower allocated to the project was increasing, as were familiarity and experience with the system. At a later point in time, however, excessive rates of raw growth were attempted, documentation (as is quite usual) slipped behind, faults from earlier releases were reported as the number of users increased, new programmers were introduced and the second term began to dominate, leading to a falling-off of productivity.

Interpretation of the model also fits many of the concepts for improved programming effectiveness and quality that have evolved heuristically over the past few years. Thus, we see the macro model as a conceptual framework that explains many previously-known observations. $G(i)$, for example, corresponds to Dijkstra's concept of structured programming [DIJ70]. Similarly, the nature of the exponent as the difference of two numbers shows why the use of high-level,

```
H <- (CHN[B;7]-CHN[B-1;7])+CHN[B;10]  
A FPLOT(C42 RDC H)VS C42 RDC B
```



DEC. 14 1971/102

Figure 3. Normalised net instruction growth per day,
N-point average

self-documenting, error-avoiding programming languages makes a system more long-lived. It tends to keep the difference between the two terms of the exponent small, however large the terms may be individually.

As another example, we note that the exponent, as a difference between two numbers, would tend to grow with time even if it was initially small or even negative. However, at any time it can be made small again by spending sufficient effort (aside from the main enhancement/repair effort represented by $W(B(i))$) to update the documentation, to make it more accessible (increase $D(i)$, $A(i)$) and/or improve the structure of the system (reduce $G(i)$).

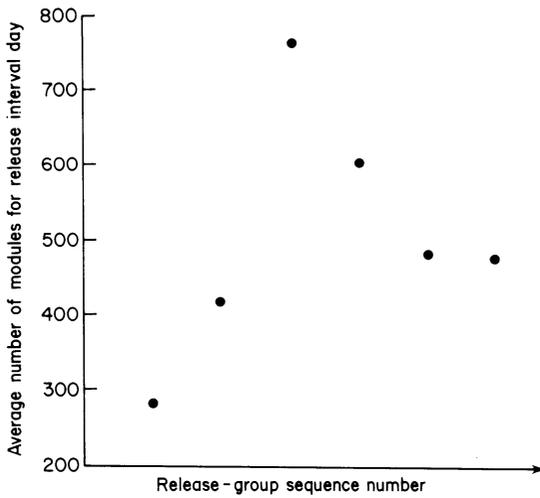
This last observation suggests that we may postulate a 'second law' analogous to the second law of thermodynamics. This would state that the complexity or unstructuredness - the entropy - of a programming system increases with time unless more effort specifically directed to its reduction is executed. That is, in our model $G(i)$ grows with time.

4 The Micro-Model

We have also proposed a phenomenological micro-model of programming systems [BEL71]. This relates to the propensity of the system to spread change and faults. $G(i)$ is in fact an abstraction of the micro-model which attempts to represent the lack of structure, the entropy, of a programming system by measuring the topology of interconnection between its elements. These interconnections relate to the sharing of objects (variables, parameters, tables, etc) by the elements.

We do not here elaborate on this model. It will, however, be observed that it also conforms to heuristic observations. For example, it confirms that parameter-sharing through defined programming-element interfaces is to be preferred to sharing of global variables, that goto statements (which represent spurious linkages between elements) should be avoided [DIJ68]. The micro-model is also consistent with the postulated 'second law' in that *almost* any change to a system will increase the number of elemental interconnections, by increased sharing of objects or adding branch instructions to inserted code sequences.

```
H <- +/CHN[b;5 10]  
R PLOT C30
```



DEC.14 1971/106

Figure 4. Normalised components handled per release-interval day, N-point average

5 Cities and other Systems

In a recent paper [BAU67] Baumol discussed the implications of unbalanced productivity growth on the macroeconomics of cities. The present authors suggest that there is a strong link between Baumol's model explaining city decay and the decay of programs as suggested by our macro-model (1). They have further suggested that a similar phenomenon can explain the increase of bureaucracy in government, the rise of middle management in large corporations and hence the decay trends in such institutions [LEH68].

These topics remain to be more fully studied. The authors, however, do not consider it premature to suggest that a common mechanism underlies all the above phenomena. They suggest that the observations reflect a decay trend present in all large systems. Thus, if the systems are left to follow their own dynamics they will tend to decay. Understanding the process can, however, lead to control of this decay process. Thus the authors recommend the study of this new and intrinsically statistical topic of growth dynamics to the participants in this conference.

CHAPTER 7

PROGRAMS, CITIES, STUDENTS - LIMITS TO GROWTH*

1 The Battlefield of Programming

In an invited lecture to the 1971 IFIP Congress, [RAN71] Professor Brian Randell of the University of Newcastle typified the situation current in the design, implementation and maintenance of computer software systems by showing a slide (Plate I) (7.1) of a medieval battlefield with ranks aligned but carnage abounding and devastation everywhere.

The situation today is really very little different. The programming systems world is a little older, perhaps a little wiser, but has in its practice not learnt much from the experiences of the last ten years. Many still ride bravely into battle (Plate IIa) determined to master the system that they wish to create or maintain. Most in one way or another fall by the wayside (Plate IIb). And programming expenditure goes up and up and up.

The total picture is however not completely bleak. Some dragons have been slain (Plate III). The need for a discipline of software engineering is recognised [NAU69]. The formulation of concepts of programming methodology exemplified by Dijkstra's concept of structured programming [DIJ72] strikes at the roots of the problem. The realisation that a program, much as a mathematical theorem, should and can be provable and proven correct as it is developed and maintained [DIJ68] and before its results are used will ultimately change the nature of the programming task and the face of the programming world.

7.1 (*Orig*) *My thanks are due to the management of the Alte Pinakothek Museum in Munich for permission and facilities to produce the first three slides, Mr P Young of the Victoria and Albert Museum for the loan of slide four and to the Directors of both museums for permission to reproduce the slides in the published versions of the lecture. Also to Professor G Seegmuller for his assistance in the preparation of the slides.*

First published 1974 as Inaugural Lect Ser., Vol 9 Imperial College of Science and Technology, London, also in 'Programming Methology' D. Gries (ed), Springer Verlag, New York, 1978 with permission from IFIP.



Plate I: Randell's depiction of the software crisis. (Detail from *Die Alexanderschlacht*, Albercht Altdorfer, by courtesy of the Alte Pinakothek Museum, Munich).



Plate IIa: Ride into Battle. (Detail from *Die Alexander-schlacht*, Albrecht Altdorfer, by courtesy of the Alte Pinakothek Museum, Munich).



Plate IIb: Fallen by the Wayside. (Detail from *Belagarung Von Alexia*, Feselen, by courtesy of the Alte Pinakothek Museum, Munich).



Plate III: Some Dragons Slain. (Panel from *Altar-piece with scenes from the life of St George* attributed to Marzal de Sas, by courtesy of the Victoria and Albert Museum, London).

Clearly, these developments are of fundamental importance. They offer the only long-term solution to the creation of the masses of programming material that the world appears to require; or at least that computer manufacturers and software houses think the world requires. Nevertheless, we must face the fact that progress in mastering the science of program creation, maintenance and expansion will be painful and slow.

2 The Systems Approach

Such progress as is currently being made stems primarily from the personal involvement of the researchers and the developers in the programming process at a detailed level. Often they only tackle one problem area: algorithm development, language, structure, correctness proving, code generation, documentation, testing. Others view the process as a whole but are still primarily concerned with the individual steps that, together, take one from concept to computation. And this type of study is clearly essential if real insight is to be gained and progress made.

But application of the scientific method has achieved progress in revealing the nature of the physical world by also pursuing a course other than studying individual phenomena in exquisite detail. A system, a process, a phenomenon, may be viewed in the first place from the outside, observing, clarifying, measuring and modelling identifiable attributes, patterns and trends. From such activities one obtains increasing knowledge and understanding based on the behaviour of both the system and its sub-systems, the process and its sub-processes. Following through developing insight in structured fashion, this outside-in approach in due course leads to an understanding of, and an ability to control, the individual phenomena but in the context of their total environments.

In terms of the previous analogy one may overfly (Plate IV) a battle, study it using all available observational tools. Thereby one would observe its ebb and flow identifying the location, global characteristics and, on closer and closer inspection, nature of the main points of advance, or of chaos and destruction. Having succeeded in this, one may hope better to understand, and hence modify and control what is going on.

In my present area of interest and concern I have adopted such a structured analysis approach to study the programming process. I shall be showing you samples of data that support

this systems approach to the study of the process. One need not search too hard for its conceptual justification. A programming project can involve many hundreds of people and many tens of managers. Many millions of pounds or dollars may be spent on it. Thus individual decisions of almost any kind make very little impact on the overall trend. It is really the inertia of people and habits, the momentum of practices and budgets, the general smoothing effect of organisations, that determine the rate of progress and the fate of a project.

3 The Roots of the Study

Some years ago I undertook a study of the programming process in one particular environment (see Chapter 3). As part of the study I obtained project statistics of a programming system which had already survived a number of versions or releases. The data for each release of the system included measures of the size of the system, the number of modules added, deleted or changed, the release dates, information on manpower and machine time used and costs involved in each release. In general there were large, apparently stochastic, variations in the individual data items from release to release. but overall the data indicated a general upward trend in the size, complexity and cost of the system and the maintenance process (Figure 1).

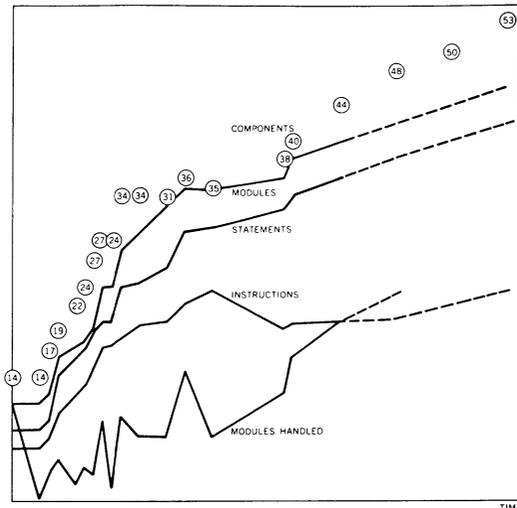


Figure 1 Early Data

As a first step in my study of this data I averaged the various parameters. The intent was of course to expose any specific trends. And expose them it did. When the averaged data was plotted the previously erratic data had become strikingly smooth (Figure 2).

Some time later additional data (Figure 3) confirmed previous suspicions of non-linear, possibly exponential, complexity growth. Moreover, extrapolation of the plots suggested further growth trends significantly at odds with the then current project plans. The data was also highly erratic (Figure 4) with major but apparently serially correlated (Figure 5) fluctuations from release to release. Nevertheless almost any form of averaging led to the display of very clear trends (Figure 6). Thus it was natural to apply uni- and multi-variate regression and auto-correlation techniques to fit appropriate regression and time-series models to represent the process for purposes of planning, forecasting and improving it in part or as a whole. In general the data suggested that one might consider a software maintenance and enhancement project as a self-regulating organism subject to apparently random shocks, but overall obeying its own specific conservation laws and internal dynamics.

All in all, these first observations encouraged me to search for laws that governed the behaviour, the dynamics, of the meta-system of organisation, people and program material involved in the creation and maintenance process, in the evolution of programming systems.

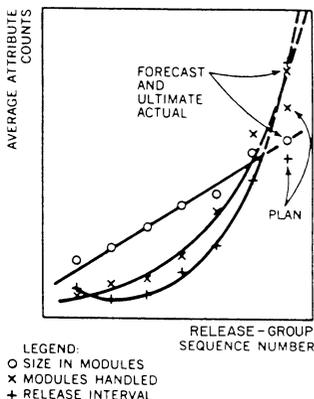


Figure 2 Initial Data
Averaged

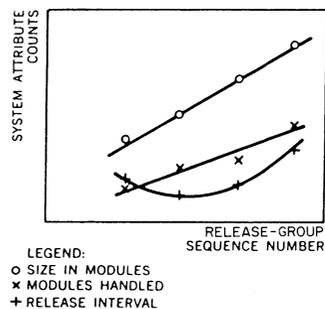


Figure 3 Later Data

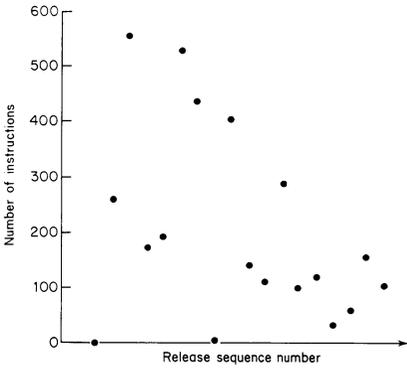


Figure 4 Scattered Data

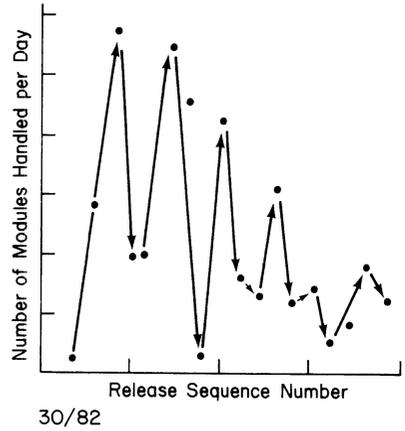


Figure 5 Serial Correlation

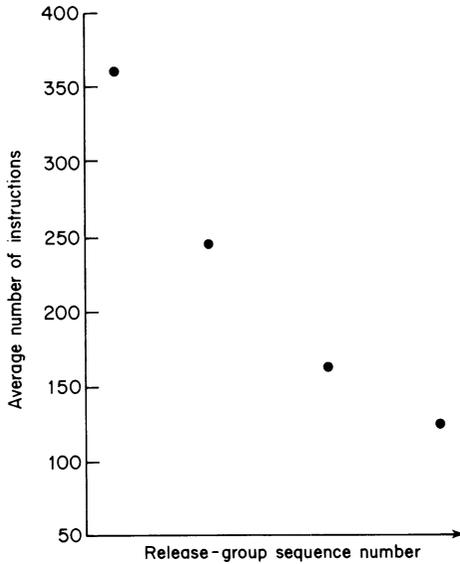


Figure 6 Overall Smoothness

4 Laws of Programming-system Life Dynamics

It is perhaps necessary to explain here why I repeatedly refer to the continuous creation, maintenance and enhancement of programming systems. It is the actual experience of all who have been involved in the utilisation of computing equipment and the running of large multiple-function programs the the latter demand continuous repair and improvement. Thus I postulate as a *First Law* (Figure 7) of Program Evolution Dynamics:

THE LAW OF CONTINUOUS CHANGE

Software does not face the physical decay problems that hardware faces. But the power and logical flexibility of computing systems, the extending technology of computer application, ever-evolving hardware and the pressures for the exploitation of new application areas all make users demand, and manufacturers encourage, continuous adaptation of programs to keep in step with increasing skill, insight, ambition and opportunity. Thus a programming system undergoes never-ending maintenance and development, driven by the potential difference between current capability and the demands of the environment.

As a system is changed it inevitably becomes more complex and unmanageable. Hence I also postulate a *Second Law*:

THE LAW OF INCREASING ENTROPY

This law too is supported by universal experience and in conjunction with evidence deduced from such programming project data as has been available and studied, has led me to the formulation of a *Third Law*:

THE LAW OF STATISTICALLY SMOOTH GROWTH

The system and its metasystem, the project organisation developing it, constitute an organism constrained by conservation laws. These may be locally overcome, but they direct, constrain and control the long-term growth and development patterns and rates. It is the study of one aspect of this third law, and its generalisation, that forms the underlying theme of the remainder of my talk.

I Law of continuing change

A system that is used undergoes continuing change until it becomes more economical to replace it by a new or restructured system.

II Law of increasing entropy

The entropy of a system increases with time unless specific work is executed to maintain or reduce it.

III Law of statistically smooth growth

Growth trend measures of global system attributes may appear stochastic locally in time and space but are self-regulating and statistically smooth.

Figure 7 Laws of Program Evolution Dynamics

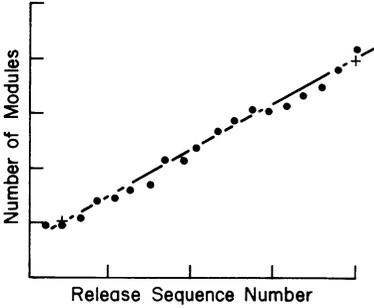
5 Statistical Models of System Growth

These laws, as formulated now, have gradually evolved as we have pursued our study of the programming task. In the first instance my observations simply led to the conception of an area of study which, at the time, we termed Programming Systems Growth Dynamics [BEL71], but which we now prefer to refer to as 'Evolution Dynamics'. In close association with a colleague, L Belady (who I am happy to say is here this evening as an SRC-sponsored Senior Visiting Research Fellow) it has also led to the development of statistical and theoretical models, such as those that I shall discuss, and to an ever-increasing understanding of the nature and dynamics of the programming process.

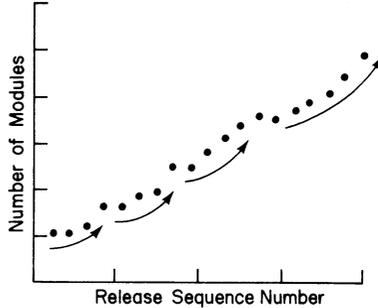
Let me give just one, very significant, illustration of the statistical models. Figure 8 plots the size of the system as measured in modules. Clearly the growth measured as a function of release numbers has been strictly linear. This growth is really very steady. However closer examination of the plot reveals a superimposed ripple (Figure 9). To control engineers this may suggest a self stabilising multi-loop, feedback system. Positive feedback arises, for example, from the desire of users to get more function, more quickly. This creates the pressures that cause the rate of systems growth to increase.

But a compensating negative feedback every now and again causes a decline (Figure 10). As attempts are made to speed up the growth process, design and implementation may become sloppy, short cuts are adopted, testing is abbreviated,

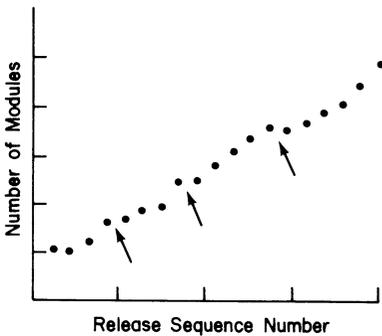
documentation falls behind, errors are made and the fault rate builds up. Changes to the code become progressively more difficult. Project and system entropies have become large, their structures have become hazy. The net result is that the growth rate declines. Both systems must be cleaned up, their structures improved, their entropy reduced, before enhancement of the program can be resumed.



17/1
Figure 8 Linear Growth



17/1
Figure 9 Growth Cycles



17/1

Figure 10 Clean-up Points

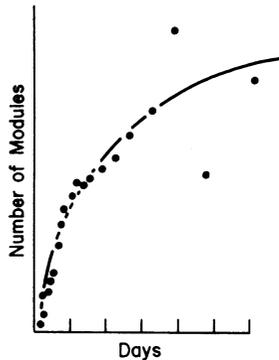


Figure 11 Limit to Growth?

Incidentally it may appear strange that apparently valid statistical models are obtained using an arbitrarily assigned 'Release Number' as one of the variables. Its use as a 'pseudo-time' variable can be justified by the 'Principle of Parsimony' [BOX70], the simplicity and regularity of the models that its use yields. A more fundamental justification follows from a fuller understanding of the role of the system release point as a stabilising or anchor point in the programming process and for the programming organisation. In fact what we have termed the 'release number' must really be viewed as the release 'sequence' number, which may well differ from the former and which, quite naturally, forms a basis for the time series type of analysis [COX66]. And there can be no question that results to date fully substantiate the claim that release-based project data provides a useful and powerful planning tool. Used in conjunction with real-time based models (Figures 11 and 12) it is also beginning to yield insight into the programming process, insight that will eventually permit its improvement.

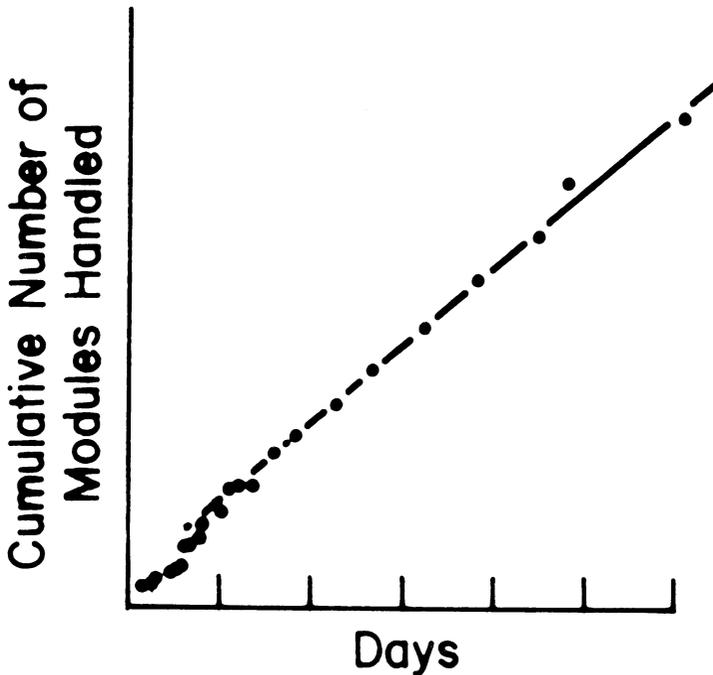


Figure 12 Constant rate of work-output

6 The Macro Models

The Yorktown Model

The statistical analysis represents, at most, a tool to guide our understanding and hence mastery of the intellectual and organisational effort that really underlies the programming process. In parallel with that analysis, Belady and I looked at fundamental aspects of the programming process, searching for representative theoretical models. It is not possible to elaborate on the details of the resultant sequence of models in the present lecture. I will nevertheless show you the forms of two of the earlier ones, without however interpreting all their terms, to give some idea of the approach taken.

At first approach [BEL67] we developed a model to represent the communication cost of such a project. We showed [BEL72] that in general the cost W_{i+1} of taking a programming system from release 1 to release $(i+1)$ can be represented as:

$$W_{i+1} = W(B_i) + wH_i^2 \cdot 2^{2G_i \cdot i} - DAL_i \quad (1)$$

Briefly the coefficient H represents the entropy, the degree of unstructuredness, of the program itself. G represents the entropy of the project. It measures the communication effort required at all times for coordination between the groups and individuals maintaining and updating the system. Finally the factor DAL represents the state of knowledge of systems and meta-system capability, structure and content, the correctness, completeness and accessibility of system and project documentation.

This macro model is static and does not express any controlling feedback relationships. It merely indicates that one source of exponential cost growth is the ever increasing difficulty of ensuring that changes to the code are compatible with its past definition and behaviour. Changes to the system today must not undo yesterday's work or prevent a repeat of yesterday's usage. Nor must they conflict with activities occurring elsewhere in the organisation, today.

Any tendency to diverge may be minimised and controlled by creating and maintaining a well-structured system (H small) and/or by having a well-structured, strongly communicating, project organisation (G small). Additionally, specification and documentation accuracy and program clarity must be

maintained, so as to ensure that all questions about the meaning, intention or effect of code can easily and unambiguously be answered and, ideally, so that the code can be proven correct. The exponential growth trend would be suppressed given perfect system structure, perfect project structure or perfect documentation but in practice these cannot be achieved or maintained.

The Berkeley Model

We now leave this primitive model and turn to one that displays the dynamics of the process. A dynamic macro model may be developed directly [BEL76b], applying the techniques of structured analysis, to yield as a simple example at the highest level and where all parameters are assumed to be functions of time:

$$W = u.F + M.v.K.F. + N.w.E. + P.x.R. \quad (2)$$

In any programming project there will be activity related to the design and creation of new code and the modification of existing code. This is represented in (2) by a factor of F. All such activity must be accompanied by additional activity, K.F, to document and record it, and both activity measures must be multiplied by expenditure rates or factors, u, v say, to yield the actual cost rate or cost terms.

There is also the activity represented by the exponential term in the previous model and now abbreviated to E, yielding a model term w.E. Finally, in any project there should be concern to improve the quality of the product and the productivity of its participants. The expenditure rate or expenditure of such activity is modelled by a term x.R. This therefore represents that part of the budget applied to methodology improvement and tool development.

We observe that the level of productive activity F and its division between, for example, repair (F_1), functional improvement (F_2) and the creation of new capabilities (F_3) is a management judgement based on fault rates, business considerations, and pressures from users. But once their average rate or level has been set, the necessary activities represented by the other terms to maintain the system health - that is, usable, maintainable and enhanceable - is predetermined.

In practice, however, under the pressures of continuous demand and imminent completion dates, the work whose neglect has no immediate consequence is pushed aside. Thus the

average level of activities represented by the last three terms will often fall *below* the level required to match program creation and modification. This reduction, a consequence of conscious or unconscious management decisions, is reflected in the model by the addition of management factors M, N, P that are, in general, less than 1.

The inter-parameter relationships

We have noted that all the variables and parameters in (2) are themselves time dependent. Additionally there will be time-dependent relationships between them. A complete model must express these relationships. Thus, as an example, if documentation is neglected ($M < 1$), the cost term u will increase in the future as knowledge of the state of the system declines and it becomes increasingly difficult to understand the content, intention or meaning of a piece of code. Moreover errors or defects in the system will increase, that is the ratio of repair activity (F_1) to enhancement (F_2 and F_3) will increase, an effect which is critical (and observable) in the dynamics of the process. Possible sets of relationships, the families of differential equations they lead to and the relationships between the solutions of these equations, available project and program data and our understanding of the programming process, are now being studied in our SRC supported research project under the title Systems Engineering (Growth Dynamics), and in a project 'Program Evolution Dynamics' led by L Belady at the IBM Yorktown Heights Research Laboratories.

Progressive activities

Much more could be said about this family of models. For now, however, I draw attention to just one of its characteristics. If we examine the four terms of (2) we find that the first is concerned with activity undertaken because of its assessable value to the organisation and the user. It produces code, hopefully usable code We have added as it were to the store of potential energy in the system, to its power. We have increased the value of the system. I term activity of this type *progressive*.

Anti-regressive activities

The other three terms on the other hand do not, by and large have a direct or immediate effect on the power or value of the system. System documentation, for example, must be undertaken while the work is being done or shortly thereafter. But it will be used only at some future time when it becomes necessary to modify the system. If it is there, well and good. If not trouble lies ahead. The system

will be difficult to repair, to change. It may not be possible to keep the system in harmony with a changing environment. *Relative to its environment*, it will have begun to decay.

The prime purpose of the expenditure represented by the E term is to prevent the insertion or perpetuation of a fault in the system that will, *at some future time*, result in undesirable or incorrect operation. Ideally it relates to the activity that would be required to re-prove correctness of the system each time it is changed. Since in the present state of the art this cannot, in general, be done, the term can equally be interpreted to represent testing activity which serves as a poor, but as yet essential, substitute. Thus all in all the term models the effort to minimise the number of undiscovered faults in the system. It too represents an investment in the future.

Finally the fourth term, modelling the cost of methodology and tool development, also represents a long-range investment. It is concerned with maintaining a capability to cope with system development and maintenance despite increasing size and complexity.

In general, these three elements of the model represent the cost of effort that I term *anti-regressive*. They are concerned not with immediately increasing the value of the system but with the investment of activity today to prevent system unmaintainability, and hence decay in the future.

7 Unbalanced Productivity Growth

Baumol's Principle and its Generation

It is my thesis that the complementarity of progressive and anti-regressive activity is fundamental to all human activity.

In 1967 Baumol [BAU67] discussed the consequences of unbalanced productivity growth. His principle conclusion was that human activities in areas where there is an intrinsic barrier to productivity increases must ultimately be priced out of commercial existence. But is this effect limited to instances where the obstacle to productivity increases is intrinsic? Any politician, administrator, manager, or even individual will be prepared to make an investment or an effort now, if the return is immediate or at least demonstrable. However to invest now so that in the future decay can be avoided or some other activity will be more

efficient is quite a different matter. Thus there is inherent psychological pressure for productivity to increase faster in progressive than in the anti-regressive areas. I suggest now that a Baumol-like effect will occur also in this case despite the fact that there is no intrinsic barrier to productivity growth.

What is the consequence of this? In general all progressive activity must be accompanied by some anti-regressive activity. As progressive activity and productivity grow so does the need and demand for anti-regressive activity. But so does the cost of labour in both areas. Therefore more and more anti-regressive activity, costing more and more, is required and until something is done about productivity in the anti-regressive area one of two things may happen. Either resources may be diverted from the progressive to anti regressive area. Progressive activity must then fall, growth rates decline and ultimately actual recession may set in.

Alternatively anti-regressive activity may be neglected. In that event, inevitability sooner or later further growth grinds to a halt. In a programming system, for example, this occurs when the fault rate becomes so high that the system must be cleaned up before further developments can proceed, an effect clearly visible in Figures 10 and 11.

The London Model

The preceding discussion suggests a further abstraction of our dynamic model. We may in fact model the dynamics of the programming process as a function of the interaction between progressive and anti-regressive activity, work output and system growth or evolution. By presenting a higher level view of the process, such a model may provide a more concise summary of its time-behaviour.

Also by abstracting in appropriate terms, the model may yield a wider interpretation, and therefore be used to describe a more universal phenomenon.

It is unfortunately not possible to present here a precise definition of the concepts of Progressive (P) and Anti-regressive (AR) activities, as would be required to develop such a model. Instead I shall simply outline, in terms of more primitive concepts, a model that results from some simple assumptions about the relationships between them and suggest one consequence that appears to relate to Growth and the Limits to Growth.

The primary assumption will be that productivity increases more rapidly in the progressive areas than it does in the anti-regressive.

With an additional assumption of linear productivity growth, the resultant models of manpower allocation and work output take the form of rational polynomial functions in the time variable t (Figure 13). We cannot examine these models fully here, but a sample system output shows clearly that, at best, under a set of rather unlikely conditions, the output grows to an asymptotic limit. In practice however output can ultimately be expected to decline. This is entirely due to the progressive and anti-regressive productivity imbalance and can therefore only be mastered by careful control of that balance. And this effect is very real, as can be seen from the behaviour of a derived rate-of-work measure taken from actual programming project data (Figure 14).

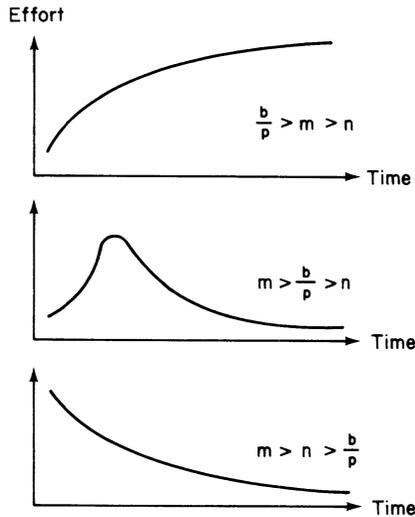


Figure 13

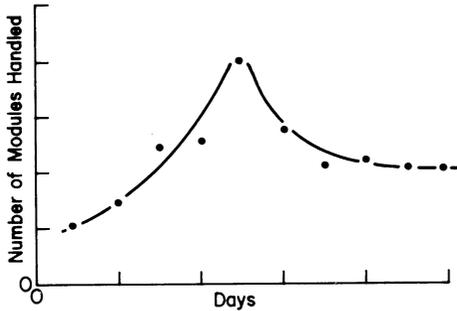


Figure 14 Average work-output rates

8 Generalization

Progressive and anti-regressive activities in city life

The high-level model that has been outlined was suggested by the earlier model of the programming process. I now suggest that it may also be interpreted within the context of sociological and economic systems. In a city, for example, we may identify as progressive (as previously defined) work output that contributes positively to the growth of the standard of living and the quality of life of the community.

But a community also undertakes anti-regressive activities. Collecting garbage, for example, does not increase the value of life in the city, it merely *maintains* the status quo. It is the *neglect* of that activity that leads to decay. Similarly police action is primarily directed to the prevention of deterioration of life in the city as a result of increasingly disordered traffic, breakdown in communication, criminal activity. It is strictly anti-regressive.

Unless the problem of productivity imbalance between these progressive and anti-regressive activities is mastered, the

previous model suggests that sooner or later the economics of a city will be dominated by the cost of the latter. And once again the underlying, psychological, cause for the resultant consequences is clearly understandable. Unless an electorate is very far-sighted, very sophisticated, the city fathers, the politicians who face re-election, the managers who seek promotion are concerned with short-range profitability, not long-range preservation. They will, in general, tend to favour progressive expenditure and investment over the anti-regressive. But any such consistent bias will ultimately but inevitably lead to slowed growth and then to decay. Indeed, as we have so recently seen, only immediate chaos and decay forces the adoption of adequate levels of anti-regressive measures.

Non-uniqueness of the P and AR assignment

To avoid confusion I should perhaps stress here that the distinction between progressive and anti-regressive activity is not always clear cut.

De-pollution activity, for example, is progressive and officials can support, even encourage, it without fear of losing their office in the next election or the next shareholders meeting. The control and prevention of pollution however *before* its effects are felt (or, I fear, when its effects are no longer felt) is a very different matter. This is something for next year, for the next generation, to worry about. This is something for the future. The same activity is progressive when it cleans up something that has already occurred and accumulated but is strictly anti-regressive when it is directed towards preventing the same thing from occurring.

Progressive-Anti-regressive conflict and balance

The preceding discussion can have given only the barest outline of the concepts and phenomena we are investigating. In fact we hypothesize that the life cycle of any complex, dynamic system is, at least in part, governed by the conflicting resource demands of progressive and anti-regressive activities. There will be an ever-increasing demand for the latter at a cost per unit of work that increases relative to the cost of the former.

There are three classical ways to deal with this problem. One may bring in resources from outside. In the United States, for example, an increasing number of State and City preservation projects are requesting and receiving Federal aid. But then the external source and the old system become

one, enlarged system, which too must ultimately limit, possibly decline. Alternatively one may divert resources from the progressive to the anti-regressive. This too implies a constraint on growth. A third alternative simply ignores the need for anti-regressive activity. But this is decay, even if it is not immediately visible. Thus individually or in combination these approaches all inevitably lead to a limitation to growth and hence to decay.

The way out exists. We must recognize that systems have content, structure and complexity. Consequent inertial and momentum effects result in limits on the rate of growth and of change, but not necessarily to growth itself. If one part of a system is caused to grow too fast another must grow more slowly or even decline. If at one time a system grows too fast, that must be followed by a standstill or even a decline. So the average growth rate is best restricted to its 'natural' value. Even more importantly the integrals of P and AR activity and their productivity growth must, on the average, be relatively balanced, however tempting it may be to leave the latter for future generations.

Anti-regressive activity in organizations

Before passing to the final theme of this evening's talk I would briefly mention two further points. First let me describe this same conflict as seen in the life of almost any organization. In a large business, for example, there are basically three levels of activity. The progressive rank and file engage in the activity that yields the revenue that enables the business to exist and grow. Executive management develops and initiates the policies and strategies that enable the business to prosper. This is also progressive. Finally one has middle management. The function of middle management is to act as a communication link, vertically and horizontally. They transmit, interpret, coordinate, protect and generally prevent the development of misunderstandings, harmful competition and internal conflicts between different sub-organisations and products.

Middle management is a strictly anti-regressive, communications, responsibility. As such it will tend to grow more rapidly than the progressive sectors of the organisation in both activity and unit cost. Moreover feedback analysis of the mechanisms of promotion, demotion and resignation from an organisation shows that the average level of competence in the middle management ranks of an organisation may be expected to decline with time [LEH68]. Unless carefully and consciously controlled, middle management will

tend to be a growing but increasingly mediocre organism. Thus a business, a government, even an educational institution, will get choked by its bureaucracy, the anti-regressive middle management, unless structure, function and productivity are carefully balanced. In particular it is vital that adequate resources for productivity development are correctly allocated over the organisation, so that at all times productivity in the middle management area balances that of the remainder of the organisation.

The Club of Rome

Let me also make very brief reference to the Club of Rome report on the Limits to Growth [MEA2]. In their summary (Figure 15) of the problems whose solution would help conquer the problems of growth limitation, all except one would be classified by me as anti-regressive. The fundamental oversight of the team was, perhaps, identification of resource exhaustion as the the primay cause of the immediate danger. In my judgement it is, in fact, primarily a symptom. The phenomena which they discuss appear to represent just another special case of my general rule.

The problem in this instance is that mankind has, by and large, not been willing to invest sufficiently in the anti regressive activity of long-range research and development, to find materials and techniques to replace curenly conventional sources of energy and raw materials when these are exhausted. The disasterous consequences of the exhaustion of specific resources is a direct consequence of the universal tendency to neglect apparently unprofitable anti-regressive activity, to prefer immediate profit to long-term security.

The sun, the atmosphere, the tides, for example, form an almost inexhaustible but as yet barely exploitable energy source. And given energy and mankind's ingenuity and creativeness, material shortages can always be overcome, provided that there is no practical limit to the growth of mankind's collective intellect. And that is the topic to which we now turn.

THE LIMITS TO GROWTH

The State of Global Equilibrium

Technological advance would be both necessary and welcome in the equilibrium state. A few obvious examples of the kinds of practical discoveries that would enhance the workings of a steady state society include:

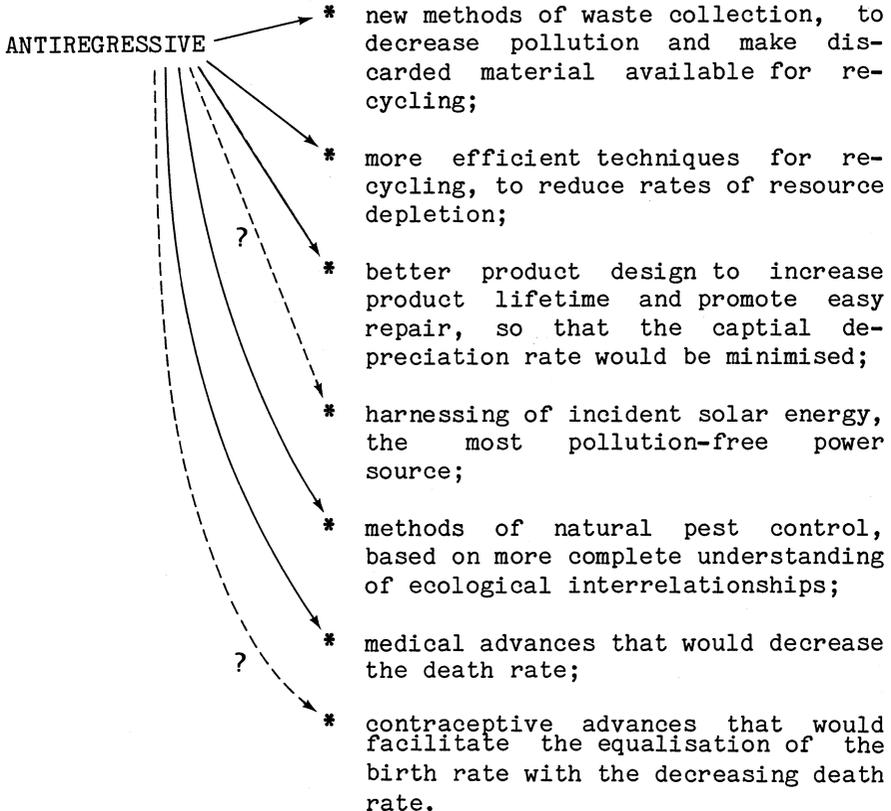


Figure 15 The true limiter of growth

9 Limits to growth through the educational process

Knowledge or understanding (7.2)

In education too there is the progressive and the anti-regressive. And, in my judgement, the same error, the analagous unbalance exists and unfortunately increasingly so.

But let me explain. The global objectives of an educational system are rarely stated or even recognised. One may hear talk of local or immediate objectives but the really fundamental goal is not discussed. In my wanderings through the world I have experienced various educational systems, as a student, as a parent, as a manager, as a teacher. Some have stressed knowledge, others understanding. The implications of this difference reach down into the very structure and heart of the society in which these systems operate. Educational systems may, in fact, be classified by the extent of their orientation to *knowledge* and to *understanding*.

In the limit, the objectives implied by these orientations are orthogonal, at times even contradictory, in terms of the methodology they imply and the results that they yield. Orthogonality does not, of course, imply exclusion. Most systems contain, and need to contain, elements oriented to and supporting both knowledge and understanding. But what we are seeking here is correct balance between them, for the individual and for society.

Knowledge

An educational system may primarily seek to transfer to its students some selected portion of the reservoir of human knowledge, as well as the means whereby this reservoir may be further accessed. Thus the knowlege which has been accumulated over millenia will be explicitly carried forward into the future.

7.2 (Eds) *These few paragraphs provide an indication of one reason why we are concerned about the current (1983) trend to the exploitation of Expert Systems. As their alternative name, 'Intelligent Knowledge Based Systems', indicates, such systems exploit the information (knowledge) in a data base and - at least with current artificial intelligence inference techniques - cannot demonstrate or exploit 'understanding' - the creation of new insight. Excessive reliance on such systems represents a real threat to society. See Chapter 22 for further comment.*

The drift to knowledge-oriented education occurs because its pay-off is immediate. The child comes out of kindergarten, is ready to go into school to absorb more knowledge there. It comes out of school and is able either to take a job and earn its living or to go forward to higher education and then earn its living. This is progressive. Each quantum of knowledge that is gained becomes the basis for further learning or for other activities as required by society. Moreover in this era of democracy and human equality a common standard of knowledge acts as the great leveller; it provides a definable standard against which egalitarianism can measure its success.

Understanding

At the other extreme, the pure understanding-oriented system seeks to instill insight; not so much the facts themselves as the relationships between the facts, valid analogies, cause and effect. An essential ingredient of this educational process is the development of analytic powers, self-expression and creative thinking, with the ability to seek out and elucidate new knowledge as required. Assessment of the degree of success achieved in the process is a matter of judgement, even viewpoint and opinion.

An understanding-oriented system is primarily anti-regressive. Its graduates cannot normally apply their insight and creative skills directly. Real life requires facts. But only a person with understanding can discard surplus or outdated knowledge. Only he or she can replace large repositories of knowledge by algorithms for their retrieval. It requires true understanding to develop new methodologies and to guide the knowledgeable to new horizons when the old ones become outdated or exhausted.

Attributes of the two systems

What are the attributes and hallmarks of these alternative educational systems? How does one determine in which direction a particular system is oriented? How can one structure a system and adopt appropriate methodologies to achieve a best balance between understanding and knowledge for the individual, for a community and for mankind at large?

The knowledge-oriented system is typified by excessive emphasis on self-learning and homework, grades and credits. It relies heavily on frequent examinations that test the extent of a student's knowledge on the completion of each quantum of study. High pass-marks and high scores result from a precise definition of what was to have been learned,

and from the statistical reliability of large numbers of multiple choice questions.

In the understanding-oriented system on the other hand emphasis must be quite different. Self-study and home learning start much later and the emphasis in marking will be on *how* a complex question was approached, not only on the correctness of the final solution. The award of a degree is not the sum of a large number of credits in different subjects each of which has been obtained immediately after a course has been taught. It follows from the completion of a course of study, for a demonstration of the nature and extent of the understanding that has been gained, of the meaning and the significance of what has been learnt. Multiple choice questions are of no use at all here. A student cannot express his understanding by marking a square black. His insight and ability is revealed by his choice of words, by the way in which he approaches a problem. We are interested to see how he has understood the facts and the concepts to which he has been exposed, what he can do with them, not so much in his ability to reproduce them.

Consequences of knowledge orientation

In a society in which education is knowledge-oriented the teacher adjusts his rate of teaching to the average absorptive capability of the class. Thus there will emerge a group of people clustered around a level of knowledge determined by their average absorptive capacity. The weaker ones will have been helped forward and this indeed is a good feature of the system. Those with more powerful intellects, a greater potential for creative thinking, will have been held back. They will not have been taught to express their own ideas but merely to repeat what they have heard from their teachers.

The consequences will be a society in which achievement must be obtained through teamwork. A knowledge-oriented society will be a technology-based society in which known facts are produced from the joint encyclopaedic memories of a team and applied to achieve specified and known to be achievable objectives, great works of technology. In themselves these works do not foster creative inventiveness or the development of new concepts or deeper insights.

Such a society must possess the capability to get people to work together, the ability to manage teams of people and guide them towards a common objective. It is knowledge orientation that leads to the need for management science.

Thus it is not in the least surprising that the USA with its 'melting-pot' society and, as a consequence, its knowledge-based, materialistic, orientation is the undoubted world leader in Technology and in Management.

And those of understanding

When we now consider the understanding-oriented system, the first thing to be noticed is that one needs more gifted teachers. Each student will interpret the teacher somewhat differently. If the teacher in turn understands the educational process, he can take each student to the limits of his individual ability. Inevitably some people will be left behind because they do not have the motivation or the ability to understand. Higher education will be more restricted and in the output of the overall educational system one finds a wide spectrum of knowledge, skills and creative ability, rather than the cluster produced by the knowledge-oriented system.

At one end of the spectrum will be highly motivated, creative, thoughtful individuals able to express themselves, to develop and to exploit new concepts. They will not require to interact strongly and intimately with others to achieve their inspiration because they have learnt to achieve results on their own or in very small groups. The demand for strong, effective management, and the ability to provide it, will not be so well developed and ultimately that may prove to be to the detriment of that society. It will, for example, tend to be scientifically creative rather than technologically productive: able to produce concepts, fresh viewpoints, new inventions and methodologies, but less able to develop and control their mass exploitation.

Mankind's intellectual growth

I should not like these remarks to be in any way misunderstood. Mankind needs both systems. The challenge today is not which way to go but how to provide the correct balance between the progressive and anti-regressive in education, as elsewhere. Knowledge alone will not suffice to ensure survival of humanity. Nor can we hope to retain all the knowledge that humanity has gathered over the last five thousand years. What must be done is to ensure that forgotten knowledge can be reproduced when requires. New ideas must be produced faster than the older ones may be forgotten. Understanding-oriented education is an investment in the future. It ensures that there will always exist those creative minds that can think for themselves, that produce and explore new concepts, that provide a sense of direction,

creative inspiration and action, possibly even a little bit of sanity, in this fact-oriented, technology-based, profit-seeking world. But knowledge is also required. At any given moment in time only a knowledge-oriented technological society can solve the immediate problems that face a community, a nation and mankind as a whole.

We have analysed education in terms of the progressive and anti-regressive system concepts. If the same laws of growth apply, we can ensure continued growth only by achieving the correct balance between progressive and anti-regressive education. Thus I would like to think that we here at Imperial College will help counter-balance a worldwide trend to knowledge-oriented education, by maintaining and emphasising the traditional understanding-orientation of British education. The new Computer Science course that we have set up has been carefully designed and structured to yield a student who *understands* and not just knows his subject. Thus I trust that we shall be playing our part in ensuring the continued intellectual growth of mankind and through that also its physical progress - progress in which computer systems and computer science will play a major role.

10 Conclusion

Finally may I be permitted to follow ancient Jewish practice and conclude my talk with a biblical viewpoint of the concepts I have presented. The author of Proverbs also recognised the potential conflict between progressive temptation and anti-regressive needs [Proverbs 6-8] (Plate IVb):

'Go to the ant, thou sluggard, consider her ways and be wise. Though having no officer, overseer or rule, she prepareth her bread in the summer, gathereth her food in the harvest.'

One might ask why the author here places bread preparation in summer before food gathering in the harvest when in fact in the Middle East the harvest is gathered in spring. The Malbim, a 19th Century commentator, remarks that the meaning of this text is that the ant eats in the winter *because* she has previously prepared her food in the summer. This she can do because during the spring harvest she is willing to store her collection rather than to eat it all. She works for a future that she may never live to see and this voluntary action occurs despite the absence of pressure from rulers or overseers. The anti-regressive is so integrated into the

progressive that the two remain balanced and in step at all times.

There is clearly no need to add anything further to these ancient words of wisdom except a final overall summary:

Programs and Cities and Students
All have the potential to grow.
With P and AR action balanced
There need be no Limit to Growth.

11 Acknowledgements

I would like to acknowledge the loyal support, the patience and the constructive criticism that I have received from my colleagues in the Computing Science section of the Department of Computing and Control over the last two years and during the preparation of this paper. In particular I single out the many thought-provoking discussions with A L Lim. Equally I must mention and gratefully acknowledge all I have learned and am continuing to learn from my colleagues in the IFIP working group WG2.3. Above all, however, I want to acknowledge the close and continuing association with L A Belady who, but for the nature of this paper, would surely have been a co-author. Last, but certainly not least, I want to thank my wife Chava for her artistic contributions to this lecture and for her constant and continuing support and encouragement which have made my work and this lecture possible.

CHAPTER 8

A MODEL OF LARGE PROGRAM DEVELOPMENT*

1 The Process of Programming (8.1)

1.1 Introduction

As a need for a discipline of software engineering has become recognised, the design, implementation, and maintenance of computer software has come into the forefront. The formulation of concepts of programming methodology, exemplified by Dijkstra's structured programming [DIJ72] strikes at the roots of the problem. The realisation is that a program, much as a mathematical theorem, should and can be provable. Recognition that a program can be proved correct as it is developed and maintained [DIJ68b], and before its results are used, may ultimately change the nature of the programming world. Clearly, these developments are of fundamental importance. They appear to point to long-term solutions to problems that will be encountered in creating the great amount of program text that the world appears to require. But even though progress in mastering the science of program creation, maintenance and expansion has also been made, there is still a long way to go.

1.2 The System Approach

Such progress as is currently being made stems primarily from the personal involvement of researchers and developers in the programming process at a detailed level. Often they tackle a single problem area: algorithm development, language, structure, correctness proving, code generation, documentation, or testing. Others view the process as a whole, yet they are primarily concerned with the individual steps that, together, take one from concept to computation. Still this type of study is essential if real insight is to be gained and progress made.

The scientific method has made progress in revealing the nature of the physical world by pursuing courses other than

8.1 (*Eds*) *The original paper did not contain numbered sections and subsections. We have added numbers and, where necessary, titles to correspond to the standard format in this book.*

studying individual phenomena in exquisite detail. Similarly, a system, a process or a phenomenon may be viewed from the outside, by acts of observing; clarifying; and by measuring and modelling identifiable attributes, patterns and trends. From such activities one obtains increasing knowledge and understanding, based on the behaviour of both the system and its subsystems, the process and its subprocesses.

Starting with the initial release of OS/360 as a base, we have studied the interaction between management and the evolution of OS/360 by using certain independent variables of the improvement and enhancement (ie maintenance) process. We cannot say at this time that we have used all the key independent variables. There is undoubtedly much more to be learned about the variables and the data that characterise the programming process. Our method of study has been that of regression - outside in - which we have termed 'structured analysis'. Starting with the available data, we have attempted to deduce the nature of consecutive releases of OS/360. We give examples of the data that support this systematic study of the programming process. Again, however, we wish to emphasise that this study is but the beginning of a new approach to analysing man-made systems.

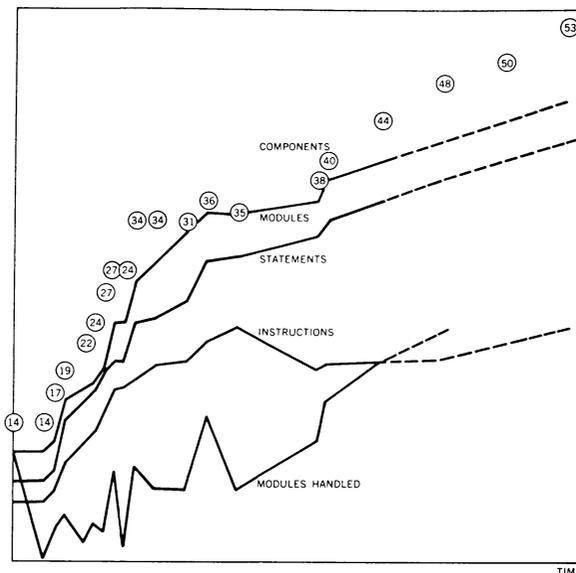


Figure 1 Growth Trends of System Attribute Counts with Time

1.3 The Programming Process

The authors have studied the programming process [LEH69] as it pertains to the development of OS/360, and now give a preliminary analysis of some project statistics of this programming system, which had already survived a number of versions or releases when the study began. The data for each release included measures of the size of the system, the number of modules added, deleted or changed, the release date, information on manpower and machine time used and costs involved in each release. In general there were large, apparently stochastic, variations in the individual data items from release to release.

All in all, the data indicated a general upward trend in size, complexity and cost of the system and the maintenance process, as indicated by components, modules, statements, instructions, and modules handled in Figure 1. The various parameters were averaged to expose trends. When the averaged data were plotted as shown in Figure 2, the previously erratic data had become strikingly smooth.

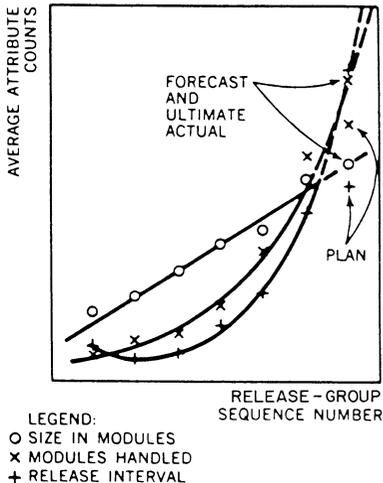


Figure 2 Average Growth Trends of System Attributes

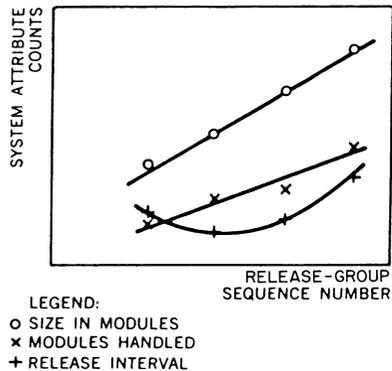


Figure 3 Average Growth Trends of System Attributes Compared with Planned Growth

Some time later, additional data were plotted as shown in Figure 3 and confirmed suspicions of nonlinear - possibly exponential - growth and complexity. Extrapolation suggested further growth trends that were significantly at odds with the then current project plans. The data were also highly erratic with major, but apparently serially correlated, fluctuations shown in Figure 4 by the broken lines from release to release. Nevertheless, almost any form of averaging led to the display of very clear trends as shown by the dashed line in Figure 4. Thus it was natural to apply uni- and multivariate regression and time-series models to represent the process for purposes of planning, forecasting and improving it in part or as a whole. As the study progressed, evidence accumulated that one might consider a software maintenance and enhancement project as a self-regulating organism, subject to apparently random shocks, but - overall - obeying its own specific conservation laws and internal dynamics.

Thus these first observations encouraged the search for models that represented laws that governed the dynamic behaviour of the metasystem of organisation, people, and program material involved in the creation and maintenance process, in the evolution of programming systems.

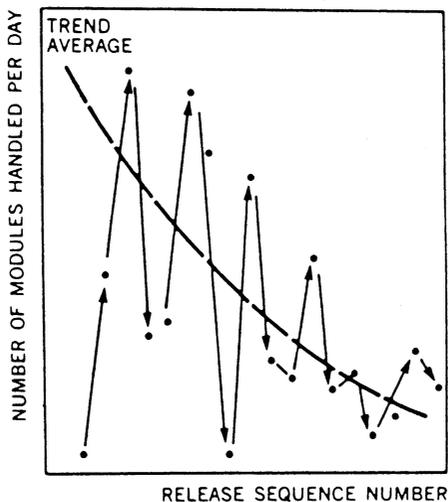


Figure 4 Serial and Average Growth Trends of a Particular Attribute

1.4 Laws of Program Evolution

It is perhaps necessary to explain here why we allege continuous creation, maintenance, and enhancement of programming systems. It is the actual experience of all who have been involved in the utilisation of computing equipment and the running of large multiple-function programs, that such systems demand continuous repair and improvement. Thus we may postulate the First Law of Program Evolution Dynamics [LEH74].

I Law of Continuing Change

A system that is used undergoes continuing change until it is judged more cost effective to freeze and recreate it.

Software does not face the physical decay problems that hardware faces. But the power and logical flexibility of computing systems, the extending technology of computer applications, the ever-evolving hardware and the pressures for the exploitation of new business opportunities all make demands. Manufacturers, therefore, encourage the continuous adaptation of programs to keep in step with increasing skill, insight, ambition, and opportunity. In addition to such external pressures for change, there is the constant need to repair system faults, whether they are errors that stem from faulty implementation or defects that relate to weaknesses in design or behaviour. Thus a programming system undergoes continuing maintenance and development, driven by mutually stimulating changes in system capability and environmental usage. In fact, the evolution pattern of a large program is similar to that of any other complex system in that it stems from the closed-loop cyclic adaptation of environment to system changes and vice versa.

As a system is changed, its structure inevitably degenerates. The resulting system complexity and reduction of manageability are expressed by the Second Law of Program Evolution Dynamics.

II Law of increasing entropy

The entropy of a system (its un-structuredness) increases with time, unless specific work is executed to maintain or reduce it.

This law too abstracts vast experience, in part reflected by data to be presented later in this paper. This, in turn, leads to the formulation of the Third Law of Evolution Dynamics.

III Law of statistically smooth growth

Growth trend measures of global system attributes may appear to be stochastic locally in time and space, but, statistically, they are cyclically self-regulating, with well-defined long-range trends.

The system and the metasystem - the project organisation that is developing it - constitute an organism that is constrained by conservation laws. These laws may be locally violated, but they direct, constrain, control, and thereby regulate and smooth, the long-term growth and development patterns and rates. Observation, measurement, and interpretation of the latter can thus be used to plan, control, and more precisely forecast the product of an existing process and to improve the process so as to obtain desired or desirable characteristics.

The 'laws' that we are expounding upon have gradually evolved as we have pursued our study of the programming task. When we began our studies, observations led to the concept that we termed 'programming systems growth dynamics' [BEL71]. We have now renamed this subdiscipline 'programming evolution dynamics'.

The remainder of this paper describes some of the statistical and formal models of the programming process that we have been able to develop by pursuing the consequences of the laws of programming evolution dynamics. It is our conviction that the extension of these studies can lead to an increasing understanding of the nature and dynamics of the programming process. Hence, studies such as these may yield significant advances in the ability to engineer software, ie, to plan and control program creation and maintenance.

2 The Process Observed - A Statistical Model

2.1 The Basis of the Study

The basic assumptions of programming evolution dynamics spring from viewing the program being implemented, enhanced, and maintained and its metasystem - the organisation that generated and undertook the development of OS/360 - as interacting systems. The evolutionary process and life cycle of a program are at least partially governed by the structural and functional attributes of both the program and the human organisation. Their size, complexity, and numerous internal interactions suggest the use of statistical techniques for interpreting observed behaviour.

Detailed studies of available data in conjunction with the almost universal experience of the programming community indicate that a large programming project has many of the properties of a multiple loop, self-stabilizing feedback system. The overall trend has been summarised in the previously discussed three laws that underlie the dynamics of evolution of large programs. The present section presents some of the accumulated numerical evidence derived from experience with OS/360 - one model of one system from one environment.

2.2 Available Data

The project data presented here originate from OS/360, which is now some twelve years old. This system has been made available to an increasing number of users in a series of over twenty user-oriented releases. These releases have extended the capability of the operating system by correcting faults, improving performance, supporting new hardware, and by adding newly conceived functions.

These and other intermediate releases were assigned names or numbers as identifiers. Each release may, however, also be identified as a program that - with its documentation - forms an identifiable and stable text in an otherwise continuously changing environment. Assigning *Release Sequence Numbers* (RSNs) to versions receiving the same degree of exposure, yields a sequence of integers that forms a pseudo time measure in the sense of Cox and Lewis, [COX66], that may be used to describe the time-dependent behaviour of program evolution.

Of the releases considered, the first represents the culmination of the basic design and build (ie, system integration) process. The iterative process that yields the specification, architecture, design and the first implementation of a large program system differs significantly from subsequent maintenance and enhancement activity. In particular, there is at this stage no feedback of fault reporting or performance assessment by independent users. Hence data relating to that first release are not included in this analysis. The build process itself may, however, be studied by using data obtained periodically during the development activity.

Data from a second release were also unused because they were shown to represent a component development somewhat off the main stream. In the final analysis, the model and the plots

to be presented are based on twenty-one sets of observations. This relatively small number of data points implies that extreme care must be exercised in interpreting the results of the statistical analysis. Subsequent data from the OS/360 environment, augmented by data from other environments, have generally confirmed our observations and conclusions.

2.3 Observables of System Evolution

The release sequence number (RSN) is taken as the first of the system evolution parameters. The second is the age of the system D_R at release with $RSN = R$. Equivalently, D_R is the inter-release interval I_R ; in other words, the interval in days between releases with $RSN = R-1$ and R , respectively. A third available parameter M_R measures the size of the system in modules. We present the results of our analysis in terms of modules, though other size measures - such as numbers of components or instructions in the system - could also have been used. The suitability of the module stems from the fact that in OS/360 the concept of module - though imprecisely defined - represents at one and the same time a function and implementation entity and, for execution, a unit of system generation and storage allocation. A fourth parameter MH_R records the number of system modules that have received attention, ie, those that have been handled during the release interval and, more specifically, during the integration process. We have used this as an initial estimator of the amount of activity undertaken in each release. The measure is imprecise, but represents the best available information over the entire sequence. From MH_R and I_R , in turn, we determine an estimate of the handle rate HR_R for the activity that produced the release with $RSN = R$.

From the very first beginnings of this study of the programming process [BEL71b], it has been clear that the changing complexity of a system, as it is modified, plays a vital role in the aging process. Unfortunately, there is no clear or unique understanding of what complexity is and how it can be defined and measured. The choice of complexity definition cannot, in fact, be disassociated from the use to which it is to be put. But complexity of the system, of the organisation, and of each particular series of changes is fundamental to the maintenance and to the resultant aging process. Hence some measures of complexity must be established.

For the purpose of the present analysis, *complexity* C_R has been defined as the fraction of the released system modules that were handled during the course of the release with $RSN = R$. This definition is clearly inadequate. It does not separately measure the various independent complexity factors involved. It does not discriminate between system organisation and the nature of the work undertaken. Nor does it measure the amount of activity involved. But at least it is a measure for which real data exist. Moreover the data give interpretable results. Hence $C_R = MH_R/M_R$ will suffice until better measures become available (8.2).

2.4 The Present Model

We have just identified five observable and measurable parameters of the programming process. Our hypothesis implies that these parameters do not vary independently, at least when viewed over a relatively long period of time. In fact, we have been able to determine, for example, four bivariate relationships among them. The complexity parameter, however, is derived from two of the others. hence, on the basis of present data, we are entitled to fit only three independent functions. The fourth relationship, then, must be derived from the other three and tested for fit. As in all data fitting, the forms selected must also pass a test for conceptual reasonableness.

We stress that, in general, any statistical goodness of fit test is insufficient to establish any relationship as an element of the total model - as an expression of causal relationships - unless it can be convincingly interpreted in the light of one's insight into the process. Ultimately, it is only through the interplay and iteration of observation, modelling, and interpretation that real progress can be made in understanding and mastering the large-scale programming process (8.3).

8.2 (Eds) *We note that even now, in 1984, no real progress has been reported on defining adequate software complexity measures. A full discussion will appear in [BEN84].*

8.3 (Eds) *It is perhaps this realisation of the essential role of interpretation in terms of more fundamental phenomena that distinguishes this study from other phenomenological studies Software Physics, for example.*

2.5 Nature of the Relationships

The statistically derived relationships to be presented here comprise a model of the programming process with respect to this system's life cycle. The relationships represent a simple, but recognisably incomplete, model of what is happening. In practice, the statistical model has been used to improve the planning for this particular system. With the insight gained from the model's development, further statistical and analytic models have been and will continue to be developed that may explain the process and eventually lead to the insight that permits improvement of process planning, control and cost/effectiveness.

In the first instance, we must identify the global nature of the process as expressed in the relationships to be, or that have been, developed. The previously stated Third Law suggests that smooth long-term trends can be seen in the measures even if short-term behaviour tends to be erratic. This is supported by the fact that we have been able to construct statistically significant relationships consisting of three parts; the first expresses the long-term, deterministic trend; the second describes short-term cyclic effects; and the third part expresses any system-relative stochastic influences on the process.

The stochastic influences arise, in part, from a certain arbitrariness in the selection of the new function and, therefore, new code to be included in any given release. It is influenced to a significant degree by the user and management pressure, the availability of new hardware devices, and by business considerations that are not directly related to the internal dynamics of the process. Equally, the release target date, and hence the age of the system at the release point, is strongly influenced by factors external to the programming process.

The cyclic trends that we have observed in the data, and that have long been accepted on a heuristic basis by managers and observers of programming practice, may well contain the clue to current limitations of the process. In part, at least, inter-release effects arise from the interaction of repair and enhancement activity, particularly when they share common resources and are undertaken in parallel. It is probably the interplay between the levels and rates of the various activities and, in particular, their divisions at any given time between repair, functional improvements and new capability additions that charts the fate of a programming

system. Long-term trends, however are perhaps of greatest significance in understanding the process and in foreseeing and influencing the future. It is this effect that we shall mainly stress in our analysis.

Figure 5 shows the size of OS/360 in modules plotted with respect to release sequence numbers. Relative to the non-uniform time measure, growth in size is more or less linear. Indicated by arrows around the linear trend line is a visible ripple. This cyclic effect can be understood if the total organisation is viewed as a self-stabilising feedback system. That is, the design-programming-distribution-usage system has a feedback-driven and controlled transfer function and input-output relationship.

Some feedback results, for example, from constant pressure to supplement system capability and power. As the growth rate and work pressures build up, thereby increasing the size and complexity of the operating system, reduced quality of design, coding and testing, lagging documentation, and other factors emerge to counter the increasing growth rate. Sooner or later, as indicated by the segments marked C, these lead, at best, to a need for a system consolidation, a release that contains little or no functional enhancement and in which correction, restructuring, and rewriting activities predominate. As a result, the system size does not grow significantly during such a release and may even shrink. At the worst a fission effect F may occur, as at RSN = 20 to 21 where excessive prior growth has apparently led to a break up of the system.

Figure 6 presents the net growth of OS/360 in each release. Analysis confirms the cyclicity of the growth process as indicated in the Figure. A second observation may, however, be of even greater significance in estimating the limits of growth. With three exceptions, the net growth points may be seen to lie in a band bounded at about the 400-module level, a level that does not appear to have changed significantly in size during the lifetime of the system. Moreover, in the three instances where this growth level of OS/360 was exceeded, the record shows that, in the first case, the release was of such quality that it had to be followed by an unplanned clean-up release. The later two cases had equally unplanned consequences, significant schedule slippages, relatively disappointing performance, and - in the case of release 20 - the previously unplanned division of the operating system into at least two independent systems. Moreover, note that releases with net growth near or in

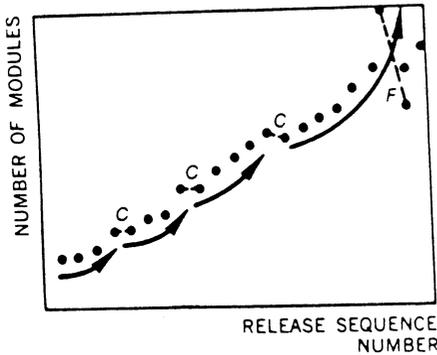


Figure 5 History of Growth
in Number of Modules.
C: Consolidation effect;
F: Fission Effect

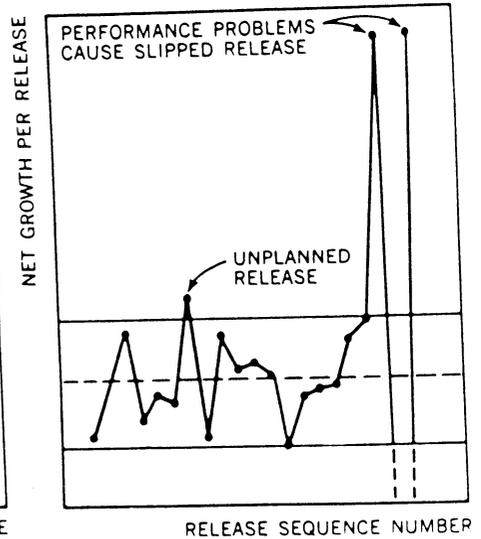


Figure 6 Cyclic Nature of
Net Growth of Operating
System Releases

excess of the indicated bound tend to be followed by one or more releases with a much reduced net growth.

If we may generalise our conclusion, it is that a large system grows through the addition of new and modified code, the system requires the regular establishment of a unique base reference to both code and documentation, such as is attained when the system is to be released for significant usage outside the development and maintenance group.

Also, in the present state of the art, complete and unambiguous specifications of changes or additions to be made are not normally achieved or even achievable. Nor is it possible to continuously prove the specifications to be consistent, and their subsequent implementation to be correct with respect to the new program behaviour desired (or even with respect to previous program behaviour). Hence the code and the system are tested. But tests can reveal only deviations from desired or expected behaviour [DIJ72b], they do not demonstrate absolutely correct behaviour or the absence of faults. Furthermore, the extent to which testing

reveals deviations or faults is limited by both the resources that can be consumed to conduct them and by the view that test designers and interpreters have of the total program, the changes, and the intended behaviour of both.

Thus, a further intrinsic consequence of a system release is that the program is suddenly exposed to an environment in which both the expected behaviour and the actual usage may - and usually do - differ from that to which the system was exposed in the development, maintenance, and test environments. Inevitably, therefore, release of the code results in the discovery of new faults. We conclude that sufficiently early release to users of stabilised code and documentation prevents a build-up of undiscovered faults. On the other hand, too many code changes that are undertaken without exposure to a wider usage pattern than can be generated in any test shop causes an accumulation of interrelated faults and system weaknesses, such as poor performance, that are far more complex to unravel. The data on which Figure 6 is based suggest that there existed a nonlinear effect with a critical growth mass in the operating system we are discussing of some four hundred modules.

This critical growth mass had been essentially invariant in almost a decade of OS/360 project and system life, despite methodological and technological improvements: increasing use of high-level languages and programming support tools; and increasing experience of designers, implementors, and management. Thus the characteristic is likely to be an attribute of the entire organisation that relates to this system. That is, we appear to have identified a combined system and metasystem invariance. In view of the posited multiloop feedback nature of the process, one can expect to change and improve this characteristic growth rate only when one begins to understand the structure of the process and its relationship to the organisation and to the system.

Without speculating further about the nature of the process, we may represent its invariance as observed in the present data by the following relationship:

$$*M_R = K_{11} + S_1 + Z_1 \quad (1)$$

or by

$$M_R = K_{10} + K_{11}R + S_1 + Z_1 \quad (2)$$

Here, $*M_R$ represents the net growth of the system between (RSN) = (R - 1) and (RSN) = R. A least-squares fit to the available data yields values of 760 and 200 for K_{10} and K_{11} ,

respectively. The S and Z terms represent the cyclic and stochastic components whose nature and magnitude can be determined using statistical techniques, such as those described in the literature [BOX70]. The small number of available data points, however, restricts the possible significance. We note that Equations (1) and (2) reflect directly the First and Third Laws proposed in the introduction of this paper.

In the absence of a more satisfactory measure, we represent the complexity of the activity required during the interval preceding release R by the fraction C_R (of modules of the total system) handled. Figure 7 shows this measure plotted against RSN.

One possible (and least square-wise significant) fit is by a quadratic in R. Other functional forms (particularly an exponential fit) are also significant. Both the quadratic and exponential representations appeal to our need for models and limitations on the program development process, but more data will have to be obtained to determine the one that more closely reflects a particular process. On the basis of the principle of parsimony [BOX70] select the following quadratic form for the current model:

$$C_R = K_{20} + K_{21}R + K_{22}R^2 + S_2 + Z_2 \quad (3)$$

For the present data, K_{20} , K_{21} and K_{22} are respectively 0.14, 0, and 0.0012.

We note immediately that the monotonic growth trend implied by Equation 3 supports the Second of our three Laws. The Third Law is once again supported by the identification of a significant trend.

Notice that the residuals for this quadratic fit, and equally those for an exponential fit, are generally rather large for $R = 2$ through, say, $R = 14$. This variation is, of course, absorbed by the cyclic and stochastic terms, but in fact the residuals correlate very strongly with the handle rate HR_R . This correlation is not statistically conclusive, since both measures are in the present instance derived from related parameters. Nevertheless, it suggests a more complete representation of the following form.

$$C_R = K_{20}^1 + K_{21}^1 R + K_{22}^1 R^2 = K_{23}^1 HR_R + S_2^1 + Z_2^1 \quad (4)$$

where a least squares fit to the present data yields the

values 0.037, 0, 0.0013, and 0.008 respectively, for coefficients K_{20}^1 , K_{21}^1 , K_{22}^1 , and K_{23}^1 .

An interpretation of this model suggests that more rapid work leads to greater pressures on the team, and hence to more errors - which, in turn, require greater repair activity. The data indicate that this is mainly incurred in the same release rather than discovered and undertaken thereafter. Furthermore, since it appears to lead to an increase in the fraction of the system handled, it suggests that the maintenance teams tend to remove the symptoms of a fault rather than to locate and repair its cause. This deduction has been confirmed independently by a number of observers of - and participants in - the process, a fact that strengthens one's confidence in Equation 4 as a more complete representation of one aspect of the process.

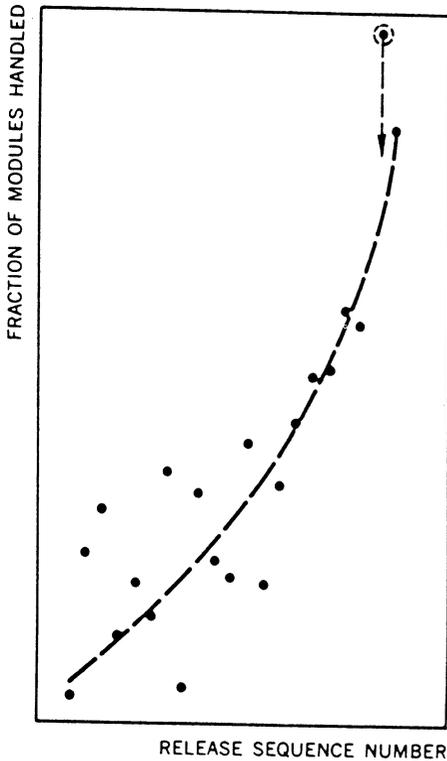


Figure 7 Complexity Growth during the interval prior to each release

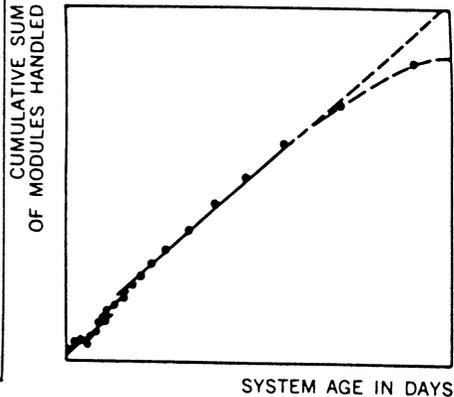


Figure 8 Handle Rate of Modules over System Lifetime

3.6 Work Rate

The work associated with each release is measured in this instance by modules handled MHR . This measure is, in each case, associated with a particular release and also with the release interval that separates the release from its predecessor. However, many releases overlap - particularly those releases that include major functional growth - and a new release may be integrated successively against two or even more predecessor releases.

Data on the degree of overlap between the various releases were not available to us. Therefore we first examine the cumulative sum of modules handled (CMH) as compared with the age of the system, in an attempt to neutralise the overlap effect in determining the handle rate. Figure 8 shows these data fitted, as a first approximation, by a straight line. Such a fit suggests that the major changes that have occurred during the lifetime of the operating system in methodology, tooling, and staffing levels have had no significant impact on handle rate. This has stayed essentially constant over the period at some eleven modules per day.

The data at the extremes of Figure 8 suggest that in the early life of the system, and in the most recent two releases, the handle rate may have been a little lower. This can no longer be confirmed for the older data. As far as present trends are concerned, however, since the handle fraction is approaching unity, we expect the slope of the cumulative handle plot versus system age to drop off from its previously constant value. It appears that even though the straight line fit is adopted as an initial model, an S-curve provides a more faithful representation over the life to date of the operating system.

We may now usefully examine the handle rate HR_R as determined by the ratio of the handle-to-release interval for each release, as shown in Figure 9. Because of the effect of release overlap, the range of rates achieved is exaggerated, but it is indeed centered around an average of about eleven modules per day. Also note that where the release rate has exceeded this average the figure for the next release is lower. We conclude from the data for Figures 8 and 9 that the handle rate is stationary with cyclic and stochastic components that are confirmed by analysis to be significant and have a three-release cycle.

Thus we adopt as our third relationship an expression of the following form:

$$HR_R = K_{31}^1 + S_3^1 + Z_3^1 \quad (5)$$

or

$$CMH_D = K_{30} + K_{31}D + S_3 + Z_3 \quad (6)$$

CMH_D counts the total number of modules handled between the first release of the system and day D, that is, when its age from release 1 is D days. HR_R represents the module handle rate in the Rth release interval. The S and Z terms once again represent the cyclic and stochastic components. For the present system, K_{30} and K_{31} are 1100 and 11 respectively. The statistically significant determination of a long-range trend with cyclic and stochastic components once again confirms the proposed Third Law.

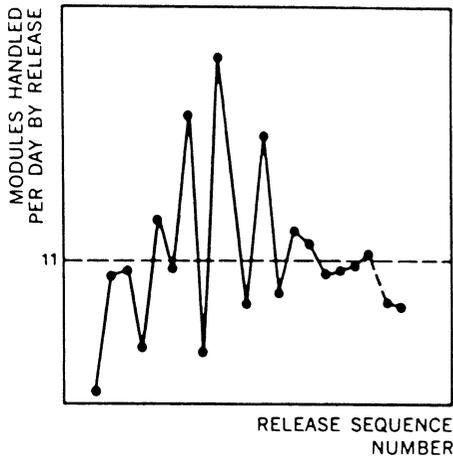


Figure 9 Handle Rate as a function of Release Number

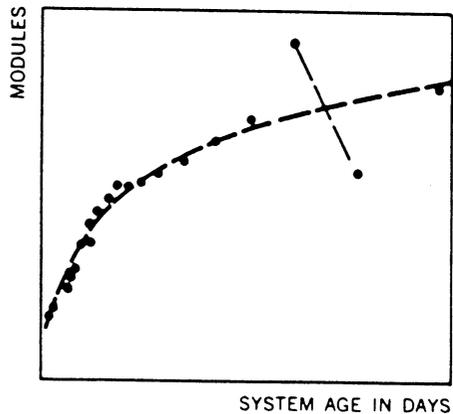


Figure 10 System Size as an indication of Declining Growth Rate

2.7 System Size

We must now consider the data of Figure 5 which we have presented as a function of system age in days. As indicated earlier in this paper, the relationship developed to represent this third trend must be compatible with those already expressed in Equations 1 through 6. Of the alternative forms that can be significantly fitted we have selected the following expression:

$$M_D = K_{40} = K_{41} \log (1+D/K_{42}) + S_4 + Z_4 \quad (7)$$

Here, a least squares fit yields K_{40} , K_{41} , and K_{42} as 89, 1350 and 51 respectively. The value of the intercept is not significant because the representation is not meaningful where D approaches zero. In reality, of course, system age was not zero at the time of $R = 1$, which is the assumed origin of our time scale. Nor, in view of the assumption that the build and maintenance processes are intrinsically different, may we expect to express the actual system age at first release in the same terms, even if this were known.

We note that the logarithmic representation is not asymptotic. Nevertheless, it suggests unlimited growth potential, though at a decreasing rate. This corresponds to our intuitive understanding that, as a system ages, it is always possible to change another instruction or add another module. However, the time required to do this tends to increase, unless the system is restructured and cleaned up.

One further observation of interest follows from the logarithmic representation selected. This representation is compatible with the constant incremental growth implied by Equation 1 provided that the release interval is growing polynomially or, in the limit, exponentially. But this is precisely the behaviour of interval growth, as shown in Figure 11. As it so happened, the earliest and very successful forecasting undertaken by us was based on this very observation and on the resultant exponential fits to the data.

2.8 Summary

Equations 1 through 7 provide a model of the maintenance process for the operating system, OS/360, based on five parametric concepts, but with only four available measures. The model would be complete with the determination of the statistical parameters of the cyclic and stochastic terms.

The small number of data points, however, precludes the determination of significant values.

Recognising the essential interdependence of the various parameters, one can also gain in descriptive power by determining compatible multivariate relationships such as are shown in Equation 4. These relationships could, of course, involve additional or lower-level breakdowns of existing parameters.

The number of basic relationships presented has been deliberately restricted to the number that is necessary and sufficient with respect to the existing degrees of freedom. Equations (1) through (4) have been selected because they bring out apparent invariants of the process. The recognition of invariances is fundamental to the application of the scientific method. As such invariant detection in an analysis of the programming process not only strengthens our basic assumption of regularity in the process development, but it also provides hope that the analysis can be further developed and eventually permit improvement of the process.

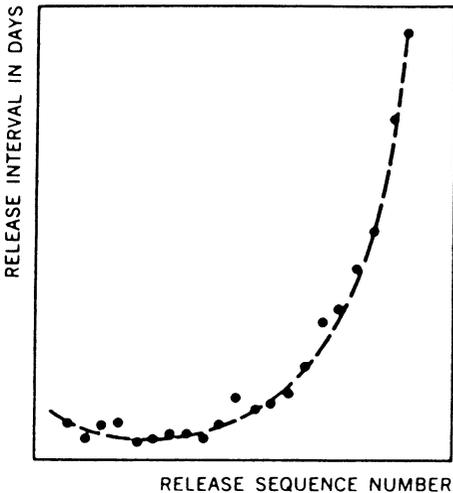


Figure 11 Increasing Release Interval

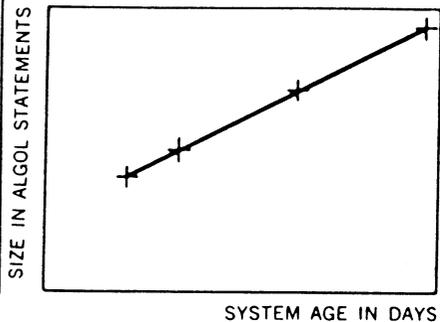


Figure 12 Constant Growth Rate Exhibited by a Second Operating System

Although the present model *represents* the observed behaviour it does, however, not *explain* it. Moreover, the representations break down at the extrema of observation. We have commented on this in the case of Equation 8 when D approaches zero from above. Similarly, Equations 3 through 6 are seen to be invalid representations as the fraction handled approaches its intrinsic limit of one. In fact, the expected nonlinear trend is visible in Figure 8. Good reasons have been given, however, for expecting a constant handle rate to be valid over the major portion of the interval considered. Thus it is not surprising that forecasting and planning techniques based on these representations have been useful in providing accurate data to improve planning in this particular environment.

It now appears that further development of statistical process models should be directed toward an examination of the behaviour of other systems from both IBM and from other program development organisations, so as to determine the range of applicability of the observed phenomena. First confirmation has come from data on a second though smaller operating system that originated in the same organisation. With minor differences, this operating system shows the same characteristics and trends, though with markedly different parameters. Preliminary data from a totally different organisational environment have also been examined [M0075]. As indicated in Figures 12 through 14, the smaller operating system confirms the basic observations of constant growth

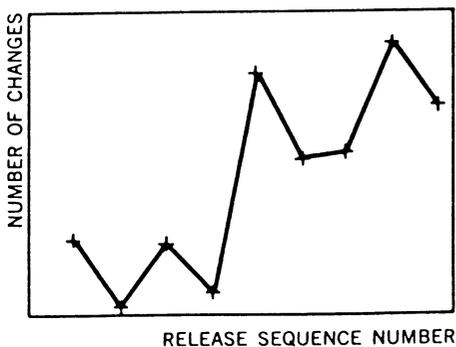


Figure 13 Number of Changes as a Function of Release Number of a Second Operating System

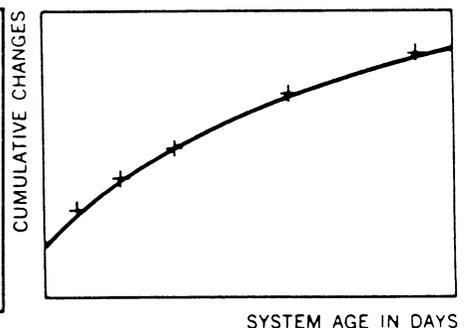


Figure 14 Declining Work Rate Exhibited by a Second Operating System

trends, cyclicity, overall smoothness, and declining work rates. The confirmation that this implies is of particular interest because the source is a programming organisation outside IBM that created structured programs in ALGOL for execution on non-IBM machines. Thus the organisational environment is quite different, but the phenomena are visibly present.

Clearly, these data - especially the invariants - should be studied further, for example by examining actual work rates within a release interval. With further study, one hopes to discover the reasons for the phenomena and ultimately to remove the limitations that they imply.

In parallel with the study of invariants, one should also proceed with the development of abstract models that represent and formalise our perception and understanding of the large-program development process itself or of aspects of the process. We describe examples of our earliest approaches to this problem in the following section.

3 Formal Modelling of the Program Development Process

3.1 Program Faults

Since our goal is to understand and to learn to control the programming process, one view of the process is to see it as the interaction between two entities. On the one hand, the large program in all its representations and with its documentation we call the 'object'. On the other hand, the human organisation that implements the process in its manipulation of the object is termed the 'team'. The function of the team is to execute changes in the object.

In conjunction with user-provided data, the object enables a computing machine to perform useful work. During its lifetime, all kinds of changes to the object are necessary. The (hardware) machine, or some of its components may be changed or replaced. New devices may be added. Computing requirements may be redefined to serve new users. New ways of using the system may be devised. In general, the behaviour of the system deviates from that anticipated or desired because of *faults* in the system. We term faults related to changes in the environment *defects*, whereas an *error* relates to the difference between actual and anticipated behaviour. When faults manifest themselves, the team is required to undertake corrective action, to perform changes on the object.

Observations related to those discussed in the previous section suggest that system evolution is to some considerable extent influenced by fault repair activity. Our earliest formal models, therefore, have been designed to examine fault distribution in the system. These models were based on the following assumptions:

- * Changes, that is, object handlings, are, in general, imperfect. When changes are performed, errors are injected by the team with probability greater than zero. This by itself would imply a continuous need for change, even if the environment were fixed.
- * There is a delay between the injection of an error and its first detection and recording, and another delay exists between recording the error and its final elimination.
- * Some errors are ordered in that one of them must be repaired before the other can be detected. That is, there is a layering of errors in the object that is representable by a directed graph.
- * The team creates and uses documents, which are kinds of representations of the object, to study faults and possible courses of action. The documentation may be viewed as an integral part of the object.
- * Team members, while involved with changes, communicate with each other in the language of these documents.
- * Team members have to be educated in the documentation; moreover, the team has the additional task of updating the documentation to reflect changes performed on the remainder of the object.
- * Deficiencies in documentation influence the effectiveness of the process and, therefore, cause deficiencies in the object.

From these assumptions, we have developed two classes of models. The first emphasises the internal distribution and propagation of errors in the object. The role of the team is simply to eliminate observed faults.

The second class of models gives the team a more active role. Management is free to make decisions as to those particular tasks, error repair, documentation, or other activities to which the team should turn. The object responds to these actions by manifesting different error generation rates.

3.2 Model of Fault Penetration

The model of fault penetration that we now discuss is a measure of complexity due to aging. Consider an elementary change activity in the time interval $(i, i+1)$. This is depicted in Figure 15, where the width of each arrow band may be interpreted to be proportional to the number of faults it represents.

At time i , a number of faults is assumed to exist. As a result of team activities, the following occurrences are likely:

- * A fraction E of the total faults is removed (extracted).
- * New faults G are injected (generated) due to imperfection in the activity.

Thus at time $i+1$ a new composition of faults appears that consists of residual R and newly generated errors.

Preserving the distinction between residual and newly generated errors is fundamental to an understanding of the evolutionary process. A system cannot be effectively maintained if that distinction is not understood. And complete understanding demands a knowledge of the history as well as of the state of the object at all times.

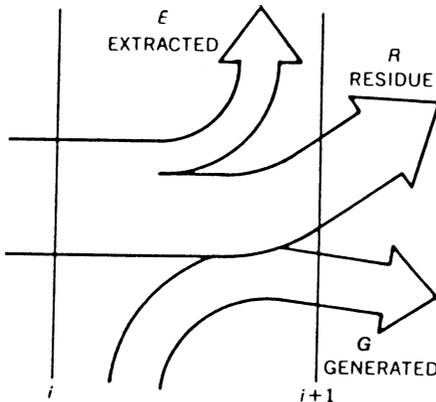


Figure 15 Primitive Model of Fault Penetration

The primitive change activity of Figure 15 spans the network of Figure 16, where i is a discrete measure of age (or release number) and j is a variable used to introduce the tree structure. For each node, the residual design faults R and the generated faults G may be expressed as follows:

$$R_{i-1,j} = R_{i,2j-1} + E_{i,2j-1} \quad (8)$$

or

$$G_{i-1j} = R_{i,2j} + E_{i,2j} \quad (9)$$

and $G_{i,2j-1}$ and $G_{i,2j}$ are to be defined for each node by the following additional assumptions:

- * $G_{i,j} > 0$ (imperfection hypothesis)
- * We define $C_i = 2^{i-1}$ *fault classes* for every i . Each class has a unique label that consists of a two-valued $\{R,G\}$ character string of length i , with the first element always R , meaning residual design faults. For example, $R R G R G G R$ represents a node or fault class at $i=7$. More specifically, faults in this class are the:-
 - residue (... R) of
 - faults generated (... $G R$)
 - while extracting faults generated (... $G G R$)
 - while extracting the residue (... $R G G R$) of
 - faults generated (... $G R G G R$)
 - while extracting the residue (... $R G R G G R$) of
 - faults in the original design ($R R R G R G G R$)

The model as described represents an increasingly large and complex network of fault trajectories or histories, even though the total number of faults present may have been stable or even declining as a consequence of non-zero E_s . Faults are identified in terms of unexpected or undesired system behaviour in execution. Thus we have excluded from consideration here simple faults that manifest themselves locally in a single element of the system. That is, we may omit from consideration those faults that may be detected or removed by operating with or on any one element alone, and consider those situations where rectification of a fault requires coordinated changes in two or more system elements and in their interfaces. Interactions among inter-element and inter-generation effects represent the conceptual complexity of the fault pattern. And it is the increasingly complex fault structure that underlies increasing object complexity.

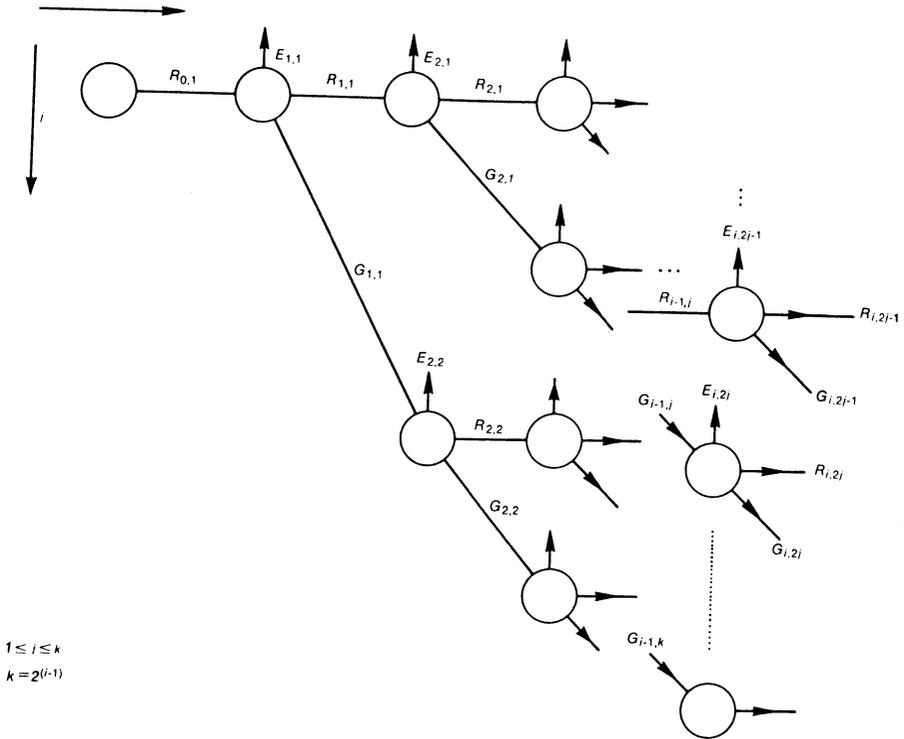


Figure 16 Network Showing Faults Extracted and Faults Generated

Thus periodic restructuring of the object is necessary to reduce complexity because increasing object complexity is itself a fault that impinges on the maintainability of the system.

The connection between the relational complexity of errors and the structural complexity of the system implies that relational complexity may be a measure of communication requirements for the team and the underlying cause of fault extraction and generation over the entire lifetime of the object.

We now give a quantitative interpretation of the fault penetration model, which is a simplified view of structural aging. To analyse the above fault generation model so as to obtain even a simplified view of the resultant ageing,

additional assumptions must be made about the fault extraction and generation variables E and G.

The simplest hypothesis is that, for each node,

$$E = G \quad (10)$$

that is, as many faults are extracted as are generated. Under this assumption the system appears to be in a steady state.

Let us consider the number of fault classes C_i as a measure of complexity of the system. Analysis has shown that complexity increases even in steady state, that is, when the number of faults in the system remains constant.

A degree of freedom can be eliminated by establishing a relation between fault extraction and the fault content of a given class. A reasonable assumption could be that $E_{i,2j-1}$ is in fixed proportion to $R_{i-1,j}$, and no new errors are generated. Thus we have the following fault elimination-to-fault residue ratio:

$$\frac{E_{i,2j-1}}{R_{i-1,j}} = (1-p) \quad (11)$$

and fault decay follows a geometric distribution with parameter p which is constant such that $0 < p < 1$. After i intervals, and having started with a given collection of faults S , the remaining number of faults in the original collection is $S(1-p)^i$, whereas $S(1-(1-p)^i)$ faults must have been extracted. Since $G_i \equiv 0$ for all nodes, the system approaches an error-free state asymptotically (approximately exponentially). Thus in all cases considered, the geometric distribution reflects the reasonable assumption that the smaller the fault content the fewer the faults there are to be discovered and extracted. This, however, still implies a monotonically increasing C_i until, if ever, a fault-free state is reached.

More elaborate relations between E and G may be required, so as to represent currently observed situations.

It is important to note that E and G at each node are not independent, but are coupled via the team and the process.

3.3 Qualitative Interpretation of Fault Penetration

As already indicated, even with decreasing fault content ($E > G$), the complexity measure C increases monotonically. This results from and reflects the increasing stratification of the system because of the increasing heterogeneity of faults.

The resultant structural deterioration experienced as an increasing difficulty in executing change alerts the team to the need to counteract the aging process. On the basis of our previous assumptions, the latter may be considered proportional to $2^{G(i)}$, where $G(i)$ is a monotonically increasing function of i that reflects higher-order variations not considered here, as well as the complex relationship between fault and system structure.

To cope with the situation, the state of the system has to be precisely defined. Documentation must be accurate, complete and accessible. In addition, the administrative organisation or responsibility of team members must be well defined. Finally, team members must be aware of the state of the system by learning. Fulfillment of these needs can effectively reduce the effect of growing complexity, and can be represented symbolically as follows:

$$C_i \text{ (modified)} = \frac{2^{G(i)}}{2^{DAL(i)}} = 2^{G(i)-DAL(i)} \quad (12)$$

where DAL means 'Documentation, Accessibility, and Learnability' which are constructive factors. Equation 12 is a qualitative one, and one that is closely related to our earlier fault penetration model [BEL76], [LEH74]. Real-life situations are much more complex. Communication complexity required to overcome system stratification may, for example, be further increased by geographic scattering of the team activity. Nevertheless, the model enables one to address some very real questions about the program maintenance process. For example, since the model mirrors a domain that is discrete (indexed with i), the model suggests that perhaps increasing the number of intervals i (ie, decreasing the inter-release time) should permit faster extraction of faults. This would occur if such an increase were to imply an increase in the frequency of restructuring and of providing adequate team knowledge of the state of the system. That is, $G(i)$ and $DAL(i)$ must be kept in step.

by no means clear. As a consequence of one of our early assumptions, namely, that faults are layered and manifest themselves in a partially ordered fashion, one has to go through the process of gradually repairing the system, with the inevitable result of generating complexity. In addition, short intervals provide less opportunity to exercise the system in actual use for fault manifestation, thus reducing the number of faults that can be extracted. The size of the optimum interval is, therefore, undecided. A more detailed model is required if this is to be formally explored with the objective of helping solve a problem that arises in real system development.

3.4 Management Decision Model

We now discuss our management decision model, which reflects our earlier formulation [BEL72, 72b], [LEH74] and which is based on the following assumptions

Budget B, the available budget, bounds the total activity. During the change process, every unit of fault extraction (termed 'progressive' P) activity, measured by $G(i)$ in the model given by Equation 12, is associated with a certain amount of documentation, administration, communication, and learning activity (termed 'antiregressive' A) as measured by $DAL(i)$ in Equation 12.

Neglect of A activity results in the accumulation of additional work demand to cope with increasing complexity C. This cumulative demand can be removed only by a (temporary) increase in the intensity of A, which, as a result of the limited budget B, causes a (temporary) decrease in progressive activity P.

Management is assumed to have full control of the allocation of its resources and the division of effort between P- and A-type activities. Management cannot, however, directly control the growth in complexity that accumulates, except by utter concentration on complexity control through restructuring. This is an activity that is strictly antiregressive and, as such, is psychologically difficult to inspire, since it yields no direct, short-term benefits [LEH74].

To examine these concepts further, we now present an alternative formulation of the model. In a somewhat simplified fashion, we assume that resources are fixed (by budget) and that they are equally applicable to either P or A

activity. B and activities P, A and C can be measured in cost per unit of time, which express the budget rate and its expenditure rate on progressive, antiregressive, and complex control activities, respectively. In addition, we use the following relationships:

$k = A/P$ represents the inherent A activity required for each unit of P activity so that complexity does not grow.

$m =$ management factor, which is the fraction of progress kP that is *actually* dedicated by management to A activity

At any time, the total expenditure on all activities must be equal to the budget, hence the formula for the budget is given as follows

$$B = P + A + C \quad (13)$$

The formula for antiregressive activities is

$$A = mkP \quad (14)$$

and

$$C_A = \int_0^t (1-m)kP dt \quad (15)$$

where

$$C_A(t_0) = 0$$

The expression C_A or complexity reflects the cumulative decay caused by the neglect of A activity.

Since the values k and m are left free to vary with time, the model can be used for the investigation of the consequences of various possible management strategies in controlling the maintenance process. Further freedom can be introduced by inserting variable-length delays among the three major expenditure components. A large problem space thus results that can be explored by interactive modelling for increased insight. In this environment, real-life observed phenomena can be approached in the model by stepwise changes in model parameters.

3.5 Management Simulation

A graphic modelling facility has been used by the authors. This system was essentially an analog computer that was implemented on a digital machine such that the analog

components (delays, adders, integrators, etc) could be connected into a network on a cathode ray tube by the use of a lightpen. Upon request, the computer accepted the network and numerical parameters as inputs for a stored program. The system then computed the response, as described in [BAS68] and [IBM].

During the numerous experimental sessions with this facility, many real-life phenomena were successfully reproduced. One example was the cyclic pattern of object growth for the statistical model discussed earlier. The network consisted of a nested two-loop feedback system; present threshold values for k simulated the management decisions.

More precisely, in our simulation, after a period of persistent neglect of A-activities ($m < 1$), management becomes alarmed by the rapid reduction of P due to increasing C. Consequently, an increase in A is scheduled ($m > 1$) until the situation noticeably improves. At this point, management again becomes optimistic and relaxes k to a lower level. In the long run however, C grows monotonically. A sample output of a run is presented in Figure 17.

The authors are convinced that this type of interactive modelling is perhaps the most fertile, and certainly the fastest, way of developing a feel for the interactions involved, and gradually developing a more complex model that has the power of predicting real-life behaviour.

In contrast to previous models, management decision modelling yields an optimistic prognosis, since it includes parameters that reflect management discretion. Thus it permits the counteraction to remove the consequences of growing complexity, action that occurs in real-life situations. On the other hand, of course, the model does not reflect the internal structure of the object. In our earlier models, internal structure was modelled by combining the management model with an extension of the fault penetration model.

3.6 Model of Limited Growth

Suppose that management is free to allocate resources to grow the object, as well as to extract faults as in the previous model. Of course, both activity classes are essentially imperfect in that, while performing them, errors are injected into the object.

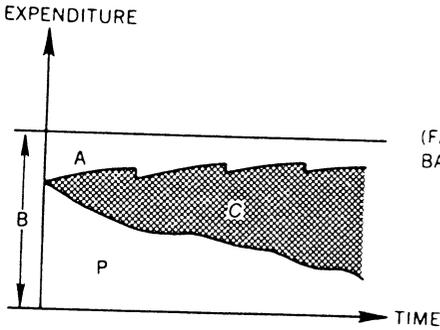


Figure 17 Example Output of a Budgeting Simulation

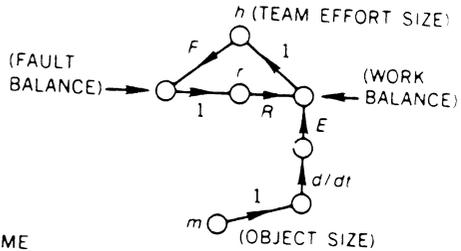


Figure 18 Model of Growth in the Presence of Error Generation that is Proportional to Effort

As the simplest case, we would like to show how the size m of the object, measured, for example, by the number of modules it contains, develops in the presence of error generation that is proportional to growth activity. In signal-flow graph form, the linear relations can be represented by Figure 18. Here E and R convert growth rate and error repair to work demand (measured in man-hours). F is the error generation rate (the number of errors per man hour) and r is the number of errors.

The corresponding equations are

$$h = Rr + E \frac{dm}{dt} \tag{16}$$

$$r = Fh \tag{17}$$

Assuming a constant work force h , the solution is given as follows:

$$m = m_0 + \frac{1-RF}{E} ht \tag{18}$$

where growth is a linear function of time. The greater the work force and the smaller the error generation, the more rapid is the growth, which is, in principle, unlimited. The reason is that, on the basis of our previous assumptions, the effort not used for repair is available to grow the object at a rate that is independent of its size.

Observations on our previous models, however, have suggested that larger and older objects are more complex and receive more errors as they evolve, as a result of growth and of fault removal. Retaining the linear character of the relationships, the flow graph given in Figure 19 represents the modified assumption, namely, that increasing size causes more errors to be generated, with gain D per unit size. The somewhat modified equations appear as follows:

$$h = Rr + E \frac{dm}{dt} \tag{19}$$

and

$$r = Fh + Dm \tag{20}$$

where Equation 19 represents a negative feedback to control size. The solution now becomes the following:

$$m = m_{er} \left(1 - \frac{1}{m_{er}} e^{-\frac{RD}{E}t} \right) \tag{21}$$

$$m_{er} = \frac{h(1 - FR)}{RD} \tag{22}$$

Equations 19 through 22 indicate that under the assumptions of this section growth is limited to m_{er} . This critical size can only be reached asymptotically. The reader may be wise to compare this result with the real-life observations previously reported.

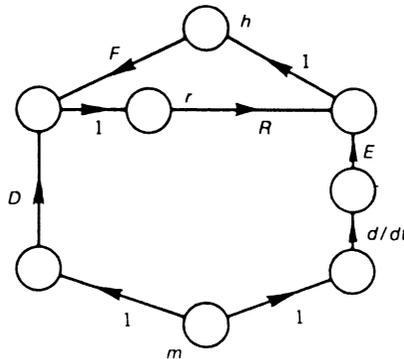


Figure 19 Model of Limited Growth in which Increasing Size Generates Increasing Errors

The critical size can be increased by increasing the size of the work force. However, this means that a subsequent reduction of the work force can create a new critical size that is smaller than the one already reached. Thus a situation of monotonically increasing error content is created.

This model has been studied under differing assumptions. The main conclusion remains, however, that object size is limited with even the slightest negative feedback of size.

This section has presented several models each throwing a different, though related, light on the program maintenance or enhancement process. Our aim has not been to present completed models. Rather, we have wished to illustrate how the modelling may be approached, and how interpretation of the may be used to study and to improve the programming process.

4 Concluding Remarks

Currently, the process of large-scale program development and maintenance appears to be unpredictable; its costs are high and its output is a fragile product. Clearly, one should try to reach beyond understanding and attempt to change the process for the better.

As a first step toward ultimate improvement, we are studying the process as it is, and as it is evolving, much as the physicist studies nature. Our immediate goal is an organised quantised record of observations that formalises the perception of what is happening and what is being done. With such global studies, one may hope to identify specific points or sources of trouble and perhaps identify areas of the process that are major causes of concern. When *what* is happening is understood in the context of the process as a whole, one may attempt to understand *why* it is happening. Only then should one attempt to change the process without risking local optimisation that is very likely to reduce significantly the degree of global optimisation. At the present time, for example, it is not clear to what extent improvements should be sought by attention to the human organisation, management, or by emphasis on the product side of the process, in order to achieve the most significant gain in and from the process.

We do speculate that communication is a major problem. If this can be confirmed then, for example, a design methodology

that expresses the understanding and intention of the designer unambiguously and completely might eliminate many difficulties (8.4). One may also hope to avoid problems in the performance area as a consequence of overspecification. Thus one might equally consider that a reduction in product complexity, by better partitioning, for example, could lessen the need for communication and, at the same time, improve performance potential. To do this effectively, however, we must be able to identify those parts of the product that are most interlaced in their logical structure.

Our data so far have been largely limited to that of a few rather large operating systems that were produced within the same large administrative organisation. Even these data are meagre. Since the initial design phase, no one anticipated the long series of changes that was to follow the initial development. We now know much better and are able to specify the kinds of data that are necessary for future analysis of the development, implementation and maintenance processes.

We are also enlarging our scope beyond the environments studied so far. It is already clear that qualitative observations similar to ours have been made at other places where large-scale programming has been undertaken. This suggests an urgent need for the definition and standardisation of process measures to facilitate meaningful comparisons between dissimilar systems, processes and organisations.

Clearly, we still must test the generality of the hypotheses presented in this paper. It will, for example, be of major interest to determine the degree of generality and the range of validity of the various invariants discovered so far, after filtering new data to remove noise due to environmental factors. This should improve the usability of program evolution dynamics concepts and techniques as planning tools, an improvement much needed by managers who are, in general, not very successful in assessing, predicting and controlling schedules and resources in the software process.

It is important for an emerging discipline, such as program evolution dynamics, to summarise its most essential concepts into unambiguously defined and measurable quantities at each stage in its development. This makes it possible to use

8.4 (Eds) *Recognition of this point has, in recent years led to advocating the general adoption of formal specification*

appropriate techniques and tools from established disciplines. Mathematics, for example, facilitates comparisons between derived results and real life, and may even help the development and communication of new ideas.

One of the most frequently used - but as yet undefined - concepts encountered in our studies is that of complexity. Particular definitions that have been established in the somewhat narrow content of computational magnitude do not appear to be useful or applicable for the study of structure and interaction. After some preliminary studies, we have concluded that a measure of complexity, applicable to the large scale programming environment, could be developed by using established concepts that are related to information, uncertainty, and entropy. Further investigation in this direction forms an ongoing activity in the authors' groups.

Given a measure of complexity expressed in terms of simple structural properties - such as the number of interactions between product or organisational elements - normalised measures for programming effort, productivity, system reliability, and security can be derived and comparisons between different products or methodologies made meaningful. Without such a measure, many of the essential parts of the developing discipline remain unconnected and phenomena are easily misunderstood. An early result in the study, for example, suggests the consideration of complexity of software and its documentation in a unified fashion. In this case, the total project workload can be better quantified, and plans and schedules made more accurate, provided that the manpower need is strongly related to complexity.

Many the directions pursued in our exploration of evolution dynamics appear to relate to the global properties of complex systems rather than to properties that result specifically from the software environment.

Thus we assume that the results of our studies may be generalisable to other complex technological projects, and to the study of sociological, economic, and biological systems or organisms. In the immediate future, however, we shall concentrate our studies on the evolution of large programs, since in this area change is observable over a relatively short period of time, and experimentation is possible without the serious penalties that could be incurred in other fields. Thus program evolution dynamics may be interpreted as a suitable prototype or test bed for the study of more general system evolution dynamics.

5 Acknowledgement

The authors appreciate the contributions of their many colleagues in IBM and in the Imperial College, and in particular their discussions with Heinz Beilner and Lip Lim. The CSMP modelling was a contribution made by Steve Morse.

CHAPTER 9

PROGRAM EVOLUTION AND ITS IMPACT ON SOFTWARE ENGINEERING*

1 Introduction

Recent international conferences on Software Engineering [ICSE1] and Reliable Software [CRS75] considered many of the fundamental issues facing the programming community. Languages and specification, design and implementation techniques, correctness and program validation, operational reliability, programming team organisation and management all received attention.

One cannot question the immediate relevance of these topics or the validity of this basic emphasis. The absence however of any consideration of the ever changing environment within which all practical programming activity is undertaken and its impact seems an unfortunate omission. With perhaps only one exception [CIC75], the papers presented at these conferences appear to have ignored the sad but indisputable fact that unless a program remains unused, it will undergo a continuing sequence of changes, additions and deletions. And unless the greatest care is exercised these will inevitably adversely affect its structure, its correctness and all other program quality attributes. A proof of correctness, or any other means of program validation, is *relevant* only as long as the environment it assumes and the code it examines remain unchanged. Similarly a program must *remain* well structured through most of its life time. Having a sound structure initially is clearly necessary. But it is insufficient if the program is to be economically maintainable to remain functionally effective throughout its operational life. The inevitable structural decay during the normal maintenance and enhancement process must therefore be compensated by restoration activity. But such work yields, in general, no immediate performance or functional improvements. It merely *prevents* functional, performance, reliability and maintainability degradation. As a consequence, it attracts no high management priority when resources are limited and the emphasis is on quick economic returns.

The fact remains however that a program and the environment within which it operates both undergo continuing change and evolution. This phenomena is in practice so inescapable that after reliability, changeability is perhaps *the* most desirable attribute of large, widely used, multi-function programs (9.1). The profession and the literature has however largely ignored, or at least underplayed, the nature and significance of the continuing evolutionary maintenance process. Thus the first objective of the present paper is to provide quantitative evidence that throws some light on evolutionary patterns; on *program evolution dynamics*. The presentation includes a brief discussion of the nature and some attributes of the growth process. We shall then indicate how an understanding of the process can be exploited in programming project management, and why it is important for and in the future development of software engineering.

2 Program Evolution

Evolutionary growth of functional capability, in particular, is intrinsic to the life cycle of large, widely used programs such as operating systems. The continuing growth reflects a continuous increase in the potential power of a system. It may be approximately measured by, for example, the quantity of its code and documentation.

In its most general terms, the unending development of programs is exactly analogous to the evolutionary process that governs the life cycle of any complex system [LEH76]. The system interacts and interplays with its environment in mutual reinforcement approaching, rather than being initially implemented with, the desired characteristics. One cause of this is the fact that the state of the art does not, in general, facilitate a precise, unique and complete specification of a system and its functional capability. Hence the programming process cannot yield a first version that is perfect for the intended application. Subsequently as users exploit the capability of a system they discover or invent new ways of using it and new applications. This in turn suggests and encourages the provision of still newer facilities. When these are implemented, their use, in turn, suggests still other applications and usage patterns. In addition, advances in application and usage technology

9.1 (Eds) *These two concepts are, in fact, unified by a concept of 'Dynamic Reliability': a system must not only be correct in the first instance, but must be maintained correct as the operational environment changes.*

encourage and support hardware development and make it economically viable. Advances in hardware technology, in turn, bring with them new devices and advanced, more cost effective versions, of old devices. All this leads, once again, to usage advance and expansion. And all these developments bring calls for continuing revision and development of software support (9.2).

In general systems (ie, not software) the evolutionary trend of necessity effects successive *generations* of system elements. That is during each redesign phase, changes are made to the specification and design of what would otherwise have been a straight reproduction of ancestor systems. These changes are then implemented in a *new instance* of the system or element. In program systems on the other hand the irresistible temptation is to modify and augment *existing* code rather than to restructure, recode and hence recreate entire sections of the system. This despite the fact that programs in execution are most complex; totally unforgiving in the face of sloppiness or any logical inexactitude over the combinatorially large number of alternative input conditions and execution sequences to which its segments are repeatedly exposed at high speed. The need for program correctness, however defined, is absolute, yet we perturb, change and generally maul programs and code with reckless abandon.

The above broad outline identifies the phenomenon. Its consequences are familiar to anyone who has had program development, maintenance, marketing or usage responsibility or experience.

3 Program Evolution Data

For some years the present authors and their colleagues have been measuring and studying the evolutionary pattern of one large operating system, here identified as system T (9.3), [BEL71b], [BEL72], [BEL76], [LEH76b]. System T is typified by its size, by the richness of its function, by the variety of hardware devices and system configurations supported, by the enormous variety of users and application environments which it serves, by its major use of assembly level language

- 9.2 (Eds) *All the insights which later led to the program classification that identified E-type programs [LEH80c] are already present in this paragraph.*
- 9.3 (Eds) *Identified in [BEL76] as IBM operating system OS/360.*

and by the size and geographic dispersion of its development and maintenance teams.

As reported in the above references and elsewhere, this system has displayed an almost continuous growth in functional capability, in size and in complexity. But despite the fact that there is a continuing pressure for the addition of new capabilities to the system, its rate of growth has shown an equally persistent decline.

The question has been raised as to whether these observations throw light on the programming process itself, or whether they reflect a peculiarity of the particular program studied and the environment in which it was developed. More recently data has become available from quite different organisational environments which reinforces the view that many of the phenomena observed in the development of system T do occur more generally.

One new source of data [H0075] comes from the executive system here identified as system P. This system is an order of magnitude smaller than T, is written as a structured program in an Algol-like language and is being developed by a small, highly disciplined and well managed group under the executive direction of the organisation which is the sole user of the program. Operation is confined to two installations for basically a single purpose. In summary, the program and the environment in which it is being used and maintained are in almost all respects the total antithesis of T and its environment.

Similar phenomena may also be observed in data relating to transaction system A. This system, together with its maintenance and usage environments, lies midway between the other two in size and diversity of purpose. System A is being developed by a major computer manufacturer and is used by only a limited number of customers. Most significantly, data on this system was available only for a phase of the project during which time, according to the project managers, no 'major enhancements' of the system were undertaken. Thus the alleged maintenance phase for which quantitative data is available was preceded by a development phase during which the system was designed and implemented by a different team at a different site. System A like system T is coded in an assembly language.

The data collected from systems A and P is rather limited in extent. Neither set of information presented on its own

would provide a very sound basis for quantitative theories about the development process of software systems. But compared and contrasted with the data from system T a general picture of software evolution begins to emerge in which perturbations from a basic pattern correspond very naturally to particular features of the individual systems (9.4).

4 System Size

The most directly useful (and often most easily available) source of quantitative information about the development of a software project is a record of how system size has changed over time. Clearly there is a choice of measures of size - lines of code, bits of store, numbers of modules or components. In practice, it was not possible to present the data in terms of a standard size unit appropriate to all three systems studied. However, this failure did not seem critical since, (i) probably all size metrics appear strongly correlated, (ii) it is the pattern of size change which is of prime interest in this context rather than consideration of the absolute size of the systems and projects.

Plots 1 (a), (b), (c) illustrate the size histories of the three systems. All three systems display a clear tendency to increase in size; more significantly growth has occurred in each case long after the system was first released for use by its customers. The proportional system growth 'after release' is smallest in the case of system A. But even there the number of components making up the system has increased by 16% over the period observed; and that at a time when both management and the participants viewed their activities as maintenance with only minor enhancements.

9.4 (Eds) *Note here the assumption, underlying much of the development of the evolution dynamics theory that the search for basic phenomena and supporting theory may be guided by observed data and statistical analysis even if the former is sparse and the latter does not yield statistically significant results. It is the resultant theory that must satisfy the accepted tests of any scientific theory. It is, we believe, failure to appreciate this viewpoint that has generated and underlies criticism such as that found in [CHO80] and [LAW82].*

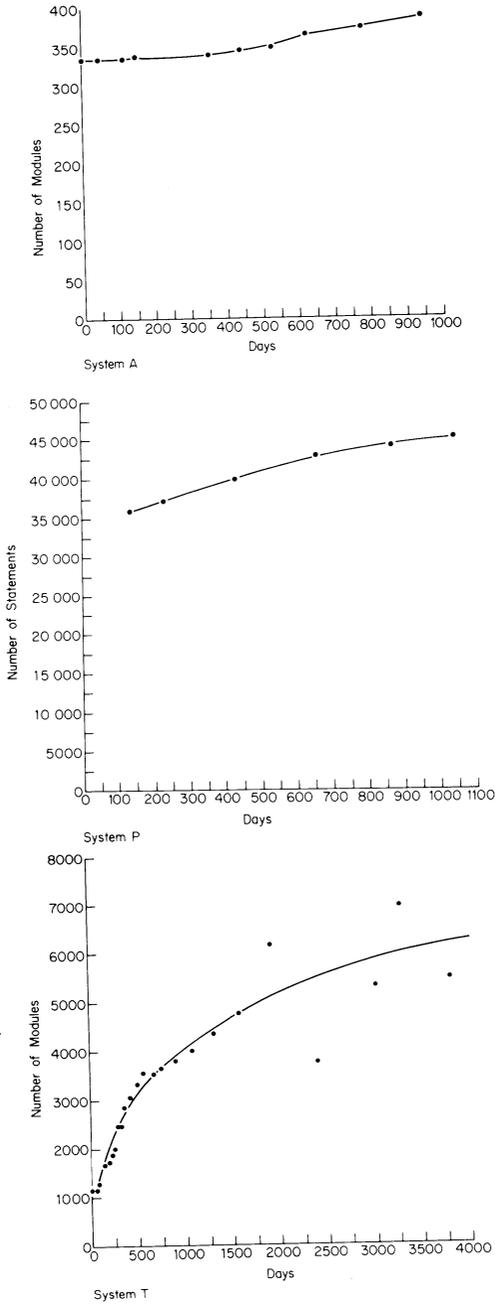


Figure 1 a, b, c System Size as a function of Age

This record of persistent growth provides further support for one of the most fundamental postulates [LEH74] of a study of software evolution: that the construction of a large software system cannot be a straightforward implementation of some preconceived design. Rather it is of necessity a process that continuously modifies and improves parts of the code to reflect changes in the environment, shifts in objectives and advances in technology.

Passing over the basic growth trends displayed by these systems one may go on to consider the *rate of growth*. Both systems P and T show a very clear falling off of the rate of growth as the system evolves. System T, for which the longest sequence of data is available, has a size which increases approximately logarithmically with age.

Several explanations for this slowing down of the growth process present themselves. Firstly it might be caused by a reduction in the level of demand for enhancements to the system. This explanation seems untenable in view of the fact that the managers of both systems P and T were adamant in asserting that there was no significant reduction in the pool of waiting requests for new capabilities. A second possibility is that the reduced rate of system growth represents a reduction in the resources made available for system development. It is known that the peak manpower and budgetary allocations for the development of system T occurred some time after the reduction in growth rate becomes apparent. Hence it was concluded that the diminishing growth rate could not be accounted for by a reduction in the resources applied. Rather the system must be becoming innately more difficult to enlarge. System P corroborates this observation by showing a diminishing rate of growth during a period of observation for which a weighted manpower index of team size fluctuated without any definite trend (Fig 2).

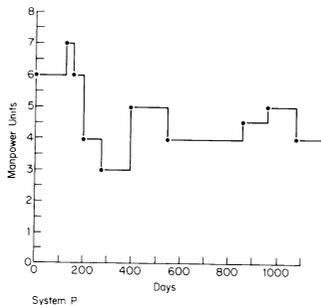


Figure 2 System P Resources

System A appears to fall outside this pattern, showing a slightly increasing growth rate over the recorded period. This is of itself rather anomalous since the system was specifically avoiding major enhancements and also had experienced a steady attrition in the size of the maintenance team. However, a satisfactorily simple interpretation of the effect is that it is a perturbation of the same type that can be clearly discerned in system T growth (Fig 1c) [BEL76]; [LEH76b] (9.6). It is known that the 'maintenance phase' of the project for which we have presented data was preceded by a 'development phase'. Although sadly no quantitative information was available for this phase, growth must have been faster in view of the size of the system at the beginning of the observational period and the indicated life time of the system. So it does appear that the net rate of growth has been slowing down in accordance with the intentions of the project's managers. We note that in the case of A and also, the rate of growth has not reduced as a consequence of the fact that the maintenance team was reduced in size by about half over the period of observation. One might even conjecture as others have done before us [BR075] that a reduction in team size actually leads to an increased growth rate, an improvement in programming effectiveness.

To summarize this section: The combined evidence extracted from systems A, P and T, support the conclusion that the products of the software projects undergo a steady expansion over their lifetime. The rate of growth falls off in a way which cannot be fully explained by a lack of demand for further enhancements or by reductions in the resources available for development. The implication is that as a software system evolves it becomes increasingly difficult to modify or to add new functions to it; its complexity, in some sense, increases.

5 The Effective Rate of Work

In the preceding section it was argued on rather qualitative grounds that fluctuations in the level of resources available for development and maintenance play only a minor role in determining the long term rate of growth of a software

9.6 (Orig) *It may be due to increasing familiarity with the system and a sense of urgency on the part of both maintenance personnel and users/customers, to attach additional capability to the system before supplier support is finally withdrawn and the system is effectively frozen.*

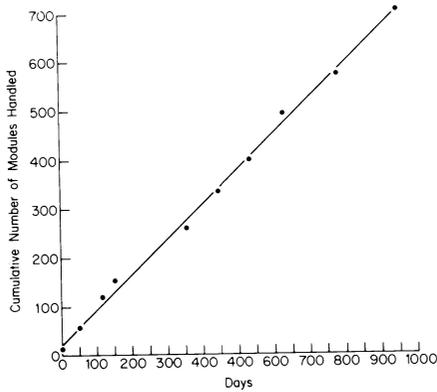
system. In this section we bolster this claim with more quantitative evidence that the effective rate of work on such systems is much more stable than a superficial inspection of changes in project team size and budget would indicate.

For systems A and T a count had been kept of the number of their elements which had been worked on during each release - we shall refer to this as 'modules handled' data. This count may be interpreted as providing a measure of the difficulty in achieving a given release. For, naively speaking, modification of say 10% of a system is likely to be a more straightforward task than coordinating changes, inevitably interacting, to say 30% of the system elements.

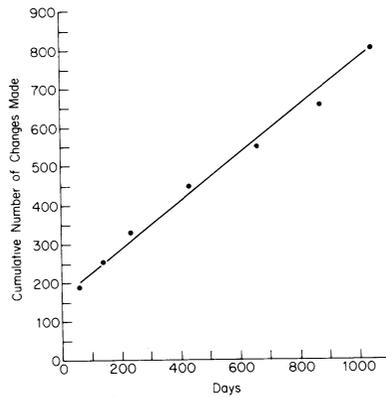
For system P, module handling data was not appropriate in view of that system's table driven design and higher level language implementation. The most comparable information available was a record of 'changes made'. Each such change corresponds to a unit enhancement of the system rather than to an amendment of a particular code element. In using such a measure one must also then assume that there has been no gross change in the programming team's understanding of what constitutes a unit change over the system lifetime.

Figures 3a, 3b, and 3c are cumulative plots of the work achieved (as represented by modules handled or changes made) on systems A, P and T respectively. The primary observation to be drawn from these graphs is that in all three cases these cumulative plots closely correspond to points lying on a straight line. In other words, the data from all three systems confirms the conjecture of the opening paragraph of this section, that the organisation is completing work at an approximately constant rate.

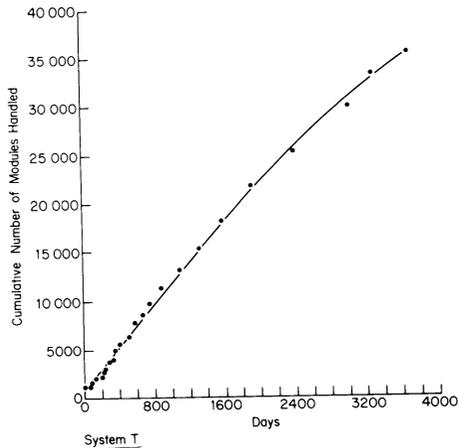
Some justification is perhaps required for considering cumulative measures of the amount of work achieved. Figures 4a, 4b, and 4c illustrate the actual work rates per day computed from the same data as Figure 3. The 'rate of making changes' information from system P shows no obvious pattern. However, the other measure of work rate - modules handled per day - shows considerable fluctuations in both systems A and T. More particularly both graphs have a zigzag shape with peaks being shortly followed by troughs. Unfortunately there is insufficient data to support this qualitative appraisal with a negative autocorrelation measure. On the available evidence, it appears that, despite this, the apparently steady work rate implied by the cumulative data conceals short term fluctuations about a well defined, average trend.



System A



System P



System T

Figure 3 (a, b, c)
Cumulative Work Achieved as a Function of Age

This is entirely consistent with the authors' basic view of the programming process as a self stabilising feedback system being constantly perturbed by management decisions affecting the level of inputs driving the system.

We must also note that to some extent the fluctuations as observed, particularly in 4c, exaggerate the differences between handle rates achieved in different releases. The rates plotted here are computed from the inter-release interval whereas they should have been based on the period of the time over which the work was executed. This release-overlap effect is of course largely removed when cumulative data is plotted.

6 The Distribution of Development Work

The two preceding sections discussed project size histories and work rates achieved respectively. In particular the cumulative count of work achieved was shown to be a linear function of system age despite release by release variations in the actual amount. In this section we relate these two sets of measurements and use them to study the distribution of development work through the system, in particular whether it is localised or widely scattered. It is intended that this should (i) clarify and justify the use of 'modules handled' records, (ii) display a relation between system size, release interval and modules handled count which refines the previous hypothesis of constant modules handled rate, and (iii) introduce a hypothesis about the distribution of development work which provides a basic framework for direct measurements of system complexity.

A measure of system complexity useful to software project management should be able to indicate when structural redesign has become essential in order to facilitate further development or to forestall ultimate unmanageability. Such a measure is expected to relate closely to the way in which maintenance and development work is scattered through the program text rather than being localised in particular sections. The justification is that the process of repeatedly modifying a system without redesign will tend to obscure its conceptual structure and introduce logically redundant entities with scattered interconnections. Consequently, logically simple alterations to the system may require physically scattered changes to the program text, making enhancement difficult, error prone and ever more likely to increase the destructing effect.

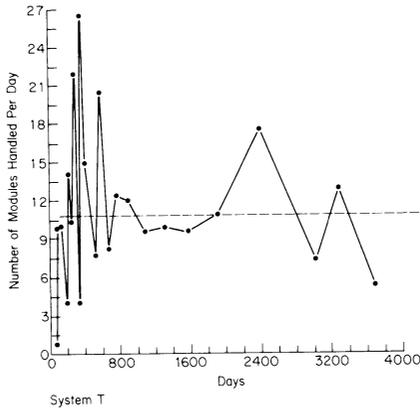
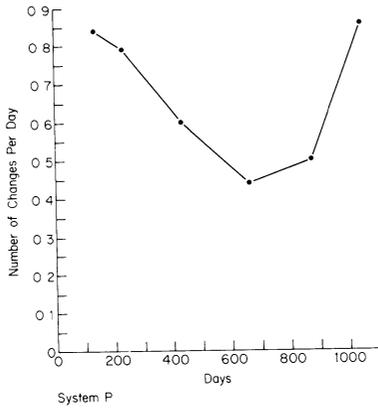
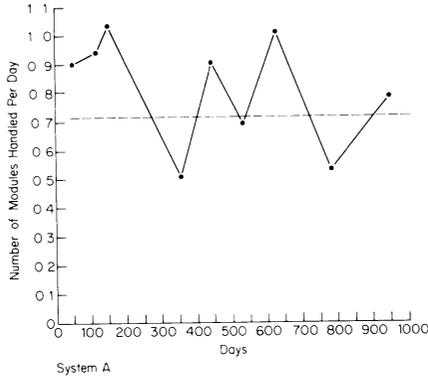


Figure 4 (a, b, c)
Work Rate as a Function of Age

The fact that system P is written in a high level language, is table driven and is only coarsely divided into components, makes it difficult to determine from available data the extent to which the work done for each release was localised. However, for systems A and T, since system size in modules and the number of modules handled are both known, a simple calculation determines the fraction of the system actively involved in each release. This is plotted in Figures 5a and 5b. Although both these graphs exhibit an upward trend, this should not be interpreted as a direct indication of increasing system complexity. It will be shown that it is more completely explained in terms of a static assumption, itself of some interest, about the scatter of maintenance and development work.

The main reason for not using 'fraction of system active' in each release as a measure of system complexity is that this statistic is rather sensitive to the length of the release interval. Now in this study we have assumed that although the rate of system growth is regulated by evolutionary effects, the actual choice of release points is made by management to conform to business requirements. Thus release dates reflect information about the state of the software being developed only indirectly, being strongly influenced by factors rather outside the scope of this report. So, without further discussion, we can observe empirically that the release intervals have lengthened in both systems A and T (and also in system P). Since the release time represents an exogenously chosen point at which the state of the system can be sampled, it can be argued that longer release intervals will involve work on a larger fraction of the system even for a program which has maintained perfectly clean structure.

In order to analyse quantitatively the effect of lengthening release intervals, we must refine our model of the system enhancement process. Our previous assumption was that the effective work achieved on systems A and T was measured by the number of modules *handled*. This is an approximation and is effective only when the fraction of the system active in each release remains small. A more accurate description is that the standard unit of work is a module *handling*. One module may receive several handlings in the course of one release (each for an essentially different reason). The postulate of constant modules *handled* rate, argued for in Section 4 can now be replaced with its refinement: the number of module *handlings* in a release is a linear function of the release interval. Finally, if one makes the simple and mathematically convenient assumption that these handlings

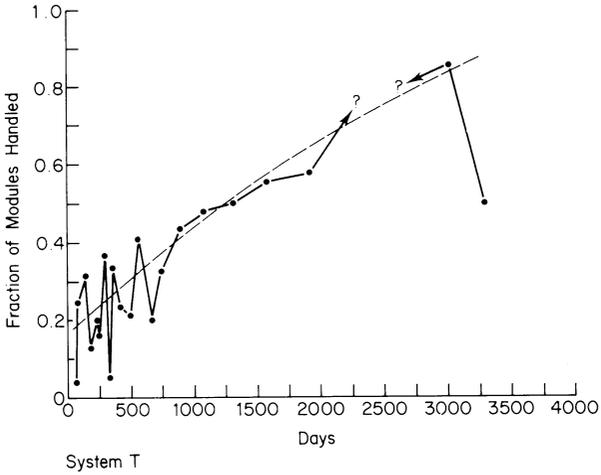
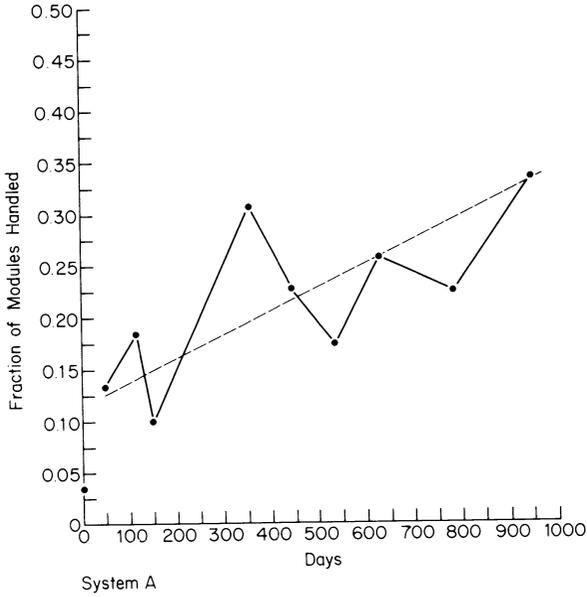


Figure 5 (a, b)
Fraction of Modules Handled as a Function of Age

are uniformly independently distributed among all the modules in the system, then it becomes possible to predict the expected fraction of the system handled in each release and hence to validate our assumptions. The actual prediction relies on the combinatorial result that if m handlings are uniformly independently distributed among n modules, the expected fraction of the modules receiving at least one handling is $1 - (1 - 1/n)^m$.

This validation procedure has been carried out for system T. In that case it is known that on average eleven modules were *handled* per day. Allowing for multiple handlings, the average handling rate must have been somewhat larger than this. Figure 6 compares the predicted fraction handled with that observed, on the basis of eleven handlings per day and using actual system sizes and release intervals. The estimate is in general low (a *handling* rate of about fifteen per day is probably more realistic), but despite this followed the actual observations quite closely. So this model is both conceptually clearer and more accurate than that following from our earlier hypothesis that the number of modules handled in a release was proportional only to the release interval.

During this validation procedure we made, on grounds of simplicity, the hypothesis that work was uniformly independently distributed throughout the system. Now in fact this corresponds to a system whose development is completely unorganised and uncontrolled. Put another way, this distribution of work would be achieved if management merely told each programmer to go away and work on the module of his choice. Now it is the mark of a well structured system that management can choose to concentrate development in certain sections of the program leaving the remainder unaltered. This would correspond to a more concentrated non-independent distribution of work around the system, eg 2/3 handlings are concentrated on 1/3 modules. In principle it should be possible to adjust the 'degree of concentration' of the distribution and the average rate of module handlings to obtain a best fit to the observed fraction of system handled. The indicated degree of concentration would then be a useful measure of the quality of system organisation.

Technically, this curve fitting procedure seems rather difficult without arbitrarily selecting particular families of concentrated work distributions. The authors havenot yet attempted to obtain a definitive best fit to the system T data. Nevertheless, it seems that the present discussions

provide one useful viewpoint of complexity and how (with increasing experience) one could hope to measure it more directly. We shall propose a further viewpoint and measure in a later section.

7 Further Comments on System T

Of the three systems considered in this paper it is system T which has been under study for the longest time and on which the richest records are available. It is therefore reasonable to examine the growth pattern of that system in considerably more detail. This has been done in refernces [BEL71b] through [BEL74].

It will suffice here to note that the most definitive feature of T, not directly observed in systems A or P, is the oscillatory nature of its size growth. In T every spurt of growth was followed by a period of retrenchment in which the system grew only slowly or, in several cases, was actually reduced in size. These recovery periods were the result either of a major redesign or of complete sub-systems being discarded as they became incompatible with further development.

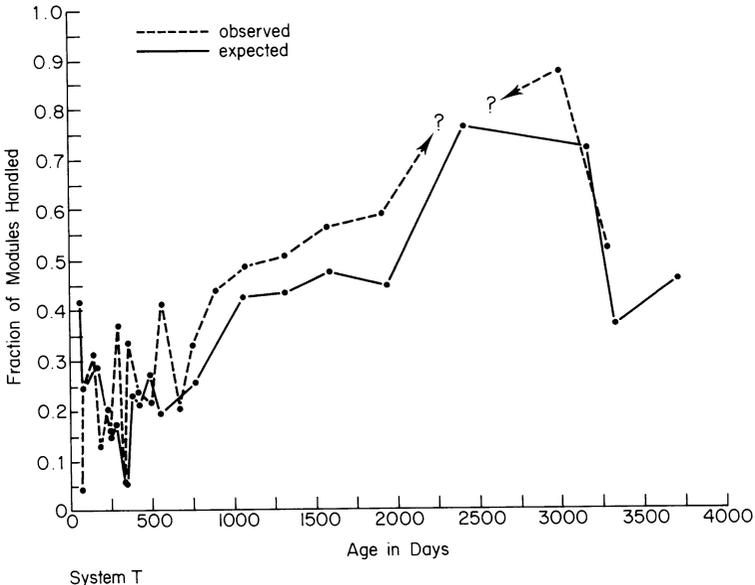


Figure 6
Expected vs Observed Handles as a Function of Age

The resultant ripple effect has been used by Riordon [RI076], following work initiated by Belady (done in 1971, reported in [BEL76]) to develop continuous system models of the growth of all three systems. Riordon's analysis shows that the observed growth patterns of all three systems are in fact compatible with a *single* continuous model. This despite the fact neither system A nor P show any of the oscillatory trends or have undergone an absolute reduction in size. In fact the different pattern of behaviour can be observed in a single non-linear model according to the values of one or other of its parameters. Thus further analysis of this type could throw considerable light on the nature of each programming process and could indicate how these might be modified to change their characteristics.

8 The Application of Evolution Dynamics Studies

In the past the implications of the observed trends patterns and invariances have been used primarily for forecasting purposes. The resultant prediction about the expected fate of specific projects or the long range trends in a system's growth have, in general, been unexpectedly accurate. Thus they have served as a verification of the basic concepts and the general conclusions drawn from the overall observations. Care must however be exercised since under certain circumstances such forecasts become self fulfilling or self negating prophecies.

Objective forecasting in the software area is mostly an unrewarding pastime nor is it very often seen as constructive. The '*Gloom and Doom*' nickname acquired by one of the authors is perhaps indicative of the forced role when one contrasts the linear extrapolations of the planners with the non-linear trends of reality.

A far more constructive role follows from the application of ever increasing insight and knowledge to the planning process. Taking into account all three factors, trends, cyclicity and invariances *plus* some specific knowledge about the particular system, past releases and work requiring to be done, one can come up with a very precise assessment of the time required to achieve the enhancement/repair normally incorporated into a single release. Similarly, the past patterns suggest, independently of business considerations, how much work *should* be incorporated into a future release, by how much one *should* plan to grow in a particular time interval if quality and delivery problems are to be avoided. From extrapolation of the fraction of modules handled trend,

as modified by other knowledge about the system and work planned, one can assess the validity of plans in relation to the amount of rework required. As a final example, we may cite the use of the data to indicate when a system restructuring is desirable; is likely to prove advantageous *in the long run*.

The above represent examples of how management can use (and has used) such data as was available in the three environments discussed. Equally our experience has suggested other parameters which if monitored over a long period could, perhaps, permit even more precise analysis of progress, potential and trends. To give just one example where we have been *unsuccessful* with presently available data, we mention development and maintenance complexity measures as determined by 'the average number of old module handlings per new module added' and 'per fault fixed' respectively. On the basis of our experience we suggest that such definitive trends are likely to exist. If such average ratios can be determined, their value over any one period, the long range trend, and deviations from it, all provide very clearly indicators and warnings of developing trouble, of local difficulties and of the successes and achievements of a programming project, group or system.

Finally, though we have not discussed this in the present paper, there is no doubt in the minds of those who have actively pursued these evolution dynamics studies, that they have yielded significant insight into the programming process, its management and planning. As these studies are pursued, they will make further significant contributions to the ability to undertake, manage and control large scale programming; to produce timely code of better quality, more efficiently and more economically. (9.6) The techniques do not and cannot replace the more direct studies of programming and of software engineering and the improvements that such studies produce. But they do compliment them and provide a yardstick whereby their effect and long range effectiveness may be measured.

9.6 (Eds) *The experience of the last six years since the above was written, and the progress made has evidenced, for example, by subsequent papers as reproduced in this book provide, we believe, evidence to support this claim.*

9 Epilogue

Research and Development time and resources devoted to the study of program evolution have been small. We believe that increased participation in this type of work, more widespread recognition of the system-like behaviour of large programs and of programming organisations can make a positive contribution to the advance of software engineering and large scale programming management.

Equally however it appears that the style of measurement and analysis here adopted can throw significant light on the nature of programs and of programming. By quantifying such concepts as size and complexity, by addressing the meaning and measurement of program power and quality and by studying the programming process, its nature and its output, we have already achieved and expect to continue to achieve a deeper understanding of communication with and the controlled use of computers; surely the central theme of Computing Science.

One last word must be added. Evolution dynamics follows one of the approaches to modern study of the physical world, by studying programming and the programming process as they are. Given this, then that follows. Particularly in view of our identification of process invariances and program size bounds, one may conclude that perhaps the man problem is that the large, general purpose software we have studied should not be allowed to exist. Maybe one should adopt quite different approaches to the provision of multiple function systems. We have not addressed this point here though we have our own judgements and viewpoints. But our presentation of what happens when such systems are created, used and maintained, may perhaps help to resolve this more fundamental question in the future, or at least cause it to be seriously considered.

10 Acknowledgements

Our grateful thanks are due to our colleagues at Imperial College and associates at IBM for many interesting discussion and observations. In particular we are happy to acknowledge the contribution of D R Hooton for his painstaking collection and analysis of the P data. We also wish to acknowledge the willing, indeed, enthusiastic, support of the A, P and T organisations and in particular of those individuals who assisted us in collecting and interpreting the data.

The Science Research Council has and is continuing to provide support to the evolution dynamics project. In particular they enabled L A Belady, who has been associated with the evolution dynamics studies from their inception to spend one half year period with us as a Senior Visiting Fellow. To him also our grateful thanks are due for his continuing collaboration and criticism. If not for the geographical separation he would surely have been a co-author of this paper. One of us (MML) must also acknowledge the contribution of his colleagues in IFIP WG 2.3 who by advancing his understanding of programming and by providing specific comments on the evolution dynamics concepts presented here have also made significant contributions to the concepts and insights presented. Finally our very grateful thanks are due to Miss J Taplin for her unending and cheerful patience and support in the preparation of the paper.

CHAPTER 10

EVOLVING PARTS AND RELATIONS - A MODEL OF SYSTEM FAMILIES*

1 Introduction

It has been recently noted that due to constant error repair and functional enhancement software systems continuously evolve [BEL76]. This evolution is characterised by change of code, addition of new modules, and sometimes restructuring of the system. Other researchers [PAR76] have studied the concept of *software families*, ie groups of similar system, evolving from the same generic program, but each member meeting different operational requirements.

Much work has been devoted to the process of constructing programs: design methodologies proposed [FRE76], supporting tools devised [DEJ73] [KER76]. Most of these efforts are mainly directed towards the construction of programs - from conception to delivery of the software product. In contrast, this paper is concerned with a more general view of the software development process, in that it focusses on the aspects of *controlling the evolution* of large software families. (Family as defined here somewhat differs from that of Parnas). A particular model of a class of software families is presented, some of the difficulties in controlling the evolution of families are described, and possible solutions offered. We also discuss methods of expressing the *relations* which exist between the parts of a system, and how these methods influence the amount of work required to maintain, enhance and install such a system. We do not deal with programs in detail, but study the problem at the level of parts: *units and modules*. This paper opens a wide research area and we hope others will be motivated to continue investigations.

2 A system of units

Let us assume that a software family is spanned by a set of units $U_B, U_1, U_2, \dots, U_n$ of instructions and data. A *configuration* is a subset of the units which includes U_B .

The unit U_B is called the *basis* of the family, ie it includes all that is common to all family members. Each unit U_i may implement some function and/or repair other units. Different configurations can be chosen to meet different (and sometimes contradictory) operational requirements. For illustration, let us assume that U_1 and U_2 are different implementations of, say, memory management and U_3 is a fix to errors in U_2 . Then $U_B U_1$ could be a configuration particularly suited for general purpose applications and $U_B U_2$ for a picture processing application, while $U_B U_2 U_3$ is similar to the latter but repaired by U_3 .

Clearly, not all combinations of units are permitted. In the previous example, a configuration having U_3 makes sense only if also U_2 is included, and similarly, U_1 and U_2 could be mutually exclusive. In general, we call combinations which are useable as real systems 'permitted configurations'.

The set of permitted configurations can be represented in canonical form as a relation over the set of units. Figure 1 shows such a relation -- each row corresponds to a permitted configuration, '1' denotes the presence of the unit in the configuration and '0' its absence. Since U_B is assumed to be included in all configurations, its column is omitted for the sake of simplicity. The relational notation allows any arbitrary set of configurations to be permitted, ie, the notation is universal in the sense that it does not constrain the expression of permitted configurations.

U_1	U_2	U_3	...	U_n	
1	0	0	.	0	
0	1	0	.	0	
0	1	1	.	0	
.	

Figure 1: Relational representation
of Permitted Configurations

There are practical examples of configurable families of systems as defined above. For example, the maintenance of OS/360 and 370 [IBM73] is effected by applying *Program Temporary Fixes (PTF)* to the system, ie, a permitted configuration includes a *system release* as basis and a subset of PTF's. In order to reduce the frequency of sometimes unnecessary changes of installed systems, the user is not

required to install all PTF's and he may choose a subset of them. Obviously, not all subsets of PTF's form permitted configurations. This is because some PTF's assume the existence of other PTF's in the configuration. In our notation a system release corresponds to U_B and each PTF to a unit U_i .

A second example is provided by the recently announced OS370/MVS *Selectable Units (SU)* [NEW76]. In this system, there is a basic MVS (a master release) to which different SU's can be added. Each SU corresponds to some functional addition or enhancement. In order to accommodate maintenance, PTF's are also included in the scheme. That is, each SU as well as each PTF corresponds to a unit U_i of our notation. As far as we know, similar schemes are being used to update other large software systems.

The number of rows in the relational representation of the permitted configurations (ie, the number of permitted configuration) can grow exponentially with the number of units. Therefore, a family of several dozens or hundreds of units (orders of magnitude for PTF's or SU's) may lead to a relation spanning many millions of rows. Thus, there are practical cases in which the canonical representation is impractical.

The size of the relation is reduced by treating each unit as a Boolean variable (ie, given a configuration S , $U_i = 0$ if U_i is *not* included in S and $U_i = 1$ if U_i is included in S) and expressing the whole relation as a Boolean function

$$f(U_1, U_2, \dots, U_n)$$

having value 'TRUE' (or '1') if and only if the values of U_1, U_2, \dots, U_n correspond to a permitted configuration. Function f could be expressed using the usual AND, OR, NOT operations and minimised by applying well established techniques for hardware [ML65].

Usually, however, AND, OR, NOT do not suit best the nature of evolutionary configurable software. In such a family there is a constant creation of new units (new function or maintenance), and when a unit is created function f should be updated to reflect the permitted configurations which include the new unit. The usual constraint of these configurations is that a new unit requires the presence of some older units and precludes the presence of others. In this case, a natural Boolean operation would be

IF U_i THEN (U_a, U_b, \dots, U_c) AND NOT (U_d, U_e, \dots, U_f)
 (a, b, c, \dots denote arbitrary indices)

which is '1' if and only if

$$\bar{U}_i + \bar{U}_a \cdot \bar{U}_b \cdot \dots \cdot \bar{U}_c \cdot \bar{U}_d \cdot \bar{U}_e \cdot \dots \cdot \bar{U}_f$$

where + is logical OR and . is logical AND.

Similarly, there may be different constructs which are advantageous in representing the relations existing in other family types.

In practice, the relations which define the collection of permitted configurations (ie, function f) is constantly used. At different computer centres, units are added to (or removed from) a configuration to meet changing functional and maintenance requirements. In typical installations PTF's are added almost every day. For each change, function f should be checked to insure that the resulting configuration is permitted. At the software factory (the development shop), new units are constantly designed and constructed to respond to detected errors and new functional requirements. In this process too, function f is constantly consulted and updated. It is therefore important that function f be expressed in a way which makes during modification work local relationships between units easy to understand. Furthermore, since f may be a very complex relation (in practical cases f may have thousands or millions of basic operations), it is important that the user understand the structure of f such that manipulations on f can be done routinely. This is achieved by representing f in a *fixed format*.

For illustrative purposes, the PTF system uses the following fixed format representation of f :

$$f = f_1 \cdot f_2 \cdot f_3 \cdot \dots \cdot f_i \cdot \dots \cdot f_n$$

in which

$$f_i = \text{IF } U_i \quad \begin{array}{l} [(\text{THEN}(\text{----}) \quad \cdot \text{NOT}(\text{----})) \quad + \\ +(\text{THEN}(\text{----}) \quad \cdot \text{NOT}(\text{----})) \quad + \\ \cdot \quad \cdot \quad \cdot \\ \cdot \quad \cdot \quad \cdot \\ \cdot \quad \cdot \quad \cdot \\ +(\text{THEN}(\text{----}) \quad \cdot \text{NOT}(\text{----})) \end{array}$$

where ---- stands for arbitrary units.

That is, each f_i states the set of all possible alternatives of required and precluded units for configurations that include U_i . This format has the advantage that, in order to check whether a configuration is permitted, it is necessary to check only the f_i 's of units participating in the configuration. This format also neatly distributes the relations among the unit, by pairwise associating each f_i and U_i (possibly as a header of the program code), thus making easy the shipment of new units, along with the corresponding update of f , to user installations.

Clearly, when a fixed format representation of f is used, the format should be chosen as to fit the nature of the program family at hand. For example, let us assume a family having many large groups of units such that at least one unit of each group is required in any permitted configuration. If such a relationship is expressed in the format used by the PTF system, then function f will be extremely complex and the basic local relationships between units will be difficult to find and grasp. On the other hand, if the format includes the construct

$$(U_a + U_b + \dots + U_c) \cdot (U_d + \dots + U_e \cdot \dots \cdot (U_f + \dots + U_g)),$$

the relationships are much more clear and concise.

The fixed format used for the PTF system is logically universal: any Boolean function f is representable. Since in evolutionary families there is a fair degree of unpredictability of the structure which the system may take by successive modifications, a logically universal format is superior, even if initially unnecessary. This will accommodate possible exception cases and avoid dramatic modification of the data base that stores the function f or the accessing programs.

3 A system of module versions

A unit may implement a fairly large and complex subsystem. Therefore, each unit is usually subdivided into several subunits called *modules*. The module is considered the basic atomic entity and it is usually of such a size that it can be fully grasped and handled by a single programmer. In practical systems the size of units in number of modules may vary greatly. For example, a PTF usually comprises a single module or just a few; a selectable unit may have a few dozens or hundreds of modules; while a base unit often spans several thousands.

There may be modules which are required to behave differently while participating in different configurations. This results in having different *versions* of the module. As an illustration, let M_a be a module of U_1 , with two versions $M_a(1)$ and $M_a(2)$. We may assume that U_1 uses $M_a(1)$ whenever U_5 or U_6 are included in the configuration, but uses $M_a(2)$ otherwise.

Some units (eg PTF's) are created for the sole purpose of updating other units. These cases create *overlapping* modules of different units. For example, let

$$U_1 = M_a(1)M_b(1)M_c(1)M_d(1)$$

and U_2 be created as a PTF to repair errors in M_b and M_c ; then

$$U_2 = M_b(2)M_c(2).$$

Thus the module versions used by configuration (10.1) (U_1U_2) will become:

$$M_a(1)M_b(2)M_c(2)M_d(1).$$

In other cases a more subtle overlap between units may exist. For example, a new unit, U_3 , could be created such that, when U_3 is running with U_1 , a change to $M_d(1)$ is required resulting in a new version:

$$U_3 = M_d(2)M_c(1)M_f(1)$$

Thus, for configuration (U_1U_3) the following module versions are to be used:

$$M_a(1)M_b(1)M_c(1)M_d(2)M_e(1)M_f(1).$$

In practice the size of overlap may vary greatly. All PTF modules also belong to other units, while Selectable Units overlap earlier modules and add new (generic) modules. Most of the units have only a single version of each of their modules. However, there are units with several versions of some of their modules, introduced by differences in configurations in which they participate. The number of these units may increase with the increasing variety of functions performed and environments supported.

10.1 (*Orig*) *Strictly speaking each configuration must contain U_B as well.*

Formally, a model of the software family can be defined as follows:

- 1 The *family* is a set of $m+1$ units:

$$f = \langle U_B U_1 U_2 \dots U_m \rangle$$

(a set is indicated by a list between $\langle \rangle$)

- 2 A subset of F containing U_B is called a *configuration*.

- 3 Each unit consists of a set of *generic modules*:

$$\text{MOD}(U_i) = \langle M_{i1} M_{i2} \dots M_{ik} \rangle$$

where $i1, i2, \dots, ik \in \{1, 2, \dots, q\}$, assuming that the family has q generic modules.

- 4 The set of generic modules of a configuration is the union of the (generic) modules of each unit in the configuration:

$$\text{MOD}(U_a U_b \dots U_c) = \text{MOD}(U_a) \cup \text{MOD}(U_b) \cup \dots \cup \text{MOD}(U_c)$$

- 5 Each unit actually has a set of *module versions* for each one of its generic modules

$$\text{VER}(U_i) = \langle \langle M_{i1}(d) M_{i1}(e) \dots M_{i1}(f) \rangle \dots \langle M_{ik}(g) \dots M_{ik}(h) \rangle \rangle$$

where $M_{i1}(d), M_{i1}(e), \dots$ are distinct versions of generic module M_{i1} (usually, most generic modules have a single version).

- 6 The totality of module versions of a configuration $U_a U_b \dots U_c$ is

$$\text{VER}(U_a U_b \dots U_c) = \langle \langle M_{u1}(p) \dots M_{u1}(r) \rangle \dots \langle M_u(s) \dots M_u(t) \rangle \rangle$$

a set in which each element is the union of those elements of $\text{VER}(U_a) \cup \text{VER}(U_b) \dots \text{VER}(U_c)$ which correspond to the same generic module. Clearly, for any set Q of units, the number of elements in $\text{MOD}(Q)$ and $\text{VER}(Q)$ is the same, and there is a one to one correspondence between them.

- 7 An *actual system* of a given configuration $\langle U_a U_b \dots U_c \rangle$ is the set of unique module versions, one from each element of $VER(U_a U_b \dots U_c)$. Again, there is a one to one correspondence between the elements of an actual systems of $MOD(.)$ of the same configuration. The *set of actual systems* (ie, the set of *all* possible actual systems for a given configuration) is denoted by

$$SYS(VER(U_a U_b \dots U_c))$$

and it is actually the cross-product of the elements of $VER(U_a U_b \dots U_c)$

- 8 If $\langle U_a U_b \dots U_c \rangle$ is a permitted configuration, then the set of *permitted actual systems* for this configuration is a subset of its set of actual systems:

$$PSYS(U_a U_b \dots U_c) \quad SYS(VER(U_a U_b \dots U_c)).$$

In practice this is a proper subset and usually much smaller than the totality of actual systems. In many cases only one actual system is permitted for each configuration.

In the next section we turn our attention to the problem of representing PSYS.

4 Permitted actual systems

4.1 Definition

The set of permitted actual systems can be described in a canonical form by a relation similiary t the relation of Figure 1. Such a description is exemplified in Figure 2. The leftmost columns stand for the units and, as in Figure 1, they describe the permitted configurations. One or more rows exist for each permitted configuration, and each row describes a unique permitted actual system as a realization of a configuration appearing in the same row. In the columns for modules, '0' means that the module is not included in the configuration while a non-zero integer identifies the corresponding module version.

In the example, there are two (optional) permitted actual systems for configuration $U_1 U_2 U_3$:

$$\langle M_1(1)M_2(2)M_3(2)M_4(1)M_5(1) \rangle$$

and

$$\langle M_1(1)M_2(2)M_3(2)M_4(1)M_5(2) \rangle.$$

$$F = \langle U_1 U_2 U_3 \rangle$$

$$\begin{aligned} \text{MOD}(U_1) &= M_1 \quad M_2 \quad M_3 \\ \text{MOD}(U_2) &= \quad M_2 \quad \quad M_4 \quad M_5 \\ \text{MOD}(U_3) &= \quad \quad M_3 \end{aligned}$$

$$\begin{aligned} \text{VER}(U_1) &= \langle \langle M_1(1) \rangle \langle M_2(1) \rangle \langle M_3(1) \rangle \rangle \\ \text{VER}(U_2) &= \quad \langle \langle M_2(2) \rangle \quad \langle M_4(1) \rangle \quad \langle M_5(1) M_5(2) \rangle \\ \text{VER}(U_3) &= \quad \quad \langle \langle M_3(2) \rangle \rangle \end{aligned}$$

	U_1	U_2	U_3	M_1	M_2	M_3	M_4	M_5	
	1	0	0	1	1	1	0	0	
	1	1	0	1	2	1	1	1	
	1	1	1	1	2	2	1	1	
	1	1	1	1	2	2	1	2	
	0	1	0	0	2	0	1	1	
	1	0	1	1	1	2	0	0	

Figure 2: Relational Description of Permitted Actual Systems

4.2 Universal Fixed Formats

The previous representation has problems of enormous dimensions, similar to that of Figure 1, but amplified. The already millions of rows for permitted configurations of practical systems are further increased by the often multiple occurrence of permitted actual systems. This multiplicity arises because of 'don't care' situations, as $M_5(1)$ or $M_5(2)$ for configuration $U_1 U_2 U_3$ of Figure 2. The relational description of permitted actual systems can be transformed into a functional form in the same way as Figure 1 can be expressed by function f . Here too, for clarity of structure, a fixed format representation is preferable. The table of Figure 2 can be functionally represented by

$$H(U_1 U_2 \dots U_n M_1 M_2 \dots M_k)$$

with $U_i = 1$ or $U_i = 0$ according to whether U_i is or is not a member of the configuration; also $M_i = j$ if version j of M_i is in the actual system and $M_i = 0$ if M_i is not in the actual system.

$H = 1$ if and only if the values of $U_1 U_2 \dots U_n$ correspond to a permitted configuration for which the values of $M_1 M_2 \dots M_k$ are a permitted actual system, otherwise $H = 0$. Function H can be decomposed into

$$H = f(U_1 U_2 \dots U_n) \cdot g(U_1 U_2 \dots U_n M_1 M_2 \dots M_k) h(U_1 U_2 \dots U_n M_1 M_2 \dots M_k)$$

in which f is the same function as discussed in Section 2, $g = 1$ if and only if the values of $M_1 M_2 \dots M_k$ correspond to an actual system for this configuration (ie, are included in the $SYS(VER)$ of this configuration), and h is 'don't care' for nonpermitted configurations of units. Usually f is given in one of the forms discussed in Section 2, and g is easy to check from the total set of modules of the configuration. Thus, we concentrate next on the problems of representing h .

The following format of h could be useful in some practical cases

$$h = h_1 \cdot h_2 \cdot \dots \cdot h_i \cdot \dots \cdot h_p$$

in which

$$h_i = \text{IF } u_i \text{ THEN } (v_1 + v_2 + \dots + v_j + \dots + v_{k_i}),$$

where u_i is a Boolean function of units U_1, U_2, \dots, U_n and v_j ($j=1, 2, \dots, k_i$) is a set of module versions collected from distinct generic modules. If the configuration makes u_i to be '1', then $h_i=1$ (or TRUE) if and only if at least one of the v_j 's is included in the actual system. In this format, the example of figure 2 will be

$$h = (\text{IF } U_3 \text{ THEN } M_3(2)) \cdot (\text{IF } U_2 \text{ THEN } M_2(2)) \cdot (\text{IF } U_1 \text{ THEN } M_5(2)) \cdot (\text{IF } U_1 \cdot U_3 \text{ THEN } M_5(1))$$

Clearly, this format is logically universal, for any relation between configurations and actual systems can be expressed. However, in a highly evolutionary family, it is expected that new units will be continually created and often new units will have new module versions which will replace old ones in overlapping areas. If, when adding a new unit, previously permitted actual systems must remain permitted, h_i 's describing modules in the area of overlap must be changed, as well as new h_i 's generated for the new configurations. For illustration, suppose that a PTF, U_4 , is added to the system in Figure 2, such that it will replace $M_5(1)$ or $M_5(2)$ of U_2 by a new version of $M_5(3)$ in all configurations in which U_4 is included. This requires:

1 The addition to h of

$$h_5 = \text{IF } U_4 \text{ THEN } M_5(3)$$

and also, in order to keep previously permitted actual systems,

2 The changing of h_3 and h_4 into

$$h_3 = \text{IF } U_1 \cdot \overline{U_4} \text{ THEN } M_5(2)$$

$$h_4 = \text{IF } U_1 \cdot U_3 \cdot \overline{U_4} \text{ THEN } M_5(1).$$

Changing parts of h can be very tedious, especially with complex families of long standing and with those having a high rate of additions of this type. The PTF system provides an example of a real life family of this type. Frequent changes of h_i 's can be avoided by a small modification to the formula of h_i as follows:

$$h_i = \text{IF } u_i \text{ NOT } (v_1 + v_2 + \dots + v_j + \dots + v_{ki}),$$

in which, if the configuration makes u_i to be '1', then $h_i = 1$ if and only if all cited v_j 's are excluded from the configurations. The v_j 's are collections of module versions precluded by configuration u_i . In this format the h function for the example of Figure 2 is:

$$h = (\text{IF } U_3 \text{ NOT } M_3(1)) \cdot (\text{IF } U_2 \text{ NOT } M_2(1)) \cdot (\text{IF } U_1 \text{ NOT } M_5(1))$$

and the addition of PTF U_4 , described above will only require the addition of

$$h_5 = \text{IF } U_4 \text{ NOT } (M_5(1) + M_5(2)).$$

4.3 Non-Universal, More Efficient Constructs

Clearly, families with different structures or with different 'idiosyncrasies' of evolution may have different best formats of function h. We describe below two additional constructs which are useful in many practical situations. These constructs are not logically universal by themselves, but they can be used either in conjunction with one of the formats previously described or in families having certain restrictions in permitted actual systems. If in all configurations having units U_i and U_j and in which the module versions of U_j in the overlapping area of U_i and U_j are

excluded from the actual system, the previous format yields the following:

$$h_i = \text{IF } U_i \text{ NOT } v_q$$

where v_q is an enumeration of all module versions of U_j in the overlap. This situation appears very frequently in real systems (eg, the PTF system), and wherever the overlap is very large (in some practical cases there may be hundreds of modules). Thus it is convenient to express this situation by the construct below which is shorthand for the previous expression:

$$U_i \text{ PREVAIL-ON } U_j$$

or, if U_i prevails on more than one unit, by

$$U_i \text{ PREVAIL-ON } (U_{j1} U_{j2} \dots U_{jm}).$$

Using this construct, the example of figure 3 will be

$$h = (U_3 \text{ PREVAIL-ON } U_1) \cdot (U_2 \text{ PREVAIL-ON } U_1) \cdot (\text{IF } U_1 \text{ NOT } M_5(1)) \cdot (\text{IF } U_1 \cdot U_3 \text{ NOT } M_5(2)).$$

As shown by this example, the PREVAIL-ON construct does not differentiate between choices of module versions supplied by the same unit. Nevertheless, this construct is useful in many practical situations.

The following construct is based on PREVAIL-ON, but the relation is implicitly given by a naming scheme. Suppose first that a family is evolving in such a way that units are added completely ordered and that the module versions of a newer unit always prevails on those of old ones. In other words, for all i, j

$$i > j \rightarrow U_i \text{ PREVAIL-ON } U_j. \quad (\rightarrow \text{ meaning 'implies'})$$

In such a family it is not necessary to state explicitly the relation, since this is implied by the ordered names (ie the indices) of the units. This can be extended to the case in which groups of units are added simultaneously, and the units of new groups PREVAIL-ON those of older groups, but inside the group the units are unrelated by the scheme (or they can be explicitly related by one of the methods explained earlier). In this case we could use a two-dimensional indexing scheme as follows:

$(1, a_1)$	$(2, a_1)$	\dots	(q, a_1)
$(1, a_2)$	$(2, a_2)$	\dots	(q, a_2)
\vdots	\vdots	\vdots	\vdots
\vdots	\vdots	(i, a_r)	\vdots
$(1, a_{k1})$	$(2, a_{k2})$	\dots	(q, a_{kq})

in which for all i, j, a_s , and a_r

$$i > j \rightarrow U(i, a_s) \text{ PREVAIL-ON } U(j, a_r)$$

In other words, the first index gives the ordering, while the second has no ordering implications, it just provides a distinguishing label. This scheme can be further generalised to the case in which each group there are ordered subgroups. This will be done by a three dimensional indexing scheme

$(1, a_2, 1)$	$(2, a_1, 1)$	\dots	$(q, a_1, 1)$
$(1, a_2, 1)$	$(2, a_1, 1)$	\dots	$(q, a_1, 2)$
$(1, a_1, 3)$	\vdots	\dots	\vdots
\vdots	\vdots	\dots	\vdots
$(1, a_1, p_{11})$	$(2, a_1, p_{12})$	\dots	(q, a_1, p_{1q})
$(1, a_2, 1)$	\vdots	\dots	\vdots
$(1, a_2, 2)$	\vdots	\dots	\vdots
\vdots	\vdots	\dots	\vdots
$(1, a_2, p_{21})$	\vdots	\dots	\vdots
\vdots	\vdots	\dots	\vdots
\vdots	\vdots	\dots	\vdots
$(1, a_{k1}, p_{k1})$	$(2, a_{k2}, p_{k2})$	\dots	(q, a_{kq}, p_{kq})

In this case the first and third indices are used to imply ordering, while the second is used as an arbitrary label which only implies grouping. For all i, j, l, m, a_s and a_r

$$i > j \rightarrow U(i, a_s, l) \text{ PREVAIL-ON } U(j, a_r, m)$$

and for all i, j, l, a_s ,

$$i > j \rightarrow U(l, a_s, i) \text{ PREVAIL-ON } U(l, a_s, j)$$

In the same way, this method can be generalised to a labelling scheme of arbitrary dimension. It can be shown that ordering and nonordering indices strictly alternate.

Then, the structure of a label will be

$$(i_1, a_{j_1}, i_2, a_{j_2}, \dots, i_k)$$

where the i 's describe the grouping of units at the same level into subgroups. In general, these schemes create a nested partial ordering over the units. In practical cases of this nature, the above implicit method will give the relations in a clear form, and may save hundreds of explicit relations. Many practical systems are decomposable by a nested partial order; for example, the combined SU and PTF system can be viewed as a two dimensional scheme of this type. Other systems could be 'nearly-decomposable' [SIM69] into a nested partial order. In this case, the implicit method could be used in conjunction with one of the formats described earlier, or with any other representation of the exceptions, whichever is a most efficient for a particular situation.

5 Discussion

In the configurable family model there are two levels of relations between entities. The first level (given by function f of Section 2) defines the set of permitted configurations of units and the second level (function H of Section 4) spans the set of actual modules (ie the module versions) which implement such a configuration. Program parts - code and data forming the units - become useful only if meaningfully interconnected by relations. Furthermore, at the system level, there is a *duality* between the concept of evolving program parts and the concept of their relations. Whenever an addition, a repair or any other modification is made to the program, a corresponding change must be made on the relations. As illustrated by the examples given in this paper, for large and highly evolutionary families the relations as well as the manipulation of these relations could be of significant complexity. Indeed, we hope to have shown that the predominant and most challenging design problems presented by an evolving large system are those of *maintaining the internal consistency of its numerous versions*. It, therefore, pays to give special attention to understandability and conciseness.

The relations are regularly used at many locations. Installations at each distinct site should check the relations before a new system configuration is loaded. Questions like 'is this a permitted configuration?' or 'if we want certain function (ie unit) what other units are

required?; which alternatives do we have?', etc, should be answered. Therefore, easy ways to access the relations should be provided. For example, in the PTF system, relations could be distributed as headers to PTFs and at each installation a program called SMP (System Modification Program) [IBM73], use the relations for applying the fixes.

In a broad sense the relations, together with the program utilities using them, act as a System Generator (SYSGEN). In a traditional SYSGEN program, relations are imbedded in the program. In contrast, the above scheme offers a clean separation of relations and the program that uses them. This is a noteworthy advantage, especially in situations of frequent and continuous change.

Relations are also used in the development shop while manipulating (designing, testing, maintaining) the family, and they are updated whenever a change is made. This manipulation is presumably done by a team of programmers, where communication and synchronisation of human activities are required. As already stated, there may be hundreds of units, thousands of configurations and many thousands of module versions. All these point to the need for an integrated (computerised) facility, similar to the program development aids [HAN76] already proposed, but also including the evolutionary aspects of configurable families.

As already noted, new units and modules are constantly created to respond to maintenance and further functional needs. This variety explosion can be reduced, in part, by two types of counteractions. Units which are seldom used, or which have a proper replacement, can be discontinued, and compatible units can be merged together (this is called 'coagulation'). However, there is certain amount of work (and cost) associated with both of these operations. Coagulations should be performed and relations properly updated in the central software factory. Unfortunately, installations using only parts of the coagulating units, or using discontinued units, perturb this scheme (for example, an installation which keeps a configuration even if it is declared not permitted). The feasibility of performing coagulations, or of discontinuing units, and how often, or what units, are all open questions for further exploration.

In this paper we studied mainly the problems of relations between entities of a configurable software family. Further problems not discussed here are the functional and structural aspects of the actual code (or programs), ways of minimising

dependencies between units, and ways of defining 'good' interfaces between entities, etc, in the context of configurable and evolutionary software.

CHAPTER 11

HUMAN THOUGHT AND ACTION AS AN INGREDIENT OF SYSTEM BEHAVIOUR* (11.1)

1 Systems Science

Significant sections of the scientific community are increasingly becoming involved in Systems Science, though they often do not recognise this subject as an independent discipline. Biologists, computer scientists, economists, engineers, mathematicians, sociologists and many others are realising that in many instances one cannot hope to master the systems studied by adopting the classical approaches to science.

The latter are largely based on a bottom-up approach. Individual elements are first studied and mastered. When these are at least partially understood, one proceeds to examine the properties and behaviour of more complex assemblies of these elements. Conceptually at least the methodology may be indefinitely extended to study ever larger and more complex systems as built up in more or less structured fashion from sub-systems and primitive elements.

In practice, however, the mathematical and other descriptive tools used for such studies rapidly become unwieldy. They tend to break down unless a system is effectively homogeneous, as in physics for example, or has a simple, regular structure, as in crystallography. In general, the properties of a system observed as an entity are not readily discernible, nor do they follow easily from one's knowledge and understanding of the attributes and behaviour of its components. Despite the fact that one may be concerned with global characteristics, these cannot be directly or totally related to or inferred from elemental behavioural patterns.

11.1 (Eds) *The Encyclopaedia of Ignorance was intended as a guide to unanswered, but possibly answerable questions. Invited contributions were asked to address a problem area of their own choice and to outline a problem to which they did not know the answer.*

Thus progress in revealing the nature of the physical world has necessitated the development of newer methodologies that do not rely on the study of individual phenomena in exquisite detail. A system, a process, a phenomenon may be viewed in the first place from the outside, observing, clarifying, measuring, modelling identifiable attributes, patterns and trends. From such activities one obtains increasing knowledge and understanding based on the behaviour of both the system and its sub-systems, the process and its sub-processes. Following through developing insight in structured fashion, this top-down, outside-in approach leads, in due course, to an understanding of, and an ability to control, the individual phenomena but in the context of their total environments.

The global, systems, viewpoint has been fruitfully applied, directly or indirectly, in many areas of the natural sciences. Epidemiology, Genetics, Thermodynamics, and Information Theory are typical areas of applied science where the approach has yielded important results of both theoretical and of practical significance.

In recent years, however, interest has had to increasingly focus on, in Simon's words, [SIM69] the 'sciences of the artificial', on the behaviour of systems created by man. And people invariably constitute sub-systems or elements of the artificial systems, if only by virtue of the fact that they design, build and use them.

Naturally one has sought to develop a theory of and for these systems in terms of the concepts, the techniques, the languages, the mathematical tools of the epidemiologist, the geneticist, the thermodynamicist, or the information theorist, for example. Further, after abstractions that remove dependencies on the specifics of the original system studied, one could expect to deduce observations or hypotheses about systems in general. Thus a discipline of Systems Science is developing and with it the systems scientist. His ultimate objective will be to isolate and reveal those attributes of system behaviour that arise by virtue of their being a system or some specific system.

2 Systems

But what are systems? Many alternative definitions have been proposed [ALE74]. For our purpose a system may be viewed as a structure of interacting, intercommunicating components that, as a group, act or operate individually and jointly to

achieve a common goal through the concerted activities of the individual parts. We note that, in general, the partitioning of a system into components is by no means unique. Selected components may be viewed as elemental or atomic but under further scrutiny each will, in general, be seen to be itself structured and to contain components that together satisfy the definition of a system in their own right. The component will constitute a sub-system of some other system or systems; of its environment.

The Concorde is clearly a system in the sense of the above definition. So are the total grouping of people and equipment that co-operatively ensure the safe transportation of the craft and its passengers, or of successive loads of passengers, between points of departure and destinations. A computer installation with its equipment and its staff represents another class of systems; one in which the perceived attributes and characteristics will be as much dependent on the attitudes, skills and managerial decisions of the staff as on the technical capabilities of the equipment. Equally each of its sub-systems, the central computer for example, is a system in its own right. Its sub-systems, in turn, include storage, arithmetic, control and peripheral access units.

3 System Measurement and System Models

Attempts to measure and express the performance of computer installations in general terms have not been very successful so far. Any measures must not refer directly to detailed workload or machine characteristics. The sheer variety and quantity of available data would quickly make the data and its analysis unmanageable. Moreover, measures are often desired to facilitate the comparison of alternative machines or workload environments, or the installation at different times and when some components will have inevitably changed. Inclusion of detailed characteristics of the installations and/or environments to be compared in the measures would, in practice, invalidate the measure. Differences in measure will in fact often be due to differences in characteristics and/or to changes in the system that cannot be uniquely or meaningfully quantised or ordered.

Thus we require to define system measures that are global in nature. For example, we might determine the average turn around time (interval between submission of a job request and receipt by the requestor of the result of its execution) for all jobs or for jobs of a particular class or type. This

measure would depend both on machine capability and on the performance of installation personnel. Given some further measures that relate system performance to job stream and environmental characteristics, one might then produce a model of the installation of system behaviour at some point in time. And given several such models one might arrive at a generalised installation model and theory.

4 The Emergence of a Problem

There is, however at least one snag to the meeting of this objective. Both the environment that generates the workload and the installation that executes it include people. Their behaviour pattern (in relation to each other and the system) depends on observation, interpretation of system behaviour, and also on less tangible factors. The more complete their understanding of apparent system behaviour becomes the more will they modify the system or adjust their behaviour in relating with and to the system, so as to obtain the closest possible approximation to what is considered optimum performance.

In general, as knowledge and understanding of an artificial, man changeable, system increases we attempt individually and collectively to modify the behaviour of that system. We attempt to make it behave in some other, preferred, fashion. The modification is obtained by changes to the system, to usage protocol and so on. The resultant configuration is and must be treated as a different system which requires a new model to represent it. Thus artificial systems and their models appear to be essentially transitory, *continuously evolving*. Does this constitute an intrinsic barrier to complete understanding and mastery of the system?

5 First Attempted Solution and its Critique

One might of course argue that in order to study the system all change proposed in response to observed behaviour be inhibited. In a system model this would be represented as an opening up of the representation of the feedback path that permits humans to modify the system in response to observing or experiencing its behaviour. However, if this is done, one is now studying a different system. And this is not just a superficial difference. This same feedback loop underlies both system improvement and its adjustment to an ever-changing environment. And adaptability, if it exists, is perhaps one of the most important properties of a system.

System *adaptation* and evolution reflects and results from the mutual reinforcement of the system and its environment. In non-biological systems human *observation, thought, action* and *reaction* play a major role in forcing and guiding system development and evolution. Thus, for example, a computer installation in which procedures, the components or the structure may not be modified in response to experience is an essential part of managerial responsibility. An installation in which all change is inhibited is in fact a dead or dying installation. (11.2)

6 A Further Example

A further example may make the basic dilemma even clearer. One of the biggest problems occupying the attention of computer scientists is that of programming methodology. The majority of those studying it, investigate the primary problem of how programs should be created in the first place [DIJ72b], [TUR75], [WIR71], [PAR72]. Others note the fact that programs undergo a continuous process of maintenance and change [BEL71b], [LEH74] driven by the evolutionary pressures mentioned above. The dynamics of the resultant program evolution can and are being studied [BEL76] for a variety of environments and programs.

In these studies there have been a number of instances where detailed numeric data has been available relating to the progress of a programming project and the growth of the subject program system [BEL76], [H0075]. In each case a similar pattern of evolution has been observed, and interpreted as a reflection of program evolution dynamics [LEH76c]. This has permitted the construction of conceptual and statistical models that reflect increased understanding of the characteristics of program development and maintenance and of the process whereby this is achieved. Consequently, it has proved possible to develop techniques based on these models for more precise planning and prediction of the programming process in relation to a specific system.

In parallel with this development the insight gained from these global, system-like studies of the programming process has permitted the development of proposals for improvement of programming methodology in general. But implementation of these proposals immediately invalidates the models of system

11.2 (Eds) *This is, of course, the continuing theme of this book, but see particularly Chapters 2 and 4.*

behaviour since the system, the process, has changed, and since it has changed in response to deductions that were made from these very models. Similarly application of the forecasting and planning data derived from a set of models to modify the output of the system invalidates these same models as representations of the programming organisation, its tools and its activities that together constitute the system. If the output of the models are accepted as essentially correct, activities are reorientated and adjusted to conform to model-based forecasts. The outputs from the models become a *self-fulfilling prophecy* of the behaviour of a new system. The latter includes the old system together with some new elements, namely the models, the model implementors, its interpreters and so on. But paradoxically the models are based on data generated from a system that excludes the models.

At the other extreme, the output of the models may incite activity to prove them wrong. This also leads to change in the system and negation of the models. In practice the stratagem adopted falls between these two extreme responses. But the consequence is the same. The mere act of studying the system leads to a change in the system which paradoxically invalidates the earlier models. (11.3)

7 Introduction to the Area of Ignorance

One may, of course, create a new model of the new system, a model that includes the system model as an element of itself. But, at best, this can only be achieved by applying an iterative procedure. Will such a procedure converge? Can it? Must it? Therein lies the problem, a problem that will force an ultimate admission of ignorance. And we shall see that this ignorance is not just due to insufficient knowledge, understanding or wisdom. We have here an area of

11.3 (Eds) *Occasionally one also observes the opposite effect, where belief and application of the model reinforces it. This is likely to cause stagnation, act as a block to progress, as the models appear, more and more, to be cast in concrete, to represent a 'law of nature'. A good example of this is to be found in cost models for software projects. If unintelligently applied they can become a total block to productivity growth by setting false, unambitious targets, or by concentrating effort on the improvement of factors that, with a new technology or environment are really unimportant or even irrelevant.*

uncertainty and indeterminacy that has its roots in the freedom of thought, of interpretation, of choice and of action of mankind, individually and collectively. As such it appears to be absolute and unbreachable.

8 The Relevance of Goedel's Theorem

An initial attempt to resolve the issue of convergence can be based on Goedel's theorem. In simple terms the latter states that one cannot prove the consistency and completeness of an axiomatic system using only the axioms and the rules of inference of that system. Informally one can state that an assertion about a system (and a model of a system represents an assertion about the system) cannot be shown to be absolutely true from within the system; by using only known facts about the system.

Suppose now that we assume that there exists an absolute theory for some artificial system. The latter could be represented by an axiomatic model in which each part and activity of the system reflects either an axiom or a theorem. From Goedel's theorem it follows that the correctness of the model cannot be demonstrated from within itself. That is, the behaviour of the system cannot be predicted absolutely from within the system. But demonstration of the validity of the model, of comparing the predictions of the model with the behaviour of the system, is an essential ingredient of artificial-system behaviour. By its very nature such activity is a part of the behaviour of the system as represented in the model. Hence there does not, in this respect exist an 'outside' to the model. Thus one cannot obtain an absolute theory, a demonstrably correct model. Attempts to include the model in itself must lead to non-convergence. Ignorance about the total behaviour of artificial systems is intrinsic to their very existence.

9 Uncertainty

In the above discussion we have gone one step beyond Goedel. We have asserted that the very activity of proving its model correct is itself part of artificial-system activity. Thus there exists no environment external to the system from which its total and absolute behaviour may be observed. Hence exact system science is not knowable, is meaningless, does not exist.

Over and above this theoretical limit there exists an even more significant barrier to total knowledge of the state and

behaviour of an artificial system. There is an essential uncertainty which at all times implies some degree of ignorance about the system. The situation can be formulated in an Uncertainty Principle, analogous to that of Quantum Physics. By measuring and modelling an artificial system we increase the extent and precision of our knowledge and understanding of its mode of operation. In general this causes the system, the environment and/or the interactions between them to be changed. Thus the more accurately we measure the artificial system the less we know about its future states, if we believe, as I believe, that in the final analysis the humans or groups of humans that change the system are unpredictable.

The parallel is of course not complete. Heisenberg's principle arises from the fact that the mere act of observation must affect the position and momentum of the object being observed. Measurement, knowledge and change of state come together, they are inextricably bound. In our case, on the other hand, knowledge appears to come first. The system change occurs subsequently, after the information gained has been digested and applied to 'improve' the system.

But the difference is a delusion. People are components of the system. Complex artificial systems cannot be designed absolutely correctly *ab initio* (11.4). Observation and modification are, in practice, intrinsic ingredients of system operation. Therefore the very act of observation that gives the humans in the system additional information about the system changes the state of the system, ultimately causes the system to be changed.

In the uncertainties of the exact sciences we observe paired indeterminacies. Thus conjugate parameters such as, for example, the position and momentum of a particle cannot, by the very nature of things, be simultaneously determined with absolute precision. But the product of the indeterminacies can be bounded precisely. We know just how close to exact knowledge we can get.

In the case of the artificial system on the other hand, one cannot (at the present time) predict with precision, what will be observed, how it will be interpreted, what action

11.4 (Eds) *In fact, in most instances the concept of correctness, is not strictly defined, the concern will be with the behaviour of the system in the 'real world' and the satisfaction that it gives. See Chs 2, 18 - 22.*

will be taken as a result. One cannot even determine the state of the total system with any certainty by applying probabilistic judgements as in Game Theory [NEU53]. After all each situation, each sequence of events, will occur only once. Thus for our case the uncertainty is of a different order of magnitude. To talk of absolutely conjugate variables as measures is not meaningful. One cannot even identify with absolute certainty, the totality of system measures that are involved in any given measurement activity. Far less can one determine an absolute and general limit to the accuracy with which even the known measures can be observed.

Are these uncertainties absolute? I think 'yes', but I do not *know*.

10 Continuous Evolution

It has already been observed that a system can in general be considered as a sub-system of some other system. Amongst its several implications the discussion of the last section suggests that every artificial system with its environments forms such a family of super-systems in which the system boundary cannot be clearly defined or confined.

Observation of system state, behaviour and performance by humans in the environment leads to changes in the system, in environmental-system interactions or in system usage. All of these represent changes in the system itself, changes that result from the unpredictable interpretation of the human observer (11.5)

In fact as humans in the system and its environment observe system behaviour they adjust their own responses, behaviour and objectives to optimise utility in some sense. This represents a change in system state. It also leads inevitably to imposed changes in system mechanisms, implemented in order to achieve more cost-effective operation in the changed human environment. These changes also, provide new opportunities for interface and environmental changes, that, for example, modify system behaviour and/or

11.5 (*Orig*) *Note the intrinsic circularity here (as elsewhere in the paper). Until an exact model is known, observed behaviour must be interpreted and the system is strictly unpredictable. But if the system is unpredictable, no exact model can exist. Hence no exact model can be found by the Scientific Method.*

usage patterns. These in turn suggest and ultimately cause renewed system state changes and so on. Thus we observe the 'mutual reinforcement of system and environment', previously referred to. It is the major cause of continuous system evolution.

It may be true that there exists an optimum strategy for system improvement, as we find with the minimax strategy of von Neumann and Morgenstern [NEU53]. But such a strategy merely maximises the minimum benefit. It can in no way lead to an exact system theory, totally predictable behaviour, a foreseeable evolutionary path to a final state.

11 The Role of Human Thought and Action

Thus we arrive at a recognition of the fundamental factor that appears to make an inexact science of the science of systems, created, used and controlled by men. The evolution of such systems depends on people's observations, thoughts, interpretations and actions. Man's ingenuity to invent new rules, new twists, new interpretations, new objectives, is as unbounded as the language in which he expresses his thoughts. [KOE71]. Total knowledge, the final state, can never be reached. Ignorance must always be present.

This observation penetrates the core of the problems and exposes an ultimate area of ignorance. If mankind's ability and expression of ability were bounded, then perhaps there could be an exact Science of the Artificial, despite some difficult questions that remain unanswered and are perhaps unanswerable. But if, as I believe (but cannot know), such a bound does not exist [LEH74], then the theory of man-made systems, of systems including people or operating in an environment that involves people, can never be complete or precise. How good can it get? That too I do not know.

CHAPTER 12

LAWS OF PROGRAM EVOLUTION - RULES AND TOOLS FOR PROGRAMMING MANAGEMENT*

1 Program Evolution

1.1 Process and product dynamics

A basic assumption underlies current *practice* in software engineering; the design, development, maintenance and enhancement of large computer programs and the management of the groups of humans that undertake these activities. Though seldom stated explicitly, planning and control techniques implicitly assume that decisions, plans, project standards and so on may be based on an assessment or on measures of *current* project, product and environmental states. It is assumed that, as for most other engineering activity, the direction and rate of progress are a direct consequence of management decisions in the light of *current* environmental conditions.

Recent work ([BEL76], [BEL77], [LEH77d], [LEH77e]) has, however, shown that this is not, in general true for large software projects, except possibly in their early stages when the groundwork is being laid and project and product characteristics determined. Once under way the overall progress of such projects, whilst still subject to management decision, is increasingly shaped by intrinsic project factors that become ever more firm and dominant as the system ages. Managers appear free to decide specific, local, issues in ways which may in the short term guide system development in the desired direction. But globally the pattern of progress of the project tends to return to an historic trend whose nature is largely determined by product and organisational characteristics. The parameters of this trend are normally not controlled and, in the current state of knowledge and understanding, often not controllable. Thus for maximum cost-effectiveness, management consideration and judgement should include the entire history of the project with the current state having the strongest, but not exclusive, influence.

We may, indeed must, regard the organisation developing and maintaining a large program as a system in the system theoretic sense [SUT75]. Its subsystems include both the organisation of personnel comprising the project groups and the program with its accumulated documentation. The former is subject in its behaviour to human decision taking and can be modelled only stochastically. The latter represents a rapidly growing inertial mass and smoothing influence. The combined system is itself a subsystem of the organisation that provides the pressures and directives controlling program evolution and use, and the resources that make evolutionary maintenance possible.

Observation has shown that the system behaves as a self-stabilizing feedback system. The practical implications of this will be discussed later. At this point we merely assert that being such a system implies the existence of a system dynamics. Indeed it is the reality of such a dynamics applicable to a wide class of project that constitutes our most fundamental observation. It is its discovery through measurement that has formed the subject matter of a previous publication [RIO76]. It is its formalisation in a series of laws that we consider here.

1.2 Largeness

The laws to be discussed have emerged from a quantitative and phenomenological study of the life history of a number of *large* computer programs. Thus formulation of the laws has had to be in terms of a property of *largeness*.

This property is not intended to reflect the numbers of instructions or modules comprising the program. Nor do we refer to the quantity of documentation required or to the program's resource requirements during execution. We do not even intend to emphasise the wealth of function contained within it. There is always a level of description at which the function is recognised as an entity, a payroll program, an operating system.

All the above mentioned measures of program size can be expected to increase as a program grows larger in the sense to be defined. But the defining property of 'largeness', the root cause if the characteristics that large programs possess and from which large programming projects suffer [BEL78], is the attribute of *variety*. A program is large if its code makes provision for such varied, manifold, conditions that the execution sequence may adapt itself to the actual

condition of its variable operational environment whilst running; ie, the requested output and the environmental condition, both in the execution system and in the user environment.

A program is large if it reflects within itself a variety of human interest and activities. And if it does it is likely to lie outside individual intellectual grasp. It will require an organised group of people to design, implement, maintain and enhance its many parts. It is the *connections* between the various parts of a program reflecting the *communication* between the variety of activities initiated and controlled by the program in execution, the communication within the implementing organisation, the communication between the implementors and their product and finally the communication between all these and the operational environment that lead to the emergence of the 'largeness' characteristics.

The preceding discussion has related the attribute of largeness to intangibles. We cannot measure or even define the amount of variety in a system or the extent to which the system lies within the intellectual grasp of a single individual; or of a small group of individuals who are in total harmony with one another. We cannot even measure the extent to which a group or groups of people working jointly on a program, a system or a system of programs, communicate, or how adequately they communicate. To make the laws meaningful, an operational definition must be devised that delimits the range of systems to a class for which they can be shown to apply.

The working definition currently adopted defines a program as 'large' if it requires, or is given, an organisation of at least two levels of management for its development or maintenance. When such a project organisation exists, the phenomena associated with largeness will appear. A management aware of the trend may however, be able to mitigate against it, for example by adequate definition, specification and control of the objectives, activities and projects of individual groups and the interfaces between groups. If only one level of management operates, appearance of the phenomena may perhaps be avoided altogether. But that is another story. Nor do we know whether and to what extent a dynamics and laws exist for programs outside this class, that is for programs that are not large in the sense of our definition.

2 The First Law

2.1 Statement - law of continuing change

A large program that is used undergoes continuing change or becomes progressively less useful. The change process continues until it is judged more cost-effective to replace the system with a recreated version.

2.2 Explanation

Any program is a model of some part or aspect of the real world or of activity within it. The real world changes continuously. A program that is to remain useful must therefore be adapted to the environment it is intended to model and serve. It must be continuously changed to keep pace. The changes may be repairing faults, providing new facilities or usage modes, supporting new application areas or exploiting more cost-effective hardware. But be changed it must or it will decay into obsolescence.

2.3 Consequences

The unending sequence of enhancement that software systems undergo is thus not necessarily or primarily due to short sightedness or lack of planning. It is intrinsic to the very being of a program. Therefore the need for change must be taken into account at all stages of specification, design and development. *Changeability is, in the long run,* as crucial a factor in the total cost-effectiveness of a program as are attributes such as its resource usage, speed or freedom from faults.

2.4 Rules, tools and technological implications

The relationship of program *structure* to the verifiability of programs and the attainment of correct programs is, by now, widely understood. Structure is equally crucial for changeability, and for the same reason. It makes a program understandable by breaking it and the processes it controls into meaningful parts and in a meaningful pattern. Each partition as an entity and the pattern as a whole must be comprehensible by an individual [DIJ74]. If this is achieved, the corresponding program text *ceases to constitute a large program* since the individual may understand the whole in a progressive manner, and understanding the whole program and its parts is a prerequisite for successful change.

But structure, whilst necessary for changeability, is not sufficient. 'Structure' in the sense used here relates to the topological relationships - the general connectivity - between substructures. It does not define the precise nature of the link. From the point of view of changeability it is essential that the links, as the *interfaces* between subsystems, (the pieces of the program), be completely defined and their properties fully specified. Equally the subsystems, as *functional components* of the program, must also be fully specified. It is not, in general, sufficient to rely on the fact that the code of each portion of the program completely determines its effect during execution subject only to the value of parameters at its defined interfaces. Each component must be explicitly specified to express its transfer function, the intent of the designer and implementor and the need of the application.

If a complete and correct specification of all parts of the system is not available, it becomes impossible to foresee fully the effect of any local change on other parts of the program and therefore on the detailed behaviour of the program in execution. It thus becomes difficult and expensive, if not impossible, to change the program without side effects that modify its behaviour in unplanned fashion.

To be useful specifications must however also be *accessible*. The purpose of a complete specification is to permit one to answer in a cost-effective manner any relevant question about the object specified. Thus to achieve general changeability of a program as a whole one requires a *structured specification* that permits ready location of all parts of a program requiring modification so as to be able to implement a required change without any side effects.

A true and complete method for developing and documenting structured specifications does not yet exist. But the techniques of successive refinement [WIR71] represent a significant step towards its evolution. Any such methodology will provide an *implicit* record of the internal connections in a program. Interactions between two parts of a program can arise only if there exists an *explicit* program object, data, table, control variable, label, queue, and so on, that acts as a communication link between them. Therefore a *concordance* explicitly listing all occurrences of all objects represents an important tool to aid in the achievement of full change control [LEH69] particularly if it is provided with data for each level of refinement and a mechanism for inquiry and search.

Once the inevitability of continuing change is accepted a further implication follows. Financial management of the process must be based on *life-cycle costs*. Opportunities for trade offs between the various stages of the process will always exist. It will always be tempting for the low level manager to cut corners, saving time or resources to achieve immediate results. Subsequent penalties are overlooked or ignored in favour of more immediate benefit but may be costly in terms of product adaptability, project costs and the achievement of organisational goals and profitability targets. Only true *life-cycle management* can hope to avoid this, and software life-cycle management techniques [SLC77] are essential to successful long term software effectiveness.

2.5 Additional note

One might of course comment that this property of continuing evolution is not peculiar to software systems. All artificial systems [SIM69] must, like biological systems, undergo adaptive evolution or gradually become obsolescent. But software systems do so more rapidly in general and with more marked consequences.

The reasons for this have been discussed elsewhere [LEH77c] in detail. Here we merely note that the intellectual effort required to design and plan a change is often not rated very highly. Nor are the successive costs which are incurred from inception to final, correct, implementation totally attributed to the change which caused them. Thus the apparent ease with which a change may be initiated, implemented in one instance of a program and then apparently with ease extended to all other instances, leads to the situation in which change is superimposed upon change to yield an ever more stratified and complex system.

That is, of course, in strong contrast to physical systems in which changes are implemented via a re-design process in the creation of new instances of successive generations. Thus the problem of managing change needs perhaps more careful and immediate attention in the software field than is required in other areas of human enterprise. The experiences of the last two decades amply supports this view.

3 The Second Law

3.1 Statement - law of increasing complexity

As a large program is continuously changed, its complexity, which reflects deteriorating structure, increases unless work is done to maintain or reduce it.

3.2 Complexity

The concepts of structure, complexity and entropy are closely related and, in the context of software engineering, all imprecisely defined. There do not, in fact, exist generally accepted definitions and measures such that one may, for example, state the complexity of a program absolutely, relative to an earlier version of itself or relative to another program.

Within the context of Computer Science the term *complexity* (of an algorithm or program) has been used with reference to the number of steps required for a computation [RAB77]. Computational complexity undoubtedly has a relationship to the complexity concerns of the software engineer. But it does not adequately reflect all of the attributes of a program that contribute to the difficulty of working on and with a program so as to achieve required operational characteristics. For example, measures of complexity of computation weight the contribution of a looped or recursive procedure according to the number of steps in the procedure and the number of times it is executed. But from the point of view of the engineer its complexity relates to the semantics of the loop, its input/output relationship(s). Once it or they are understood completely the number of traversals is, in general, largely irrelevant. What is then important is the static and dynamic relationship of each procedure to the remainder of the program.

In other words, from the engineering point of view complexity relates strongly to the macro and micro structures of a program. The programmer and the user must be able to visualise and understand the linkages between the various parts of the program and of its code. Functional connections will exist that represent, or (during execution) cause, interactions between various aspects of the entity or process being modelled or controlled. There will be relationships that arise from programming convenience or a desire for economy. Calls for the execution of a coincidentally (almost) identical sub-sequence from different procedures, or

repeated use of the same name for different semantic purposes, represent examples of structural degradation that increase the difficulty of understanding a program. Yet computational complexity concepts do not involve these or similar system structural aspects at all.

The above discussion suggests that in a software engineering context the term complexity is used to express the degree to which a program is intelligible. It reflects the effort required to deduce its transfer function, the totality of the consequences of its being executed under any one of all possible environmental circumstances. And we claim that this effort depends primarily, though not exclusively, on the extent and pattern of program interconnectivity; on program structure.

With this view-point we may make a number of general observations as follows:

- * Program complexity is *relative*. An absolute measure of understandability cannot be formulated.
- * Program complexity is relative to a *level of perception*. A program may be studied at, say, functional, subsystem, component, module and instruction levels. The topology or pattern of interconnections at each level will be largely formed from a subset of the interconnections at the next lower level. Each will be a representation of system complexity. Which one is selected will depend on the objective of the study.
- * Program complexity is relative to *prior knowledge* about the program and its use. In particular we may distinguish between the *internal*, *intrinsic* and *external* complexity of a program. The former we define as that which reflects only the structural attributes of the code. It expresses inversely the relative ease with which a program may, without prior knowledge, be understood directly from the object text. A formal definition would add the assumption that comprehension of the semantic intent of individual statements requires no effort.

For a well structured program the internal complexity will be related to an *intrinsic* complexity that reflects the variety, extent and interconnectedness of the various aspects of the application; all parts of the problem addressed by the program.

External complexity is seen as an inverse measure of the ease with which a program text may be understood when read in conjunction with its documentation. In the limit it would measure the difficulty of understanding the code when accessible, complete and consistent documentation that describes the intent of individual code sections and their joint action in unambiguous terms, is readily available. External complexity really measures the remnant difficulty of understanding when the program and its documentation are read and absorbed sequentially. With well structured documentation, however, the external complexity measure at one level is likely to be closely related to the internal complexity at a (the next) higher level.

3.3 Explanation

On the basis of this somewhat extensive definition we may now explain the second law which is seen as an analogue, if not an instance, of the second law of thermodynamics.

When a system change, a repair, a performance improvement or a functional enhancement is planned and authorised, the instructions to the implementors will include qualifications that address objectives other than the raw functional requirements. The directives could require the work to be achieved at minimum cost, by a certain date, with minimum performance degradation, maximum performance gain, minimal use of additional resources during execution, or some combination of these factors.

The one directive rarely, if ever, included is one requiring minimisation of structural degradation or even just its restriction to a specified amount. The latter can in fact not be done since measures of structure do not exist. But why should the former be so rare? The ultimate purpose of all software change is economic gain, gain that can be assessed on *completion of the change* in terms of increased cost-effectiveness. The value of good structure is, however, long-term. It is an anti-regressive property [LEH74] whose benefits are apparent only in a negative sense, in that the system does *not* deteriorate, or that there is a comparative *absence* of problems when the system is changed. As a long-term investment, structural maintenance does not, and cannot, figure very high on the priority list of the line or project manager responsible for some specific implementation within system evolution. Hence it tends to be forgotten or ignored until it can no longer be overlooked.

Only occasionally when decay has reached the point where lack of structure makes further maintenance difficult, costly and unresponsive, will so-called system clean-up be initiated to restore system maintainability. During other changes, especially those that implement features not considered when the basic design was laid down, structure will inevitably deteriorate, and the system will decay.

Other effects are also present. In any enhancement process the larger or more revolutionary changes will be delayed. Often the need for them will arise only late in the life of the system and when more subtle changes have taken place in the environment. The likelihood that a desired change matches system structure naturally or can be designed to fit it without serious degradation, decreases with system age. This effect is further aggravated by the superimposition of changes on one another. The net result is reflected in the second law.

3.4 Consequence

The direct consequence of the law is a continuous increase in the difficulty of maintenance and enhancement; a sequence that is broken only by specific action to restructure. The cost and time per unit of work, the number of remnant faults, all increase. Measures of attributes such as performance and quality are likely to deteriorate. Management may, of course be able to counteract and avoid deterioration through the application of additional resources, but cost and time overruns will inevitably result [BR075].

Sooner or later the rectification of the problem will require structural restoration through re-design of at least the worst components or through recreation of the entire system. In either event further system evolution will be delayed while the structure oriented work is undertaken. And this work must address both the code and its documentation. The second law applies to both in equal measure.

3.5 Rules, tools and technical consequences

The rules that follow from the second law must now be self evident. It is not sufficient to create a program using the techniques of structured programming. *Structure must not only be created but most also be maintained.*

It does, however, remain a matter of management choice whether to follow a strategy of continuous or discrete structural

maintenance. That is, one could demand that each and every change to the system must be so designed that no structural deterioration results. Where this is impossible some other additional change would be desirable just to improve the structure in some respect such that the net change is zero or even positive. Such a strategy could perhaps be viable but will certainly be difficult to plan and control under economic and user pressures and while structure is neither well-defined nor adequately and readily measurable.

Alternatively one may ignore the structural consequences of individual changes and plan for or accept the necessity for periodic restructuring. The latter will then represent a time during which progress and even response to urgent user needs will be difficult and costly.

It is not possible to demonstrate that either of these two strategies is in general to be preferred from the economic or the technical point of view. We note, however, that irrespective of the strategy or combination of strategies followed, the net proportion of resources needing to be applied to structural maintenance increases with time [LEH74]. Thus it leads inevitably to a limitation of system size and a need for total system recreation. It also provides one more reason for emphasising the need for life cycle oriented maintenance.

The observations and models of evolution dynamics [BEL77a] provide a tool for the monitoring of complexity that can act as a guideline for the planner. Measures that have been derived and others that are under investigation can be used as indicators of complexity growth [LEH77a]. They define planning objectives and can assess the degree to which a prepared plan conforms or transgresses complexity limits.

4 The Third Law

4.1 Statement - law of statistically regular growth

Measures of global project and system attributes are cyclically self-regulating with statistically determinable trends and invariances.

4.2 Explanation

It is of course this law which, without the benefit of hindsight, is so unexpected; perhaps even unreasonable. After all, the power to take decisions that determine the

course of the project, the content and frequency of new releases, the sequence in which repairs, changes and additions are made, the rate at which they are to be undertaken, the resources applied to system maintenance and growth are all subject to management planning and control, albeit with some dependency on one another and on external factors.

All this is true. Yet measurement of more than six large programs of different types, treated and maintained by quite different organisations reveal, in general, the same pattern of behaviour. Well defined long range trends of various parameters, regular cyclic variations about the trend line and certain invariances can be clearly identified and determined. Much of the data has been discussed elsewhere (eg, Ch 19, this book). Thus here we only refer to the observations in a general way.

The characteristics of the data have directly led us to postulate the third law. In addition the data provides qualitative and quantitative support for the other laws. Each of the systems observed has grown, and therefore changed significantly, in a fashion totally consistent with the first law and the concepts outlined. Continuous change as asserted by the first law has occurred in all the systems observed, despite major differences in environmental circumstances and the period of its life cycle over which each system was observed. It has also proven possible to identify and evaluate an approximate complexity measure. The fraction of modules changed or handled in each release is seen as an indication of system interconnectivity, a complexity measure. For three of the systems the available data clearly supported the second law. It showed a dominant trend of increasing complexity though with a strong two to three release-cycle period as in Figure 1. The same measure applied to a fourth system strengthened the earlier conclusion. The architecture of this system was significantly different from that of the other three, in a way that makes the complexity measure as conceived irrelevant. It is therefore (with hindsight) not surprising that for this system the complexity measure did *not*, in fact, show an increasing trend [LEH77e]. On the contrary it yielded a cyclic variation around a constant or slightly decreasing fraction, as in Figure 2. We now realise that this is precisely what should have been expected with this measure for this architecture. For the two remaining systems for which detailed data is available, the complexity measure as defined could not be evaluated and no alternative measure has yet been formulated.

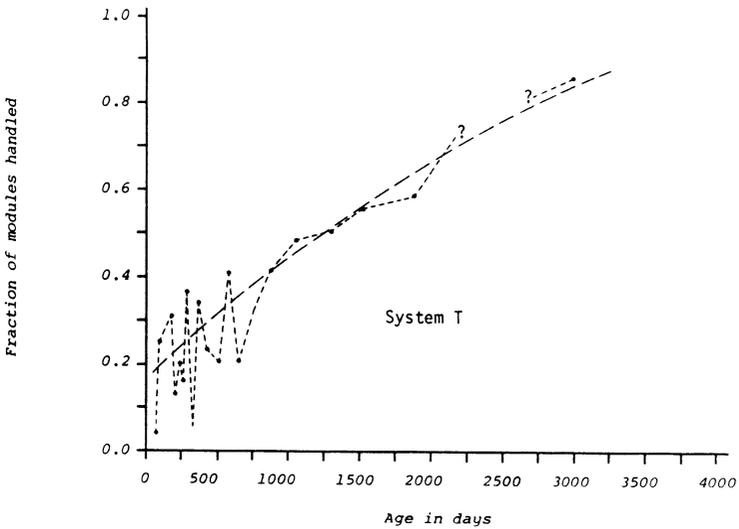
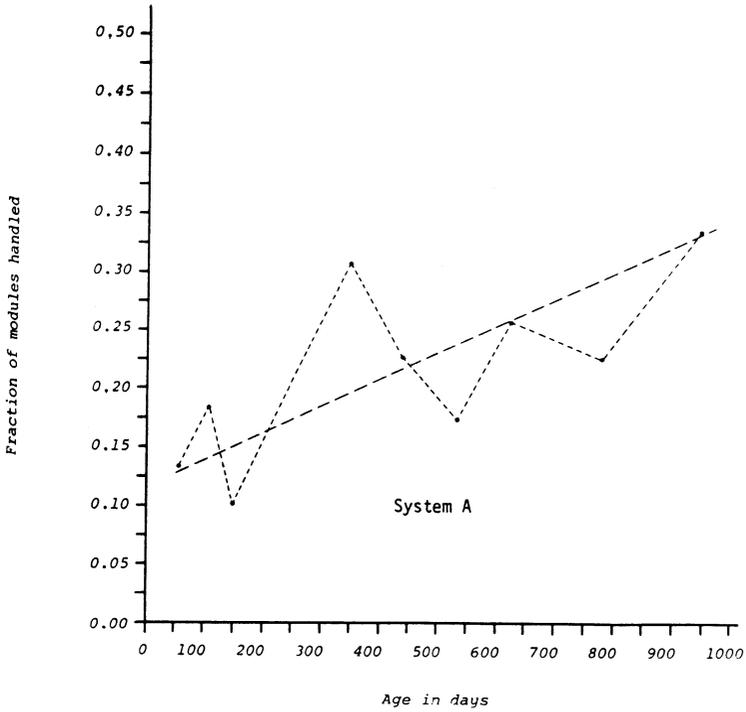


Figure 1 Increasing Complexity

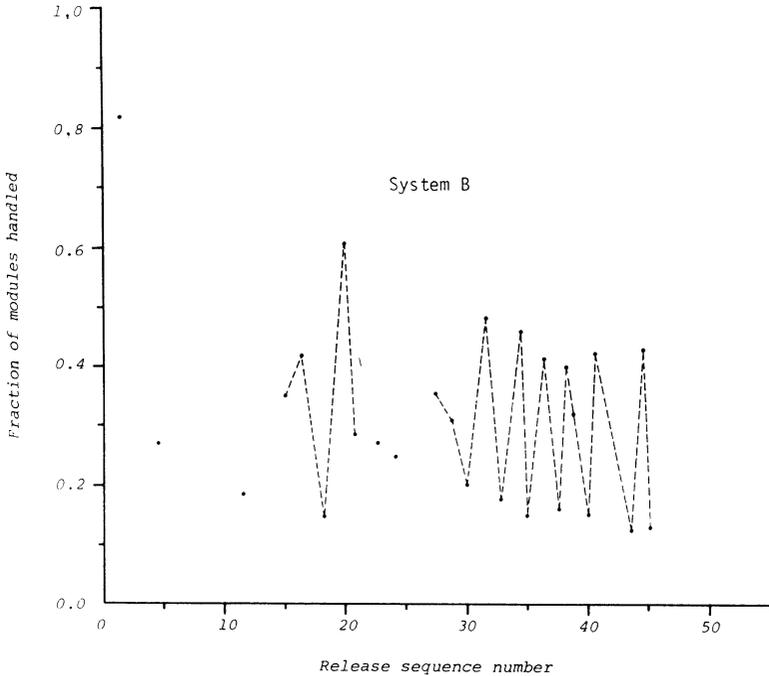


Figure 2 Inappropriate application of the complexity measure

The fourth and fifth law to be discussed later are similarly supported by the available data. We should in fact stress that each of the laws so far formulated was *first* suggested by study of the available data. Only afterwards was each law given a phenomenological interpretation and seen to be reasonable in the light of our growing understanding of the programming process.

4.3 Interpretation

The third law is unexpected because it is counter-intuitive in terms of general understanding of the **management** process, and of the power of the individual manager. But, once observed, it is easily explained in terms of several complementary phenomena. Using terms from other areas of science, we identify them as the *inertial*, *momentum* and *feedback* effects.

Space does not here permit a full explanation and we must restrict ourselves to a brief outline.

Inertial effect

As a program increases in size its complexity (by the second law) and the probability of error also increase. But program code is unforgiving. Any error will cause incorrect behaviour when encountered in execution since for a given set of executing mechanisms the program has a unique interpretation. Moreover because of the high rate of execution, any error is likely to be encountered sooner or later. They should therefore be avoided, or eliminated as early as possible in the change process [FAG76]. But the implications of any change are often widespread and often obscure. Major effort must therefore be exerted and applied to avoid or eliminate all error and to implement and complete a change in such a way that the system settles down once again to satisfactory behaviour.

Once a program exceeds some minimal size it therefore reacts to change like an inertial mass. It imposes a pattern of behaviour that can be locally modified but that offers great resistance to major change. We suppose that it is this inertial effect of the code that makes a programming project behave somewhat more like a process following the laws of a natural science than do other socio-economic systems.

The momentum effect

A program that satisfies our definition of largeness will be supported by a significant organisation with a substantial budget. Moreover it will itself play a major role in the organisational pursuits of its users. All of these together constitute economic and sociological investment and a behavioural pattern and trend that can generally be changed only slowly.

In addition a program is changed only through the actions of people. The identification of potential for change, the management decision process, the design of change, its implementation, integration and verification are all human activities. All engaged in these activities form habits and practices that strengthen both the inertial and momentum components of project and program evolution.

The feedback effect

Both the inertial and momentum effects play a significant role in determining program evolution dynamics. But it is the feedback effect that is most directly observable and gives each system its characteristic features. The management and implementation of change in the programming process leads to an organisation and a process dominated by feedback. In one organisation that we analysed in some detail we were able to identify at first glance some twelve major feedback paths that maintained and controlled the direction and momentum of the process.

We suppose that such feedback loops exist in any large project. The majority are negative in that they tend to stabilise the process, to implement checks and balances. As a result, the main tendency is towards the appearance and maintenance of stable long range trends, as is indicated, for example, by the emergence of invariances as discussed under the fourth and fifth laws. Such stability is, of course, not surprising. Most large programs are owned and managed by large organisations. By their very nature such organisations are dedicated to stability. They will inevitably develop control procedures that generate corrective action to ensure long range stability that helps attainment of corporate goals.

Other negative feedback influences also exist. Suppose for example, that programmer productivity is increased. This might be achieved by the use of programming tools; terminals, higher level languages, on-line program files and immediate execution facilities, for example. Each of these will appear to make it easier for the programmer to generate code. Unfortunately, unless carefully trained and rigorously controlled, this will tend to cause the programmer to *think less* about the program being generated, and to use trial and error rather than intellectual effort. The net result will be more code, but also more errors, less documentation and less structure. And all that in turn will slow down the integration and validation process in the factory or in the field. That is, the net contribution to overall, life-cycle productivity is likely to be minimal or zero. Many other such examples of stabilizing feedback could be cited. The one must suffice.

But all is not stable. Positive feedback loops also exist. Change generates change, improvements lead to the need for more improvements. Poor structure leads to obscure code

which in turn leads to even poorer additional code. This effect, if overlooked or left uncontrolled, can lead to an explosive situation that may ultimately cause the project to fail.

4.4 Rules, tools and technologies

The law we have stated simply asserts that certain regular trends exist in a large program project. It is based on detailed study of a number of very different systems stemming from widely separated points in the implementation environment and usage spectrum. This latter fact, in conjunction with our interpretation of the law as expressing natural properties of the implementation environment as a self-stabilising feedback system, leads to certain rules for successful project management. These include recognition of the fact that the trends may be overcome temporarily by management action. Once appreciated and understood, they can perhaps also be permanently changed. But responsible management demands that the possible consequences of deviation from process and system trends be considered and allowed for if a successful project is to be achieved and maintained.

By a successful project we mean one that produces the required product with predetermined properties by application of predetermined and controlled resources according to a pre-planned schedule. In other words, a successful project has two distinct phases or a sequence, possibly overlapping, of such phases. The first phase is that in which the project is planned and evaluated in both a technical and economic sense. In the second phase the plan is implemented. For example, a program is written or changed. The results of the evolution dynamics studies as expressed in all our laws, but most generally in the third, have relevance to both phases.

In the first instance management should accept as a fact that its project and product has, or will develop in the course of time, certain natural parameters and trends [LEH76b], [LEH77]. Secondly, they should attempt to understand why the trends occur and how they may be overcome. They will observe, for example, a growth trend that can be modelled by some function of time or of a pseudo-time variable such as a sequence-number. The model will be an accurate predictor of the long range growth trend unless management can take informed and successful action to modify the system parameters. Even in the absence of such positive action the system may be made to deviate temporarily from the trend curve with deviation that

consists of both a cyclic and noise component. But then the inherent feedback effects will soon restore the *status quo*.

Unfortunately, for most systems, there will not be sufficient data points to permit a rigorous statistical analysis and the estimation of a complete model. But in all our experience the trends have been clear and the cyclic variations have shown measurable regularity. In such circumstances the trained observer wishing to check a current project or to determine a reasonable growth target for some future change sequence may do so with confidence that he has been neither under-ambitious nor over-confident. This is of course subject to the condition that enough life-cycle history on the particular system has been collected so that the trend model and the variation about it has been established and numerical guidelines determined.

Similar measures have been observed for complexity change patterns, for incremental growth per release and for work rate. Other parameters relate, for various types of changes, to such factors as numbers of existing system elements that need to be examined and the number requiring change per system element added. Still others provide a numerical assessment of global fault appearance rates.

The parameters develop during the early stages of project establishment and system development. They can be determined with increasing confidence using statistical analysis as the evolutionary maintenance process continues and can be used in the manner outlined above for project planning and control. They act as checks on management estimates of work and time required and as a guide to the development of an *achievable* plan.

It will now be asked why management must ever accept the limitations implied by the identified parameters. Should they not be seen as a challenge to be overcome? Cannot normal management practice steer a project in any desired direction at any required rate?

The history of many major software projects provides the answer [BR075] (12.1). But it is nevertheless our conviction that the evolution dynamics studies will lead to an understanding of the process that will permit its significant and permanent improvement.

12.1 (Eds) *The reader is referred to Chapter 18 for a more complete discussion of the answers to this question.*

One of the properties of a multiple loop feedback system is that any local change of forward or feedback characteristics is unlikely to produce any significant change in the externally observed performance. Feedback via the other, unchanged, paths will ensure that. However, if by chance a significant change is nevertheless achieved it is likely to be experience either as a decay process or as instability. Both of the latter responses to process change are generally not cost-effective and are highly undesirable. Thus real improvement can only follow understanding. In the first instance we must accept the system as it is.

In summary then we may assert that the intrinsic non-linear feedback structure of any programming organisation and process together with their inertial and momentum properties suggest the following:

- 1 Apply the techniques of specification and structuring as recognised by the wider programming community.
- 2 Measure project and system parameters such as those described in the published evolution dynamics literature and estimate project models, modifying them as new information comes to hand.
- 3 Use the models to plan further evolutionary maintenance using both long range and cyclic trends (but see the next point below). Where technical or business considerations make it appear desirable to exceed the identified bounds, allowance must be made for deterioration in quality and/or time and cost over-runs. It may however be possible to avoid these effects by appropriate preparation such as clean up, or overcome them by subsequent corrective action.
- 4 The activity of model identification and interpretation should yield increased insight into and understanding of the programming process as practised locally. If this is achieved, it may be used to improve the process gradually, to yield more cost-effective parameters, increased productivity for example.

In summary, organisational and project characteristics cannot for long be forced along arbitrary directions or at arbitrary rates if these happen to be counter to intrinsic trends [BR075]. Wishful thinking and management edict can never be more than temporary panacea. In the first place facts must be accepted as they are. Measurement and analysis can then

provide a guide to attainable commitment and also the insight subsequently to permit permanent improvement of the process.

5 The Fourth Law

5.1 Statement - law of invariant work rate

The global activity rate in a large programming project is invariant.

5.2 Explanation

This most surprising of results is in fact the best founded of all our observations made for each of the systems. To make it meaningful, however, the terms 'programming activity rate' ('work rate') and 'invariance' must first be clarified.

We state right away that measures of the resources, for example man-hours, applied to the system do not yield a measure of the rate at which work is executed on the program. One of the surprising results is, in fact, that none of the growth or change rates observed correlate in any way with manpower or other resource expenditure rates. This suggests that large programming projects tend to operate in a saturation mode where a change of resource or of staffing level, while still keeping everyone busy, has no perceptible long range impact on evolutionary characteristics. The evidence to hand does not yet permit us to postulate a law that adequately abstracts this observation. But confidence in its reality is strengthened by the success of the so-called chief-programmer team concepts for project organisation [BAK72], [IL76].

What then are the measures of global activity and activity rates to which the law refers? For several of the observed systems, data has been available for each dated release of the number of modules changed or handled to achieve the new release. For another system the number of changes incorporated in each release has been recorded. Some of this data is shown in Figure 3 as a cumulative plot of modules changed or handled, or of changes made as a function of time. The overwhelming impression of these plots, confirmed by statistical analysis, is one of linearity. That is, the average rate of activity as measured by these parameters has remained *constant* despite changes in definition, programming and management experience, tool support, staffing and resource levels and so on.

This constancy is the invariance to which the law refers. We note that it is an average rate since the rate from release to release or from period to period varies significantly. But such a periodic variation where a high rate release, higher than average, has a high probability of being followed by a low rate release, lower than average, is precisely what could be expected (with hindsight) from a self-stabilising feedback system. Too high a rate of work will probably cause less careful design, sloppy implementation, more errors, less documentation, tired programmers. The inevitable consequence is a period of difficulty with some attempt at clean up, during which time the work emphasis is on the rectification of errors, shortcomings and structural and documentation deficiencies. Hence it prepares the system for a further period of high-rate working. And so on.

Such a cyclic pattern is clearly present in the data. We have also observed that module based measures show a more consistent pattern than do trends based on either instructions or on changes. This too is not surprising since on the one hand individual statements, unlike modules, have little logical independence. Absolute changes on the other hand represent a much more variable work load than do modules changed or handled. It is these module-based activity units that have given the most consistent results so that we now use them exclusively in our planning applications. We have accepted the observed cyclic invariance as reasonable, interpreting it as the reflection of a self-stabilising work rate. The net invariance then indicates a highly stable system. The question still arises, however, of why the global rate should have remained invariant over a period of technological advancement and major system growth. As we have already stated, the answer to this lies in the very nature of the large organisations that develop when large programs are produced, maintained and used. In particular we assume that the main drive is always for increased production and profitability of one sort or another. Increased productivity in one programming area is, however, often most easily achieved at the expense of decreased productivity in another, at another time [LEH77f], with the connection between them escaping notice. It is precisely this type of relationship that leads to long term stabilisation or even decline. However, when higher management becomes aware of such a decline, it will divert funds from progressive investment in product development to anti-regressive investment in technology and tool improvement [LEH74]. But however well intentioned such investments are at the time they are made, to whatever extent they are then seen as necessary and

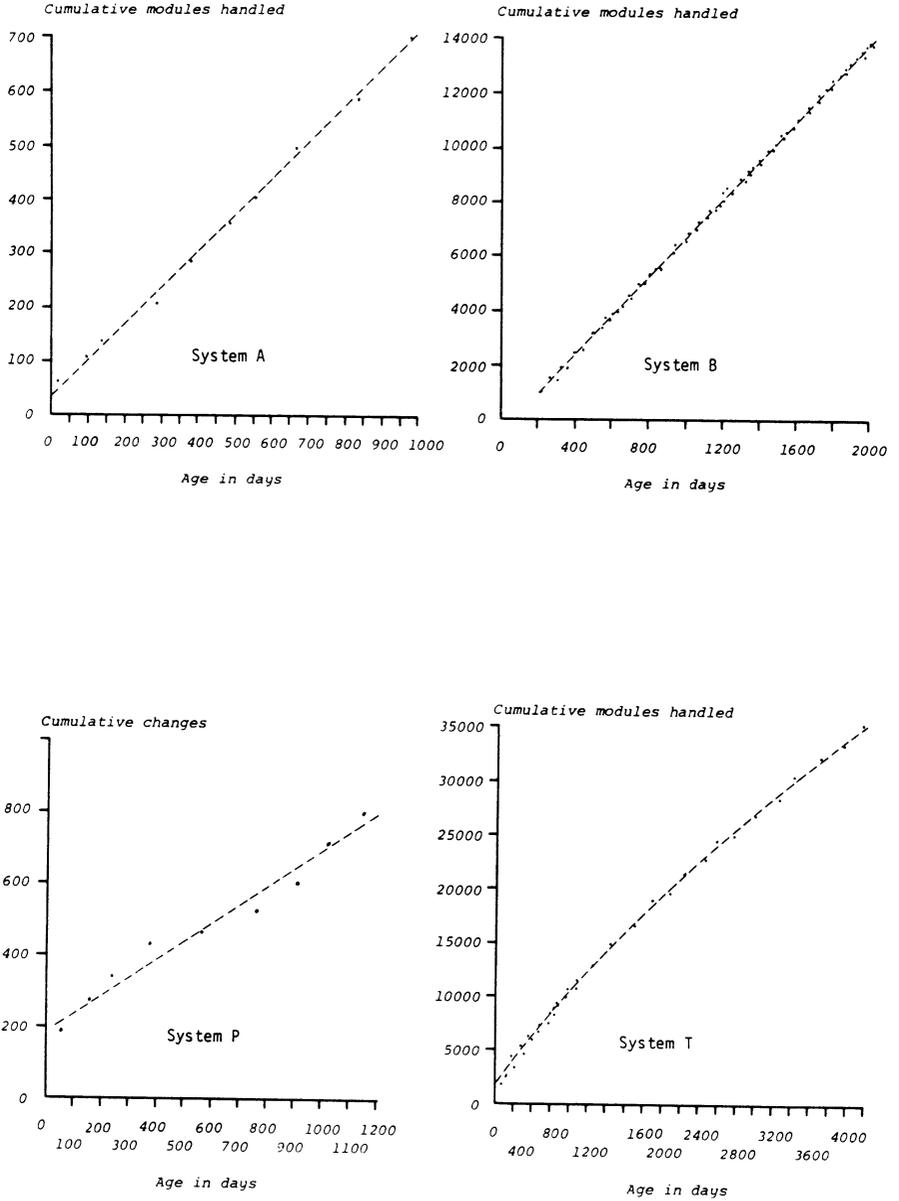


Figure 3 Invariance of global work rate

expedient in maintaining - or improving - productivity, the time will come when the status quo has been restored. Then all methodological work is seen to absorb funds that could be 'better' invested in product development. So the work is stopped, funds are switched and the cycle proceeds.

This phenomenon has been observed in at least two programming organisations. We feel certain that it explains the overall constancy of the work input rate for all our measured systems. Nevertheless, we cannot explain precisely why the result should be a simple invariance rather than some other well defined trend.

5.3 Consequence

As observed above, one should expect that sufficient understanding of the program evolution process as a feedback network will permit a permanent change of work rate. However, the fourth law based on our observations suggests that wise management with a desire to meet planned objectives would, *in the first instance*, accept the limitations implied by the invariance with possible variations as determined by the cyclic pattern. From time to time business or other strategic considerations might make it expedient to force a pace outside the rates identified by the historic trend. But this would then be done with a full awareness of the infringement and its likely consequences. Now any competent manager planning a spurt of activity to reach some specified target would plan for a subsequent period of lower productivity to allow for corrective action; for the recovery of both system and personnel. What the evolution dynamics investigations have achieved is to measure and to size the net effect, taking into account all the influences, known and unknown. Thus it provides a technology and a tool.

5.4 Rules, tools and techniques

Any manager will intuitively resist the suggestion that there exist limitations to what can be achieved, particularly where additional resources are available for application to the system. But when managing the maintenance and enhancement of large programs, experience as now formalised in this fourth law denies him this possibility, except perhaps in spurts that must then be compensated for. Thus the guiding rule must be to live within the possibilities established by an analysis of project history.

Evolution dynamics studies have developed a technology that permits one to measure and to size the stabilisation effect in any project. It yields planning constants and guidelines whose observances will permit the setting of functional goals that can be met without cost or time over-shoot. The rules and limitations arising from this and the other laws, if consistently followed over the project and product life-cycle, will extend product life and prevent sudden project failure. Once again we may emphasise that they will also lead to developing insight that must eventually lead to and permit improvement of the process, the achievement of a consistently higher work-rate for example.

6 The Fifth Law

6.1 Statement - law of incremental growth limit

For reliable, planned evolution, a large program undergoing change must be made available for regular user execution (released) at maximum intervals determined by its net growth. That is, the system develops a characteristic average increment of safe growth which, if exceeded, causes quality and usage problems, with time and cost over-runs.

6.2 Explanation

This law cannot yet be formulated as precisely or concisely as the earlier ones. The system behaviour and data on which it was based is amongst the earliest identified. Yet only recently have we begun to understand its significance and generality. Available space permits only a brief summary of the previously published data [BEL76] [LEH76b], [LEH77e]. This has enabled us to demonstrate the phenomenon for the three systems for which the appropriate measures were available.

Each displays a steady average growth rate (different for each system) of say N_s modules per release. The actual growth increment N_{s_i} has varied cyclically from release to release. That is, an above average growth in one release is almost invariably followed by a below average increment in the following release. The actual levels are subject to management decision and therefore contain a stochastic component. However, management must clearly be influenced in the planning process by feedback effects reflecting both the pressure for additional growth and the inevitable consequences of changes and additions implemented in previous releases.

The most striking feature of the data is, however, not the well defined average increment N_s and the regular cyclic pattern. Detailed behavioural records are so far available for only one of the systems studied. But for this system each release with incremental growth exceeding $2N_s$, suffered from serious reliability, performance, delivery and cost problems. Moreover each such release was followed by a fission process during which part of the system was removed. This produced a negative growth increment for the following release. The net result was that the average incremental growth remained essentially unchanged.

Just as previously noted for the invariant work rate factor, management disregard of the intrinsic growth rate limitation may be temporarily successful, permitting a release whose incremental growth exceeds the recommended maximum. But subsequent events force a restoration to the historic pattern. And as in the previous case we see the pattern as a consequence of feedback effects, represented in this case primarily by environmental reaction to the consequences of excessive growth.

How may the overall effect be explained? Why do we have such confidence in its generality to feel able to formulate and propose it as a law?

The first four laws have been seen as the *net* consequence of many individual decisions and actions. They lead to statistically regular behaviour whose detailed parameters are determined by inertial and feedback effects. The fifth law, on the other hand, is seen as a direct reflection of the *role of the individual* in the design, implementation, usage, and change of large programs. Naturally, however, observation at a global level tends to reveal the average individual response.

When encountering a system change, each individual, whether as designer, implementor, salesman or user must understand that change in the context of the system and of system behaviour as a whole. The absorptive ability of people varies. But each individual can only absorb a limited amount of new information unless, and until, he achieves the familiarisation that results from running and using the program. That is, the degree of understanding and appreciation by each individual involved, of the full significance and consequences of the changes and addition to a system, will decline rapidly once it exceeds his absorptive limit, unless an opportunity exists for using the system;

working with it as a living entity. We believe that the facts observed and described by the fifth law are the consequence.

6.3 Consequences, rules and tools

It may be that in future more complete understanding will permit determination of the quantum growth limit for an organisation and a system. It could be expected to depend on the nature of the large program application, on detailed system function and structure, on the languages used, on the level of experience and so on. One would hope to be able to estimate the optimum increment from a knowledge of the environment and to improve it through use of appropriate design and implementation techniques.

But at the present level of knowledge and understanding, it becomes desirable and advantageous for both long range strategic and immediate tactical planning to monitor and measure a project during its early releases and to establish a *de facto* factor. Once estimated it would then be used as a guide for release planning.

It is of course not necessary to restrict individual releases to an incremental growth of N_s . A growth exceeding $2N_s$ should, however, not be permitted unless there are solid strategic considerations for doing so and accepting the consequences - at the very best a greatly extended release interval. The time allowance that is needed may, in fact, be determinable out of a historically established, statistically determined, unfortunately non-linear, relationship between required release interval (stabilisation time) and incremental growth. Moreover, even where the increment can be held to be less than $2N_s$ one must accept the fact that the following release should be restricted to a growth proportionally smaller than N_s if major problems are to be avoided.

In summary, study of the per-release growth history of a system can produce a set of indicators that play a direct role in determining release content. In conjunction with the other indicators they can help to make the evolutionary maintenance process more manageable and more predictable.

7 Conclusion

In the above we have summarised a seven year study by proposing a set of five laws. These abstract and express the common set of phenomena and behavioural patterns observed on some six large programs whose history has been studied. These programs together span the spectrum of application, usage and implementation environments. Clearly they represent only a small sample. But taken in conjunction with universal experience in the development, maintenance and application of large programs it seems highly probable that our observations and the conclusions therefrom represent a class of fundamental truths. The behaviour is a complex consequence of programming methodology, software engineering and economic and sociological factors. Presentation of our interpretation and of the data that supports it has had to be condensed so as to achieve a reasonably complete coverage. Some of what has been written is conjecture rather than established fact, but if that serves to stimulate observation and discussion that is all to the good.

We believe that the existence of program code provides a smoothing effect on the socio-economic structure represented by a large program project. This results in the regular behaviour that we have observed, measured, modelled, interpreted and described. By extending the study to more systems and parameters we may hope to develop a more complete theory. The latter would in turn increase the number and accuracy of the management tools available to plan and control the programming process. Perhaps more important still it would increase insight and understanding of the process and so permit its significant improvement.

Finally, we must briefly ask 'What of the future?'. Do these laws represent a fundamental block to system size and to programming productivity or can they be overcome? The question has been discussed elsewhere [LEH74], [LEH77b], [LEH77c] and the conclusions reached can only be summarised here.

Basically it is likely that the laws as postulated will apply to all large systems that are structured and integrated like present day large programs. However, application of improved programming and management methodologies including those that follow from the evolution dynamics study, should yield greatly improved system parameters. LSI and microprocessor technologies, together with effective program and interface specification and control will yield large systems

constructed out of well specified active elements interconnected via standard interfaces. In this way the large program problem will disappear at the level at which it is presently encountered. Whether it will re-emerge at a higher level is a matter for future generations. We may help to prepare the ground now for its avoidance or for a more simple solution.

CHAPTER 13

STAFFING PROBLEMS IN LARGE SCALE PROGRAMMING*

1 Introduction

In all software projects, whether operating systems or large application packages, the following phases of distinct activities can be identified: firming up the requirements, translation into system specifications and decomposition into major components, design of components, programming and debugging, system integration and test, operation, and never ending maintenance.

It seems fairly obvious that people of different backgrounds, expertise, temperament and expectation should populate the distinct phases. For instance, bold creativity and keen curiosity may be essential for system design, but could be detrimental in negotiating requirements with a demanding but naive customer, or in performing routine maintenance.

The above sequence of expert activities appears reasonably simple. However, matters become somewhat more complex if one considers that some specification or design choices cannot be made without an exploratory implementation (13.1) or rudimentary programming. In this case results obtained in a later phase must be fed back to an earlier one, and this cycle may repeat itself.

Phases thus do not form a simple 'top-down' sequence of work activities in increasing detail; first global decisions, last the code. Rather, the process seems inherently iterative, eventually converging on a final structure in which the basic parts, ie, program modules, gradually take shape. After this point the inverse of decomposition follows: pieces become collected into ever larger units, interspersed with intermediate and final system tests, from which occasional warnings are fed back to the design decision makers. And the last phase, maintenance, becomes the perennial source of request for modifying original requirement, design and implementation.

13.1 (Eds) *'Prototype' in current parlance.*

Reprinted with kind permission from 'Why Software Projects Fail', Infotech State of the Art Conference. Copyright 1978, Pergamon Press.

The complexity of this process may be expressed in terms of the normalised cost of a 'debugged' piece of software, which now stands at around \$120 per statement. Since this cost, except for the computer-time-consuming test phase, is almost entirely that of human effort, proper staffing is of primary importance in reducing cost by eliminating waste. Waste is generated by the wrong match of job and people; whenever people are idle due to unexpected delay in another phase or to poor planning; whenever an extra iterative cycle is induced by too hasty decisions; whenever unnecessary complexity slows down an otherwise routine project activity.

Many models and methods have been published, some claimed to be universally valid in predicting the manpower need distribution over the entire life-cycle. Instead of describing these methods here, we simply recommend that organisations carefully analyse their own past project activities or learn from similar projects, and then use the gathered data to predict and improve future projects. Even if precise values may not be carried over from one project to another, trends often can. The Program Evolution Dynamics work by L A Belady and M M Lehman is an example of systematic project monitoring. But let us now turn our attention to the first major problem of manpower specialisation.

2 Decomposition of Software Systems

We wish to distinguish between large collections of programs and large software systems. Programs of a collection serve a single purpose, yet communicate with each other infrequently while running on a machine, as do parts of a program support package, eg a set of compilers, simulators, editors and other utilities. In contrast, the components of a large software system, while running, make frequent use of each other's services and data. Often the software is the only means of coordinating otherwise independent hardware apparatus: an operating system is an example. Our interest here is limited to such software systems.

Against a fixed environment and given enough time, an expert could perhaps construct and gradually debug singlehanded even the largest of known programs. Time soon becomes a problem though and, as can be shown by extrapolating productivity of complex one-man projects, the job would take decades or more. In a realistic situation, however, requirements are not fixed and more than a couple of years' development time is rarely tolerated. More people are thus employed following the time

proven principle of subdivision of labour (13.2). In what follows we shall consider systems whose development keeps organisations of at least two levels of management busy for years. In these organisations heavy demand to coordinate activities, constant changes in the environment and commitment to tight schedules are the way of life and appear to dominate the process. The result is the increasing formalisation of many procedures and plans whose documentation is considered utterly unnecessary in a small team. In order to control the exploding bureaucracy, the independence of components, ie, the conversion of a software system to a collection of programs, becomes the acknowledged goal: the less communication between system components, the more independently can they be developed (13.3).

Nevertheless, the association of system components and branches of the human organisation must be done painfully early, when the system hardly exists. There is no time later to adjust the component boundaries which by then are firmly established among organisational groups and product parts.

3 Specialisation of Knowledge

Programs written by individuals or by small teams are, more often than not, manifestations of widely known computations, sometimes riddles, or other procedures. Examples are sorting algorithms, the eight queens problem inventory control. These problems can often be related to document studies: relevant understanding and even solutions exist prior to programming. The programmer then acts as the translator of human thinking to computer procedures, much like an artist capturing projections of reality on some medium.

In contrast, during software development and largely due to the strong tie between man and project, new information is generated: the exclusive product knowledge. While creating operating systems, for instance, entirely novel problems must be solved quickly. The result is that the solutions remain known only to the designers who, lacking time for documentation as usual, keep this knowledge for themselves,

13.2 (Orig) Fred Brooks [BR075] expertly discusses the phenomenon of why doubling manpower does not halve development time. We will not cover this non-linearity here.

13.3 (Orig) With respect to the problems of system decomposition, see the studies of D L Parnas, G J Myers, L L Constantine and others.

inaccessible to others. Even worse, knowledge may become further fragmented and restricted to non-communicating individuals. Only the development and subsequent teaching of software system science and methodology, as opposed to programming technology can solve the major problem of large-scale software: understanding and clear representation of solutions should precede system construction.

Decomposition of the product as described earlier is not always sufficient, and an extra dimension of labour subdivision is necessary. This is the assembly line approach, which has been so successfully employed in manufacturing processes. In software terms the rationale appears as follows: system architecture, or high level design, is a speciality: detail design thus becomes the task of another team which subsequently passes the specified system building blocks or modules to the coders. Machinable modules then become recollected, and integrated into increasingly large units by separate teams at each stage, and finally the entire system is tested by the experts of this particular activity. Everybody is specialised in a given job and several systems or system versions flow along this human chain at a high rate. By the time of integration the designers have long forgotten the product being tested and are busy creating and learning the next. The fallacy of the assembly line is that it works only where product knowledge is irrelevant to quality, independence of elementary operations is possible and processes are unambiguously specified. Furthermore, only local skill is needed to perform the activities well, and no choices are to be made, no relationships to be observed; these problems must be resolved prior to the set-up of the assembly line which then works well.

In software, process knowledge is about designing, coding, testing, and integrating programs, as well as managing these activities, while product knowledge is the understanding of how programs work individually and in cooperation with each other. Broadly applicable methods have been developed for the former, while product knowledge remains very difficult to generalise, teach, communicate, or even preserve, due to lack of easy 'externalisation', ie representation of ideas. This makes the accumulation and subsequent application of past experience also difficult. Literally, we have to rely on 'internal' knowledge residing in the heads of project participants. The quality of important system attributes, such as performance or usability, instead of being expressed as requirements against which the evolving system can be

checked at different stages of the process, remain only hopes based on past excellence of the team. Given this situation it is doubtful that the assembly line is the right approach.

4 Staff Requirements of Maintenance and Redesign

We have thus far discussed that development manpower must be subdivided if rapid development is essential. The subdivision has been shown to be effected by the combination of the early decomposition of the software into workable components and of the specialisation of process functions. The coordination of activities, which are thus fragmented along two dimensions, presents serious management problems, further aggravated by the fact that development continues beyond the first customer shipment date: an endless system evolution follows which is a mixture of further development and repair-like activities. We shall call all product modifications which occur after first system delivery, maintenance, as distinguished from design, even if maintenance has the flavour of design. We list below the major constituents of this perpetual maintenance process.

First, the system is far from being perfect and its use in real life soon gives rise to a never ending stream of malfunctions, observed and then reported back to where it was generated. Second, the inventive spirit is always alive, resulting in constant attempts to improve by system developers, and in introduction of new apparatus by hardware designers. There is ample evidence that product enhancement continues virtually forever. Finally, the very use of the system enlightens customers to present the manufacturer additional requirements for convenience, capability and function. The ultimate software product is the bit string on the tape from which it is loaded into the machine, in order to bring the computing system into dynamic execution. As we can now see, the string itself is not static either: maintenance as defined above keeps it in a constant flux indefinitely.

Product flux is not unique to large software. For example, we could view the development of the automobile, at least confined to one manufacturer, the evolution of a single system over several decades. An already well developed tanker aircraft actually evolved into the first successful commercial jet liner which has since been continuously improved. The history of computer hardware might be similarly characterised. These cases demonstrate, in general, that new design is considered a major investment and

it seems cheaper to change something than create it from scratch.

Software is perhaps too easy to change but, paradoxically, very difficult to change predictably. While modification of a piece of hardware is quite visible and usually requires the cooperation of several persons plus often massive equipment, software modifications are hard to monitor: usually paper and pencil suffice in the privacy of a single programmer. Moreover, changes may make necessary additional changes elsewhere in the system which frequently remain undone. This appears even more serious since, following observations made on several large systems, the extent to which the modification spreads tends to increase as the system ages. For systems consisting of modules as building blocks each modification will, on the average, involve more modules than its predecessor did.

Another special software problem is that repair is actually not repair at all. While hardware physically deteriorates due to wear, corrosion or fatigue, and repair is then defined as the replacement of the component in order to bring the system back to its original state, the elimination of software malfunction is performed by changing away from designed or constructed state. This often leads to redesign, further reducing product stability, and brings up the next problem: who should maintain software?

As we have seen, a stream of change requirements is continuously generated during the entire life of a piece of software. Since repair is actually a series of engineering changes, the designers are the best candidates to perform the corresponding modifications. They can then select the best fix such that it be the most localised. If secondary changes are introduced, the designers' intimate knowledge of the product is indispensable to make the original system structure prevail. Unfortunately, employing designers for redesign is a luxury which cannot be afforded.

The system maintenance period is so long that the spirit of the original staff cannot be sustained by the often trivial modification activities: people need more challenge. Furthermore, system designers represent a valuable resource needed to design next generation products. The experience, the acquisition of product knowledge which is not formally teachable, makes the designers of the old the architects of the new. This double candidacy creates an expert bottleneck, leaving maintenance often unstaffed.

Eventually, however, the maintenance crew must be staffed. Since its members cannot possibly have as high a level of product knowledge as the designers do, the system should be kept simple and clearly documented: the more unstructured the system, the more deterioration of structure will take place during maintenance. Lacking the necessary product knowledge, maintainers usually select those changes which are the easiest to implement and, under schedule pressure, disregard preservation of clarity and structure.

5 Productivity in Programming

By classical definition, productivity is the useful output per unit cost of human activity. The (at least approximate) estimation of productivity is obviously useful for planning purposes. But even more important, measuring productivity to compare groups or individuals, or as a basis for reward, is a significant motivator.

Fred Brooks in his book 'The Mythical Man-Month' [BR075] points out in some detail the problems related to productivity, centered around the observation that in software projects, as in many other human endeavours, productivity decreases as the size of the team increases. In other words, the next member's contribution to team productivity will be lower than the average of the original team. The usual explanation is that a large part of team work is communication and that intra-group communication demand increases more than linearly with group size.

This points to two potential remedies: the reduction of the information necessary to communicate and the improvement of communication efficiency. To achieve the first, one should be able to identify pieces of work, ie software components, which can keep individual designers, implementors and testers busy without the need to exchange information among each other. Relying too much on independence could be dangerous though, as available error statistics testify (13.4) the majority of design faults are omissions, which suggest that vital information could be missing or not available to designers.

The most important contributing factor to efficiency of communication is the right selection of people. The era of the isolated whizz-kids in programming is fading; openness in revealing solutions, even problems and difficulties and total

13.4 (*Orig*) See, for example, M E Fagan's work

visibility of all components, documentation and sketches to anyone for the asking are absolutely essential. Accordingly, formal inspections, exposing large structural charts on the wall, or hiring extroverts all reduce the intra-group communication problem.

The most frequently used dimension to measure programmer productivity is the amount of finished code produced per unit time (or unit cost). The associated problem is widely recognised: although productivity measures are supposed to motivate for cheaper programs, clever programmers invented the other method of increasing the productivity as a raito, namely by increasing the numerator. More code for the same software function appears as good as a smaller module produced faster.

This has disastrous effects; not only do we have a larger product, and one which consumes more space and perhaps execution time, but we must carry into the endless maintenance phase a bulkier and certainly more complex program, with a probably more than linear cost increase. The productivity measure, originally meant to be a motivator, becomes counterproductive.

An interesting proposal, at least for program modification, is to reward the removal of instructions more than the addition of new ones. This scheme recognises the extra intellectual effort needed to replace, rather than simply add, program pieces. And it also conforms with the advice: hire people who love simplicity; reward any reduction of complexity, even if this contradicts traditional productivity measures.

6 Additional Challenges

We have seen earlier that, for large software such as operating systems, maintenance is separated from development, and another distinct organisation responds to error reports. Many such errors are expected to be reported, requiring immediate action, and the size of the organisation is comparable to that of the developers. We have discussed that the maintainers often lack system expertise.

Nevertheless, right after system or new release delivery, service, ie maintenance, actually performs a task which appears to be the extension of the system test phase. At present, testing is a rather sparse sampling process, since real life use patterns are hard to predict and, even if

known, cannot be represented by a test load of manageable size. Real operation becomes then the test run and the maintainers become the crew which must bring the system into its final shape.

The major problem with this way of testing a young system in particular is that a significant portion of the discovered errors have their origin in design, thus spanning a long iterative development loop. The teams and their work involvement are widely separated in time and space, even on the organisation tree. The already mentioned lack of clear system representation methodology, so indispensable in the maintenance of complex machinery as, for instance, aircraft, further aggravates the situation. Nonetheless the system must be fixed at all cost, though fixes, like the original system, are not perfect. It was observed that, as time passes, an increasing portion of errors is traceable to earlier fixes. Fixes become thus more complex and diffused, impacting a greater number of modules. Accordingly, more people become involved and interference between fixers increases.

As in the case of executing programs concurrently, several concurrent fixing processes during system evolution require careful synchronisation, which should be based on product knowledge. But synchronisation of widely separated teams is difficult, so the attempted solution is another system test to find how coexisting fixes influence system behaviour. This brings us back again to the problem of the insufficiently representative test load. Nevertheless the changed system will be delivered to the customer's shop where residual errors will interfere with regular operations.

It is therefore not surprising that the customer is selective when offered another version with new fixes. While he experiences a reasonably smooth operation, why should he give up satisfaction for uncertainty? In a multi-customer situation, for example, the new fix could be a response to error discovered by another customer while, due to differences in use pattern, the same error may never surface elsewhere. This general reluctance to accepting new versions creates a variety of distinct systems and extra headache for the service organisation.

Consider modules as basic building blocks, perhaps several thousands in number. For the manufacturer the ideal situation is that of well-spaced releases and ready acceptance of fixes. In this case, there is only a single

version for each module at any given time, even in a system with many installations. With selective customers, however, who occasionally reject the incorporation of new fixes, some modules must have several valid versions: the most recent one, as well as one or more predecessors which may still be actively used at some installations. The predecessors cannot be invalidated, documentation and bookkeeping thus simplified, since the old versions still participate in the environment against which further errors could be reported and their repair requested.

It is easy to demonstrate the explosion in the number of module versions. Assume that an error is discovered, say in module A, in the presence of module B. Under some circumstances it may happen that module A cannot be fixed such that its single new version satisfies two or more systems each having a distinct version of B. Note that multiple versions of A must be offered even if only one installation reported the error. Every customer wants to be prepared against erroneous system behaviour discovered elsewhere, even if he rejects the corresponding fix.

The manufacturer wishing to control complexity will therefore try to force customers to accept unwanted fixes, by constructing a dependency network which contains all fixes made to the system. He can declare, for example, that a fix may be added to the system only if accompanied, or preceded, by a given set of other fixes. The additional task of creating and updating this network, which also reflects business considerations, presents an extra burden to the fixers, ie the maintenance organisation. It is not clear what kind of background of experience suits best this type of work.

7 Customer Involvement

The need for good relations with the customer is not restricted to maintenance. There is evidence that with the best systems the customer participated in the early requirement and specification phases of the project and remained in close touch during the entire life cycle.

The 'man-on-the-moon' project, for example, depended on the cooperation of many subcontractors, some of them in programming. It was soon learned to spend quite a period of time, up to a year, on the requirements and specification phases of large software packages, in the form of a dialogue between the parties involved. Eventually, the so-called

'base-line' system was reached, with an associated agreed-upon cost. Any subsequent deviation from the agreement, whether induced by unforeseen factors or mutual alignment among several subcontractors, was always formally treated: original documents updated, extra cost, if any, negotiated. This method worked so well that it was suggested that for projects without an obvious, easily identifiable customer, - often the case for operating systems - at least a 'virtual' customer should be appointed to play the role of a real customer with 'need, money, and authority'.

It was also observed that system programmers, quite remote from the marketplace, often feel uneasy about their lack of 'feel' as to what, in a developing product, is considered by the customer important and what is not. Slow rotational exchange of field and maintenance personnel with in-house, and often isolated, developers more than likely helps to alleviate this problem and eventually everybody will have had the chance to work closely with the customer.

8 Experience and Education

Experts agree that the single most important characteristic of a team programmer or professional software engineer is love of simplicity. However, the most capable and brightest programmers often love complexity, perhaps to exercise their cleverness; and if there is not enough complexity present, they create some artificially. This results in non-modifiable, impossible to maintain programs; and to make the situation even worse, by the time the system is in operation our creators are already busy to build the next generation monument.

The solution is not to exclude the bright but gently blend them into the team of all professionals, to create an atmosphere of balanced inventiveness. Every project presents a wealth of tough and juicy problems anyway, asking for decent solutions. But the spirit must be that of engineering: innovative application of already tested ideas.

Another frequent mistake is the rapid promotion into leadership, based purely on technical capability. Excess curiosity of the very bright often leads into virgin territory, away from established methods which are required to refine the product and to increase systematically its quality. The best way to use the 'whizz-kids' is in advisory, consulting and trouble-shooting positions.

At present there is not enough evidence as to which academic discipline serves best the staffing of large software projects. Since programming is difficult to teach, and team programming even more so, the best policy seems to be the on-the-job training of the college educated, preferably from an engineering school. But even more important is the preservation of know-how which can be acquired only by doing actual system work. Promote those who are willing to transfer, and utilise their experience by moving them with the product, into sometimes less glamorous jobs, eg, putting the system into operation, redesign and tuning. However, resistance to this is likely since it is easier to abandon the just created fragile system and then to turn to the next challenge than to be involved in a somewhat more repetitious activity.

At the other end of the quality spectrun are the 'error-prone' programmers. If they fail in a series of projects, they should be reassigned, away from the positions of easy access to design or modification. Remember: modifying software is easy, but difficult to do well.

The major message is that experience with novel, complex systems is an investment, which must be preserved and rewarded, in order to help the product survive its required life cycle. It is wrong to think about programmers as general purpose, interchangeable components of a development and maintenance process.

However, formal education must also be continuous. Programming, as other crafts, is habit forming. New tools or techniques are then ignored, their acquisition resisted. Management should not be too parsimonious with the education budget. Courses on programming and system technology must be offered, otherwise old mistakes, bad structures, wasteful programs become perpetuated. And the mandatory formal courses should be complemented by slow rotation of personnel, thus introducing into the experienced team the young who learned new methods without the burden of first having to unlearn the old.

10 Team Spirit and Communication

As we have seen earlier, some software projects contain a large component of novelty: new problems are answered by fresh solutions; new methods employed, and all these summarised into new concepts. Not surprisingly, new words solidify into a jargon, spoken and understood only by the

team, whose members are in daily contact with each other and with their work.

This furthers intra-team communication and warms the spirit to the benefit of the project: a more efficient process and a better quality product are likely to result. The great problem is, however, that 'externalisation' of the newly created knowledge is almost impossible. The novel methods and the innovative aspects of the product will be fully understood and appreciated by team members only.

While within-team communication is efficient and half-words exchanged suffice, describing the product to the user, for example, becomes an immense problem. But communication within the project also suffers: other teams of documentors, implementors, testers and maintainers remain essentially ignorant about the system. The only obvious cure is to move knowledge with the people: let some developers work with the designers, testers with implementors, etc, to help individuals feel at home in more than one 'subculture'.

This people transfer often seems the only viable method of coping with the even more difficult situation of geographically dispersed projects. Given the present situation of lack of adequate design representation methodology, we must move people who can solve problems and fix the system if necessary, even if unable to explain it to an 'outsider'. It is therefore important to have staff who are willing to travel on short notice. In cases of international operations familiarity with foreign languages and customs is also an asset, particularly if combined with planned rotational assignments.

In a rapidly changing, high technology field, there is perhaps never enough time to decode, formalise and translate the ad hoc jargon developing locally; we have no choice but to live in this Tower of Babel and transmit knowledge not from skull to skull but by infusion into the team, and mixing the very people who develop both product and jargon.

CHAPTER 14

THE CHARACTERISTICS OF LARGE SYSTEMS*

1 Scenario: The Nature of Largeness

In his survey paper on software engineering [BOE76], Barry Boehm observes that "... as we continue to automate many of the processes which control our lifestyle - medical equipment, air traffic control, defense systems, personnel records, bank accounts - we continue to trust more and more in the reliable functioning of this proliferating mass of software".

This very brief statement summarises the intrinsic environmental circumstances that have given rise to the large-program phenomenon and the associated software crisis. Mankind today, as individuals, as nations, as a society places more and more reliance on the mechanisation, using computers, of an increasing variety of applications. The latter interface with, control and are controlled by, ever more complex human organisations and activities; and all interact with one another within the operational environments, often in an unpredictable manner [LEH76]. The computing mechanisms are embodied in an increasing variety of equipment of ever greater power and speed. The resultant complexes of machines and their application-oriented and system-oriented programs or software, are conceived, created and maintained by people increasingly remote from the application, from the operational environment, from the mathematical and programming skills demanded of early practitioners, and from the management skills required by the controllers of human activity in the days before automation.

In the early days of computers a programmer, usually a mathematician, scientist or engineer, was presented with a problem. He was able to identify algorithm(s) for its solution. The details of the program subsequently written would depend on his choice of algorithm, on his skill as an analyst and programmer and on the particular set of constraints arising out of the environment in which the program was written and out of that in which it was subsequently to be executed.

The application developer might recognise that in certain circumstances the preferred solution and its program embodiment would fail; would produce a result that was at best less than optimum, at worst incorrect. Failing such programmer perception one would expect that, at best, the machine would detect the circumstances in execution (for example an out-of-bounds number). At worst the computation might complete and the error would be detected subsequently, with consequences that could range from the inconsequential to the very costly. Whatever the case the exceptional was taken care of by human intervention.

With one problem solved the individual or group pursuing some responsibility would encounter other areas ripe for computerisation. Thus in appropriate instances (and sometimes in not so appropriate instances) programs would be written with even more of the overall activity becoming computer based. But the human remained as the link between the separate computations. Still other humans were responsible for administrative tasks such as scheduling the various runs, the allocation of computing and other resources to successive applications.

All those involved in the processes described above soon realised the potential for expansion through *encapsulation*; the *binding* of the separate activities into a single larger program. The potential benefit was clear: *less* human effort (man is a lazy animal); *increased speed* and *cost-effectiveness* through the elimination of human intervention which must inevitably involve loss of machine resources; increased reliability (sic) of the machine and of machine processes. So why not let the program take care of all exceptions; why not let the program recognise and sequence the succession of activities; why not let the computer handle the administrative problems of language transformation, resource scheduling and allocation, information storage, communication? Encapsulate as much as possible within a single program structure. Create comprehensive programs. Add bells and whistles. And so it was done. More and more was included. The *large program* had arrived.

The adjective large as used here, the concept and attribute of *largeness* that we now develop and characterise, is not intended to reflect the number of instructions or modules comprising a program. Nor do we refer to the size of its documentation, or to the program's resource demand during execution. We do not even intend to emphasize the wealth of function contained within it. There is always a level of

description at which the function is recognised as an entity, a payroll program, an operating system. The amount of functionality is relative to a level of discourse.

All the above indicators of program size can be expected to increase as a program grows larger in the sense to be described, but the root cause of the characteristics we shall identify is related to the concept *variety*. A program is large if its code is so varied, so all-embracing that the execution sequence may adapt itself to the potential variety of its operational environment: the specific input, the requested output, and the environment during execution. A program *is* large if it reflects within itself a variety of human interests and activities. And if it does then it will essentially lie beyond the intellectual grasp of a single *individual*. It will require an organised *group* of people to design, implement, maintain and enhance it. And it is the communication between the variety of activities implemented in the program, the communication within the implementing organisation, the communication between the implementors and their product and finally the communication between all these and the operational environment that lead to the emergence of the *largeness* characteristics which we discuss in much of the remainder of this chapter.

2 Phenomenology: Measurement in Software Engineering

The preceding section has related largeness to variety, the degree of largeness to the amount of variety. The variety is that of *needs* and *activities* in the real world and their reflection in a program. But the real world is continuously changing. It is evolving. So too are therefore the needs and activities of society. Thus large programs, like all complex systems, must continuously evolve. Alternatively, they can only fall into obsolescence and uselessness [BEL76], [LEH74].

We discuss the continuous evolution of large programs, perhaps the most fundamental of their characteristics, in a later section. It is introduced here to provide a focus for the data and data interpretation that is first presented to demonstrate that our discussion represents reality and not abstract philosophical musings that have little relevance in the hard-nosed world of applied software engineering.

Moreover, we hope to convincingly demonstrate that the identified characteristics are intrinsic to the use of computers. If this is accepted, two important conclusions

follow: firstly, until it can be changed, we must accept the world - in this case the programming environment - as it *is* and not treat it as we would *like it to be*. Limitations that arise from characteristics we do not fully understand, far less control, must be accepted unless and until they can be changed. Secondly, we can only hope to change and fundamentally improve the software engineering environment - the world we work in and the products we create and maintain - when it is *understood*; when its characteristics *and* the causes or mechanisms that underlie them are identified.

This problem of system understanding and mastery is not new. All of the natural sciences have been built and continue to develop on the basis of a common methodology. The universe or system of interest is observed. Gross entities, patterns of behaviour, are recognised and global measurements made, until regularities, patterns, trends, invariances are observed. Only then are models and supporting hypotheses created. These in turn form the starting point for a developing theory that relies on prediction, experimentation and further observation for the gradual evolution of the theory. In parallel, there will emerge an experimental and applied science which, in response to societal needs and efforts, leads to an engineering technology.

That is, the initial development of any science is phenomenology-based. It is not in the first place built, as is mathematics, on abstract concepts, axioms, that are gradually developed into a total structure of models that pass tests of reasonableness and elegance. A formal framework and axiomatic theory follow when basics are clear, when it is known what is fundamental or critical, and what is fortuitous. Indeed, even mathematics itself has developed from observation of relations in the real world. Thus the study of software engineering too can benefit from phenomenological studies. The topic has arisen because of bitter experience in developing and maintaining *large systems*. Hence, we are concerned about a more precise characterisation of large systems. We must begin by providing some initial data that can set the scene.

3 Some Data: Traditional Indicators

A first indication of the magnitude of the phenomenon may be obtained from data and forecasts on programming expenditure and the programmer population. Table 1 presents a fairly recent *projection* of trends in the software industry [SLT]. It projects an expenditure growth by a factor of two every

Table 1: Projected Programming Expenditures

DP INDUSTRY GROWTH

	USA ¹	% of GNP	World ^{1,2}	% of GWP ²
1970	21	2	28	.9
1975	41	3	56	1.4
1980	82	5	111	2.2
1985	164	8	223	3.5
1990	328	13	445	5.6
1995	656(?)	21(?)	890(?)	8.8(?)

¹ 1970 US dollars in billions

² Understated because Eastern Europe and USSR not included

Table 2: Projected Programmer Population

US COMPUTER & PROGRAMMER CENSUS

	Computers	Programmers	P/C ¹
1955	1,000(?)	10,000(?)	10(?)
1960	5,400	30,000	5.6
1965	23,000	80,000	3.5
1970	70,000	175,000	2.5
1975	175,000	320,000	1.8
1980	275,000	480,000	1.74
1985	375,000	640,000	1.71

five years from 2% of the United States GNP in 1970 to over 20% by 1995. Table II from the same source indicates the expected growth in programmer population. Note the implied decline in the number of programmers per installed computer as indicated in column 4. We question however whether the projection really takes into account the proliferation of microcomputers or the large program characteristics that form the theme of this chapter. Thus the actual growth of the programmer population may well be larger than that projected in the table. In any event, the magnitude of the educational and organisational problems in the management of programming projects arising from even the indicated growth is clear.

We have been unable to uncover statistics that indicate how expenditure and programming effort have been divided up between small individual programs - whether application or system - and what we shall classify as large programs or program systems.

Table III provides data for a series of systems that are typical of their respective functional areas. The reader will be well aware from his own experience that this small list of 'large' programs could be multiplied many hundreds of times. For each system we give a size measure in statements (instructions plus comments) for one release where each release corresponds to a version of the system as it is made available to the end-user community. The age of the system at release time is measured from first release to the end-users.

Manpower data is notoriously difficult to obtain. Moreover, data definitions and mode of collection differ from organisation to organisation. Yet it seems desirable to provide some indicator of the effort that goes into software development and maintenance [BEL71b]. We found on several systems data pertinent to the number of modules modified between consecutive releases. This number divided by the length (measured in days) of the inter-release interval yields then a convenient normalised measure of effort: the modules handled per day. In earlier publications [BEL76], [LEH77], we have shown how maintenance effort remained constant at about 11 modules per day handled, over the life time of the IBM OS/360 (370) operating system. The corresponding figure for DOS (also constant) was about six. A major military stock control system, intermediate in size between OS/360 and DOS/360, for which we have recently been able to study data, experienced a constant module handle rate of about eight per day. For another manufacturer's OMEGA

System	Release #	System age (years)	# (Statements) $\times 10^3$	# (Modules)	Language Used
OS 360	21	6.5	3460	6300	Assembly
DOS 360	27	6.0	~900	2300	Assembly
A Banking System	10	3.0	45	-	Algol
Electronic Switching System SPI	20.4	4.0	178	-	Assembly
Electronic Switching System SPC-X3	18	3.0	212	-	Assembly
Building Society Accounting	*	8	150	800	Assembly

Table III: Size indicators of different software systems.

* In this case, there is only one installation serving 80 users. The release concept is not applied and instead changes are incorporated as developed or tested. About 150 have been incorporated in the system per year over its lifetime.

operating system a rate of about 0.8 modules (of a different size) handled per day was observed over a period of four years. Finally, in the banking application system of Table III the rate of making changes appears to have been constant at about 0.75 changes per day over a three-year period. We hypothesise that this essentially stable work-input rate, which in each instance appears not to have changed despite improvements in languages and methodologies used and changes in resources applied, will be found to be an almost universal feature of the programming environment.

With this observation it becomes clear why productivity is so difficult to define or measure in software engineering. It does not represent a meaningful, controllable parameter in the classical, industrial sense, but is determined by global system and environmental properties that, at present, lie outside our experience, understanding, or control. Nevertheless, in Table IV we provide an illustration of the programming rates achieved. The variability as a function of program type is also well illustrated by the data which shows that the programming rate for the structured and relatively simple language processor is some four times as high as it is for the much more complex control programs. We do not here attempt to analyse this data further. Clearly, the methodology of global observations we have outlined in the previous section must be applied systematically over a wide data space to achieve understanding and meaning in the definition, measure and prediction of programmer productivity.

More *varied* data about a collection of independent programs developed in a large software house is given in Table V. We draw particular attention now to the large volume of documentation and to the varied ratio of pages of documentation per kiloline of code. A similar variability is found in the size of the project as measured by the average number of personnel, and in project duration. The table thus illustrates the difficulty of making general statements about any aspect of programs and the programming process. This impression is reinforced by Table VI which shows the ranges of some project and program parameters for the products of a different software organisation involved in contract programming over a period of several years.

The present section has concentrated on providing raw data that is intended to give the reader a feel for the numbers that arise when large programs and large programming projects are observed and measured. This data does not

Table IVa: Programming rates observed on different projects

Project	New Instructions $\times 10^6$	Man-months	New Instructions Per Man Mouth	Comments
Apollo Control	1.45	~3800	381	Real Time
Apollo Ground Support	0.53	~1800	294	Simulator
Skylab Control	0.35	~1700	205	Real Time
Skylab Ground Control	1.00	~1100	909	Simulator
A Soft- ware House	up to 0.5	up to 12000	-	from a collection of ~ 40 projects
Electronic Switching System	0.166	2500	66	

Table IVb: Data

	Prog. Units	Number of programmers	Years	Man- years	Program words	Words man-yr.
Operational	50	83	4	101	52,000	515
Maintenance	36	60	4	81	51,000	630
Compiler	13	9	2 1/4	17	38,000	2230
Translator (Data assembler)	15	13	2 1/2	11	25,000	2270

Data from *Bell Labs* indicates productivity differences between problems involving a high degree of variety (the first two are basically control programs with many modules) and those that have better defined specific function. No one is certain how much of the difference is due to complexity, how much to the number of people involved.

Table V: Statistics for Programs developed by a large software house

PRO-GRAM	PRODUCT		RESOURCES TOTAL EFFORT (MM)	AVERAGE !# OF PERSONNEL (#)	DURATION (MONTHS)
	DELIV. CODE (SOURCE LINES)	DELIV. DOCUM. (PAGES)			
1	30000	200	77	6	12
2	11164	350	51	6	8
3	17052	450	46	5	9
4	140000	1900	462	15	31
5	47377	78261	241	19	13
6	229000	6100	1665	46	36
7	401099	138016	1022	42	24
8	712362	44000	2176	77	28
9	58540	7650	723	26	28
10	-	187400	186	18	11
11	80990	6000	527	42	12
12	94000	4670	673	16	42
13	76200	6520	-	-	42
14	18775	2000	199	6	32
15	14390	1200	227	13	17
16	35057	60	71	4	19
17	11122	1000	43	5	8
18	6092	427	47	6	8
19	5342	600	14	3	4
20	12000	3000	60	7	8
21	19000	120	50	6	10
22	25271	4500	169	15	12
23	20000	2000	106	8	14
24	12000	1000	57	6	9
25	7000	2000	195	21	9
26	13545	2021	112	7	17
27	14779	400	67	10	7
28	30000	3800	1107	16	68
29	69200	9700	852	24	35
30	486834	41000	11758	174	67
31	220999	15900	2440	40	61
32	57484	8000	-	-	19
33	128330	20880	673	67	10
34	32026	400	136	4	36
35	15363	700	37	5	7
36	4747	200	10	3	3
37	99000	8800	-	-	47

Table VI: Variation for Program statistics within a software organization

<u>Type of Data</u>	<u>Range</u>
(1) Number of Machine Language Instructions	15,000 to 3,600,000
(2) Number of Cumulative Trouble Reports	1 to 1685
(3) Number of Releases	1 to 7
(4) Time Span of Releases	6 to 80 Months From First to Last Release
(5) Average Error Rate: Errors Per Month Per 1,000 Instructions	.016 to .276
(6) Length of Time in Test Mode	1 to 5 Months Per Release
(7) Duration of Use Per Release	1 to 31 Months
(8) Percent of Compiled Code	0 to 100
(9) Percent of Assembler Code	100 to 0
(10) Percent of Code Increase/Decrease From Release to Release	-27 to +67
(11) Percentage of System Disabling Errors	0 to 20
(12) Number of Users	1 to over 1,000

appear to provide any general measures of the programming process, or of large system characteristics. We now proceed to analyse more systematically the nature of large programs and of the process by which they are created and, as we shall see, continuously maintained and enhanced.

4 Variety: Change and Growth

There does not, at present, exist a general system theory or design methodology for complex systems. There is in fact some doubt whether a complete theory can ever be totally developed or discovered [LEH76]. Nor are there, at least in the realm of computer software, systematic and complete methodologies for system specification and design. Even with the most meticulous requirements analysis, design and implementation process, the product as first released to its users will not, in general, possess precisely those functional characteristics and properties expected or desired in the application and user environment. The systems will require correction and modification after installation.

Moreover, once installed, the user invariably finds it opportune to use the system differently or for a different purpose than that originally conceived. That is, *use* of the system will suggest functional modification. Meanwhile hardware technology will be developing. Manufacturers are continuously able to develop new or improved processors and devices that offer the opportunity for cost reduction or performance improvements, for greater cost-effectiveness. But exploitation of new usage patterns, new application technologies, new hardware, all require the further modification and development of program support. And once operational the modified hardware/software complex can again not be entirely satisfactory, while once again offering still more opportunities for development. So the programs are again changed and the evolutionary cycle goes on. Continuing evolution, the outcome of the mutual stimulation of system and environment, is an *intrinsic* property of large systems: a property that may be formalised as a *Law of Continuing Change* [BEL76]; [LEH74]; [LEH78]: *A system that is used undergoes continuing change until it is judged more cost effective to freeze and recreate it.*

4.1 Variety Generated by the Desire to Perfect: Continuing Enhancement

The property of continuing evolution is possessed by all complex systems, more particularly all artificial systems

[SIM69] created and manufactured by man. Software systems, however, suffer one attribute that complicates the process and leads to a further property, that of continuing modification of the old. Physical systems implemented in hardware, an automobile, an airplane, an atomic reactor evolve through the emergence of newly constructed entities that are redesigned, hopefully improved, versions of older creations. While attempts may be made to modify an existing artifact for experimental purposes, completion of the redesign process leads to the construction of an entirely new instance. The system, much as biological systems, evolves over successive generations. With software systems on the other hand (and to some extent in socio-economic systems such as cities or a transportation system), it is possible and *appears* more economical, simpler, faster, and in general more expedient, to change and evolve the system gradually through the addition, modification and deletion of code or other system entities. Indeed, it may seem impossible to do otherwise.

Modification *appears* more economical because it requires a smaller immediate capital investment than would recreation. But this assessment is likely to be based on ignorance or inaccurate assessment of total *life cycle* costs. It *appears* simpler because study of a part of a program in its local environment and the paper and pencil (or interactive terminal input) exercise of code modification and augmentation seems to require a relatively small physical and organisational effort. But this is so only if the intellectual (and physical) effort of ensuring *completeness* and *correctness* of the change over the *entire* system and system behavioural spectrum, in *itself* and in relation to *all* other changes being made concurrently or being planned, is not taken into account. And it normally is not; perhaps because we do not know how to or perhaps because we do not rate intellectual effort very highly. It *appears* faster because it is 'obviously' quicker to change 'a few lines of code' than to re-create an entire system or subsystem in which a major fiscal, temporal and human investment has already been made.

But basic appearances are fallacious. The fallacy stems from the fact that for any *individual* change (repair, modification or enhancement) these assessments are generally correct. They become tragically wrong when the unending sequence of changes is considered; when the actuality of an evolving usage and maintenance environment is imposed, when it is realised that system structure *must* degenerate and entropy, as a measure of disorder, increase under a series of, conceptually mostly unconnected, changes.

4.2 Variety Generated by Imperfection: Continuing Maintenance

The preceding sections have discussed the continuing evolution, functional and performance-wise, that a software system undergoes. Definition of system requirements, development of a specification, design and creation of code that implements that design are all human, intellectual activities not yet subject to the rigor of mathematical analysis, physical laws or the accumulated, ad hoc and pragmatic, but nevertheless definitive, guidelines of engineering practice. Thus the emerging product must inevitably contain faults, design bugs as well as implementation errors. It must, therefore, be validated, either on completion, or *repeatedly throughout the entire process*, (14.1) so that faults may be detected and corrected.

Ideally such validation should be based on constructive proofs that guide the design process [DIJ72] or on a proof of the identity (in some sense) of the output of each stage of the total process with that of its predecessor stage [HOA69]. However, at the present time applicable techniques have only been developed for relatively simple, self-contained, programs. Extension of such techniques to large multi-function, multi-element systems is, at best, likely to be a slow process.

For the foreseeable future, therefore, validation must continue to rely on inspection [FAG764] and on testing. Effectiveness of the former, however formalised, depends heavily on the system *overview*, observation and *understanding* of the inspectors. The effectiveness of the latter will depend on the insight and understanding of the test designers who cannot possibly view the system from all future user perspectives. Moreover, the test designers must cope with a changing, combinatorially large, set of program boundary and environmental conditions, and hence with an impossibly large number of system states and execution trajectories. System validation activity based on these techniques can therefore never expect to locate *all* faults, can in fact not possibly demonstrate that the system is faultless [DIJ72b]. In summary, both inspection and testing are likely to be useful in ensuring elements, modules and components, that are

14.1 (Eds) 1985 *italics but used here in the sense that it is now referred to as verification. The concept of validation through prototyping and similar techniques is clearly absent from the remainder of this section.*

relatively clear of localised faults. They become increasingly costly and ineffective in the search for problems stemming from global interactions and dependencies; in ensuring correct *system* operation.

The faults that are discovered before the product is declared ready for customer delivery will generally be fixed immediately. However, particularly for multi-site, multi-configuration systems, users will, after release, subject the integrated system to configurations and execution patterns to which it has not previously been exposed. Thus new faults will inevitably be discovered and will continue to require fixing.

And the cost of this continuing maintenance is high: Figure 1 summarises the fraction of programming effort spent in maintenance for a large sample of installations [GOL73] around 1970. An elaborate organisation is often required to implement the system change activity. We discuss this in the next section.

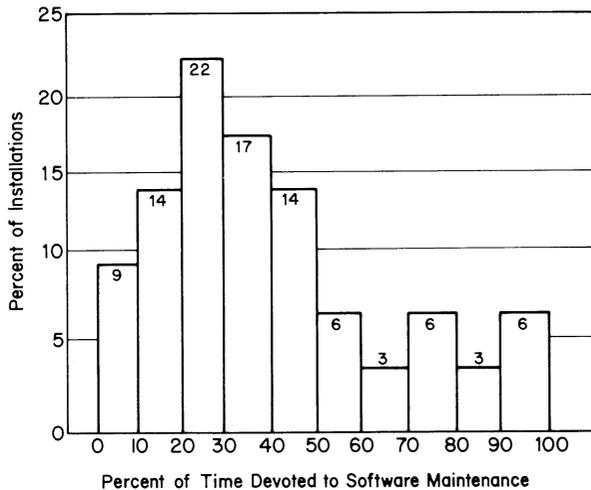


Figure 1: Percent of Time Devoted to Software Maintenance

We note that this so-called fix or repair activity is not really repair at all. Hardware physically deteriorates because of wear, corrosion or fatigue. Its repair consists of *replacement* of one or more components to *restore* the system to its original state. The elimination of software malfunction, on the other hand, requires a change *away* from its designed or constructed state. Software repair and maintenance mostly involves *redesign* which in turn may introduce further *error and* is very likely to further increase complexity. For the emphasis of a maintenance team will be on speed, on cost minimisation, or just simply on obtaining a correct fix. It will not generally include structural maintenance or improvement. And if imperfect repair or structural deterioration is likely when a single fault is fixed, the effect is likely to be compounded when several faults must be cured in the same period possibly by different groups or individuals; possibly concurrently with enhancement and development. Thus, inevitably, repair activity will be imperfect, will cause the creation of new problems.

4.3 The Result of Continued Evolution: Structural Complexity

Some programs, or parts of the same program system, may be more complex than others, as discussed in Section 3. What is important is that increasing system complexity leads to a regenerative, highly non-linear, increase in the effort and cost of system maintenance and also limits ultimate system growth [BEL71b]; [LEH74]; [LEH76b]: The trend may be summarised in a *Law of Increasing Unstructuredness* (Entropy) [BEL76]; [LEH74]; [LEH78]: *The entropy of a system increases with time, unless specific work is executed to maintain to reduce it.* This, our second law, is analogous to, perhaps even an instance of, the second law of thermodynamics.

For our present purpose its significance is not that complexity increases with age. That is universal experience. What is fundamental to achievement of better software management and minimal life-cycle costs is the *recognition* that complexity grows *unless and until* effort is invested in restructuring. Some part of one's resources *must* be invested in restructuring periodically or continuously. The alternative is to reach such a level of complexity that further evolutionary progress can only be made through re-creation; total abandonment of the system and its replacement by a new system structured, redesigned and implemented to

System	Src.	Type	Language	Purpose	Size Inst.	Users	Hdwr.	Configuration	Impl. Env.	Mang. Ctrl.	Age* Days	Rel. Seq. No.
Ome-ga	Manu-facturer	Transaction O S	Assembly	Limited	?	25	Var.	Var.	Prg. Centre	Conc.	1000	10
Bank Syst.	Bank	DB syst	Algol	Single	45k (H.L.)	1	Fxd.	Single	One Group	Unifd.	1000	11
OS/-360 VS2	Manu-facturer	O S	Assembly	Universal	>2M (L.L.)	>1000	Range	Mult.	Distr.	Distr.	3500	23

*Age refers to period over which data is available. the first two systems have a prehistory.

Table VII: Three large systems whose evolution has been studied

satisfy the most recent operational requirements. In fact, the technological aim must be to achieve the most economic balance between continuous or periodic restructuring and periodic recreation.

Clearly then it is not sufficient for a system to be initially correct. It must remain correct under an unending sequence of changes. To achieve and demonstrate continuing correctness, it is not sufficient that the system be initially well structured. Well structuredness must be maintained despite that sequence of never-ending change. And our studies of analytic models of the programming process [BEL71b]; [BEL76]; [LEH74]; [LEH76b] indicate that structural maintenance as the system grows is likely to require an ever-increasing proportion of maintenance resources. Thus maintenance must ultimately become uneconomical, making reimplementation of the system inevitable. Is this point predictable? Is it possible to forecast the point in time at which recreation of the system is required, far enough in advance to ensure completion of the new system before the old one has collapsed, has become unmaintainable?

4.4 An Empirical Study: The Dynamics of Evolution

Rather surprisingly, since every aspect of system development, implementation and maintenance is to some degree planned, and is managed by people for people, the growth patterns of a system as measured by various critical parameters are statistically predictable. This general property is reflected in a third law [BEL76]; [LEH74]; [LEH78]: *Measures of global system attributes may appear stochastic locally in time and space, but statistically are cyclically self-regulating, with identifiable invariances and recognisable long-range trends.*

Recent studies [BEL76]; [LEH76b]; [LEH77] have reported on the gross growth patterns of a number of very different software systems. Table VII lists some of these differences. The studies demonstrate quantitatively what participants in such projects have long known heuristically, that there is a continuous growth in the functional content, the size, the need for repair and the complexity of each system. Figures 2 show the growth of two systems, measured in number of statements and modules, respectively. Notice the increase in the size trends, and the declining growth rates.

From the same studies, Figures 3 capture the work rates during evolution. The measures, changes made and modules

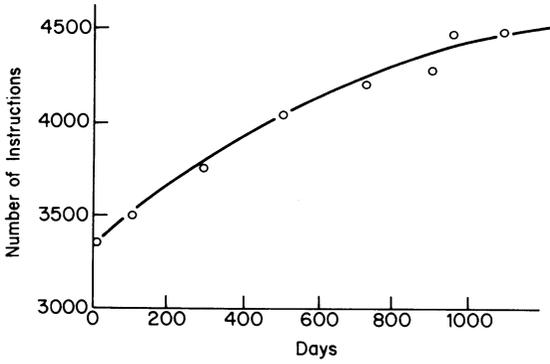


Figure 2a: Growth of the banking system

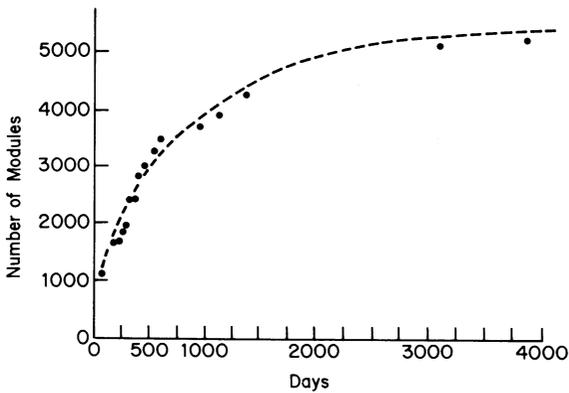


Figure 2b: Growth of OS 360/370

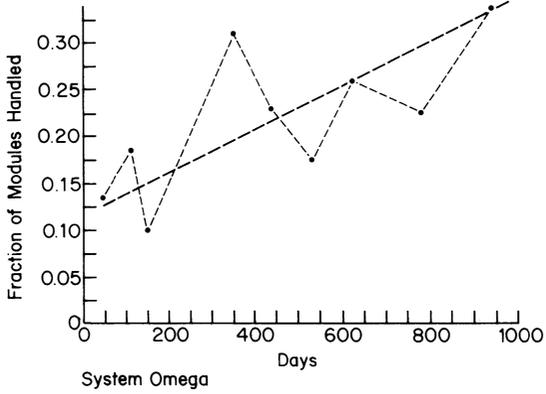


Figure 3a

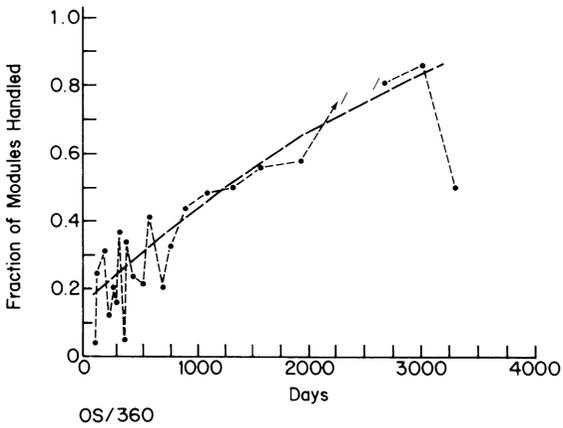


Figure 3b

handled are plotted cumulatively as functions of system age to eliminate the effects of release overlap. The slopes of these plots are effectively and unexpectedly constant, implying a constant work rate despite improving methodology, tools and changing resources. The invariance of work input reflects the stabilising influence of the many organisational feedback loops controlling system evolution.

An approach to the measurement of complexity is by the fraction of total units (ie modules) impacted by some unit change: the more units impacted the more diffused the change, the more complex the system. Conversely, in a well structured system changes tend to remain localised. Figure 4 indicates the trend for two systems studied: increasing spread of changes as the systems age.

An aspect of the interaction between the evolving system and the human organisation which drive this evolution is depicted in Figure 5. This shows the linear growth trend of two systems as a function of release sequence numbers. Closer examination shows a cyclic subpattern of increasing period. This cyclicity is the manifestation of the conflict that arises in large system management, between the pressures for an increasing enhancement rate (positive feedback) and the resultant increasing resistance to change, increasing difficulty of the work, as structure and quality, organisational integrity and knowledge decline (negative feedback).

We deduce that an organisational, as distinct from an individual, programming project behaves like a self-stabilising feedback system. This is of course quite contrary to the view that managers have. They see it as a process whose progress is determined by local and global decisions as these are made. Theoretical models of the process [BEL76], [RIO76] suggest that the observed behaviour is consistent with our developing view and understanding of the process as a complex, feedback-system-like activity.

That is, gross, historical, software project data of a variety of systems and project attributes can be used to generate project models that represent or measure the evolution process. The models provide statistical invariances, patterns and trends that are interpretable in terms of theoretical models of the programming process. All that may be used directly to better plan and control system development, maintenance and enhancement. Equally, their study leads to a deeper understanding of the nature and

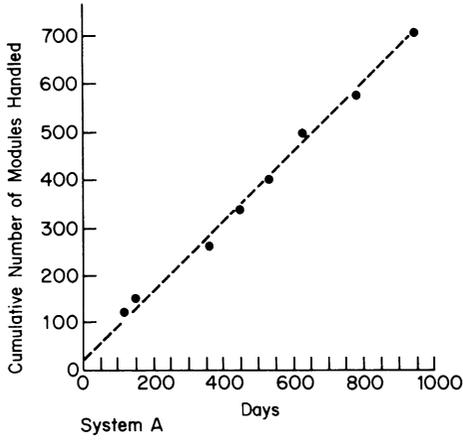


Figure 4a

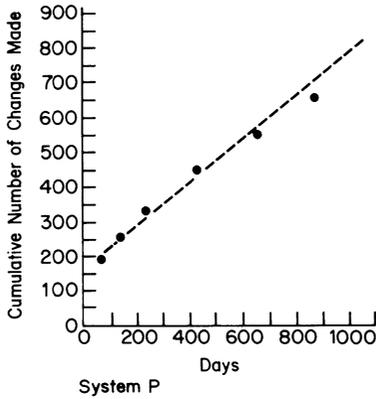


Figure 4b

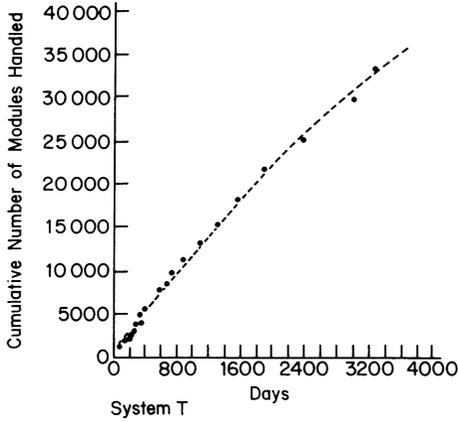


Figure 4c

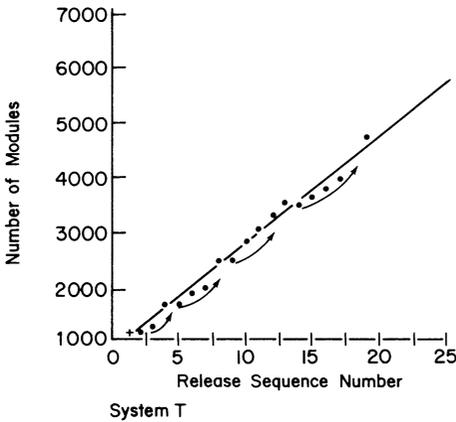


Figure 5

attributes of the programming process and of the systems that the process produces. The increasing understanding can be applied to direct and guide software engineering practice and the programming process so that the latter may yield higher quality and improved life-cycle properties for programming systems [LEH78].

4.5 The Life-Cycle Cost Pattern

The previous discussion has been largely system and programming oriented, directly addressing software system attributes. Quantitative global cost studies have also been undertaken and have produced useful models. These reflect the fact that a very high percentage (50% - 90%) of life cycle costs of a large software system may be incurred in post-first-release maintenance and enhancement [ARO74], [BOE76], [PUT76]. This data provides dramatic confirmation of the reality of the phenomena discussed above. It indicates that the initial assessment of large software systems, the decision to implement or not, must be based on an accurate projection of *life-cycle costs* not on the estimated cost of development and first implementation. But that in itself is a problem since the expenditure pattern is certainly very non-linear, with shape parameters being a (not well understood) function of the requirements of the system, the implementation and the usage environments. But in this area too, global system observation [PUT76] leads to models that can be most effectively applied in the planning and control environment. Their further development should lead to specifiable and controllable life-cycle characteristics.

5 Large Systems: Complex Interactions

In the preceding section we have concentrated on a description of the global macro-properties of large systems. As the thermodynamicist does in physics or the macro-economist in economics, we have summarised observations, measurement and interpretation of the phenomenology of evolution of the total system and the programming process in their development and maintenance environment.

We have, however, only briefly indicated how large a system must be before it can be expected to display characteristics as described. A precise measure is likely to depend at least on the nature and structure of the organisation controlling the system, the programming methodology employed, and the nature of the usage environment. But the general indicator mentioned in Section 1 can be deduced from a clearer

understanding of the underlying causes of the phenomena described. The basic problem is understanding; understanding the environmental requirements so that the system can be specified, designed and implemented; understanding the design and implementation so that the system may be validated, proven to provide the requirements, all the requirements and preferably nothing but the requirements; understanding the system's properties, capabilities and limitations so that it may be learned and effectively used; understanding the system structure and content so that it may be maintained and enhanced.

5.1 Program Systems

The degree of understandability of a system depends on its structure and its content, the internal interconnectivity between its parts; on its complexity.

More specifically, structure represents the degree to which, and the way in which, the system may be viewed as a set of interconnected subsystems, the way in which the subsystems themselves may be decomposed, and so on. Decomposition aids understanding to the extent by which it enables system behaviour to be understood in terms of, or deduced from, the behaviour of its constituent parts. The degree to which this is possible depends on the regularity of the structure and the interpretability of the parts as well-defined primitive operators or transformers. Furthermore, the comprehensibility of the system will be heavily dependent on the actual or implied interconnectivity, interaction and dependence between parts, particularly where the related lines of communication deviate from the regular system structure. That is, the understandability and therefore the complexity of even the most well-structured system will depend additionally on the nature and extent of internal, ie intra-system communication.

We may now restrict ourselves to such programs where the structure and internal communication links or dependencies are sufficiently rational to make the system understandable and manageable. As requirements grow and ever larger programs are considered, the point will come where it is judged necessary to employ two (or more) people for program definition, design, implementation and validation. The program is too large to fall fully within the intellectual grasp of a single individual. This might well be the critical characteristic that identifies a large system. The new factor that arises at that stage is of course the need

for *human communication*, intraprocess communication between two (or more) people. And this is a process whose effectiveness will deteriorate rapidly as the number of communicants increases.

A further quantum jump in the complexity of human communication, in the probability of distortion and error, occurs when the number of communicants reaches say eight, when the need for at least two levels of management first emerges. The nature and degree of communication between the members of each of the individual groups is then radically different from that between members of different groups. Comprehension of the total system has certainly slipped from the grasp of any one individual or of an informal team. The system will rapidly become a large system with all its characteristics and problems.

5.2 Program Collections

The preceding discussion has outlined the development of a single program into a large system. Such large systems are met today in the form of operating systems, transaction systems, weapons systems and so on. The components of such systems make frequent use of each other's services and data and exploiting what are otherwise independent hardware apparatus. Under certain circumstances, a set of programs may develop into a large *collection* of programs. The structure of such a collection, a set of alternative compilers or a number of independent application programs, may be depicted by a wide-span, two-level hierarchy in which a single calling element, a scheduler for example, selects for execution one of a number of alternative programs. Such a collection typically serves a common purpose yet the parts do not, in general, communicate directly with each other while in execution on a machine. In the sense that the word 'system' is used, such a collection of non-interacting programs does not form a system. During the development and maintenance phases, loose coupling may exist in the form of decisions about the placement of some sub-capability in one or other program or in a new independent program. But during execution, coupling arises only from the calling sequence.

It may, of course, be that this collection of programs is sharing, in some sense, a common data base. In that case the structure and internal dependencies of the latter will act as the communications linkage that causes evolution and degeneration. That is, the total collection of programs *with* the data base out of which and on which they operate will now

form a system. And that system may be expected to demonstrate all the system-like characteristics discussed (14.2).

6 The Software Process: Knowledge, Skill and Communication

The most natural fashion by which groups of people collaborate in a programming project [BEL80] is through decomposition of the total product into separately designable and implementable components [MYE75]; [PAR72]; [STE74]. To be considered a system, such components must nevertheless interact and communicate. Ideally, the details of protocol, paths, structure and content will be agreed by the designers. In practice, their decisions will be modified during the development process. Additional linkages may be introduced through unintended, often unperceived, side effects. 'The evil that these do live after them' [Julius Caesar, Shakespeare]. In any even the agreement reached by the collaborators as to the exact details of the interface between their components must be faithfully recorded and strictly adhered to. Subsequent modification must similarly be agreed and recorded.

That is, documentation must be created and updated continuously to record system features, individual design and implementation decisions, the considerations on which they were based, and the details of interfaces between individual system elements. The documentation, ideally a faithful record of the entire process and product, of all internal communication, itself provides a communication link between all process participants and between them and system users. More generally it should provide a permanent, accessible, complete and correct record of innumerable, transient yet possibly significant interhuman communications. In practice it is of course rare that any of this is done completely and correctly.

6.1 Specialisation of Human Activities

At some stage of the process the components, the parts of the system created by separate groups, or by the same group at different times, must be linked together, integrated. As already remarked, no technology yet exists that will *guarantee* correct functioning of the resultant system. Hence

14.2 (Eds) *A prediction subsequently confirmed by our study of a major stock control system as described, for example in [].*

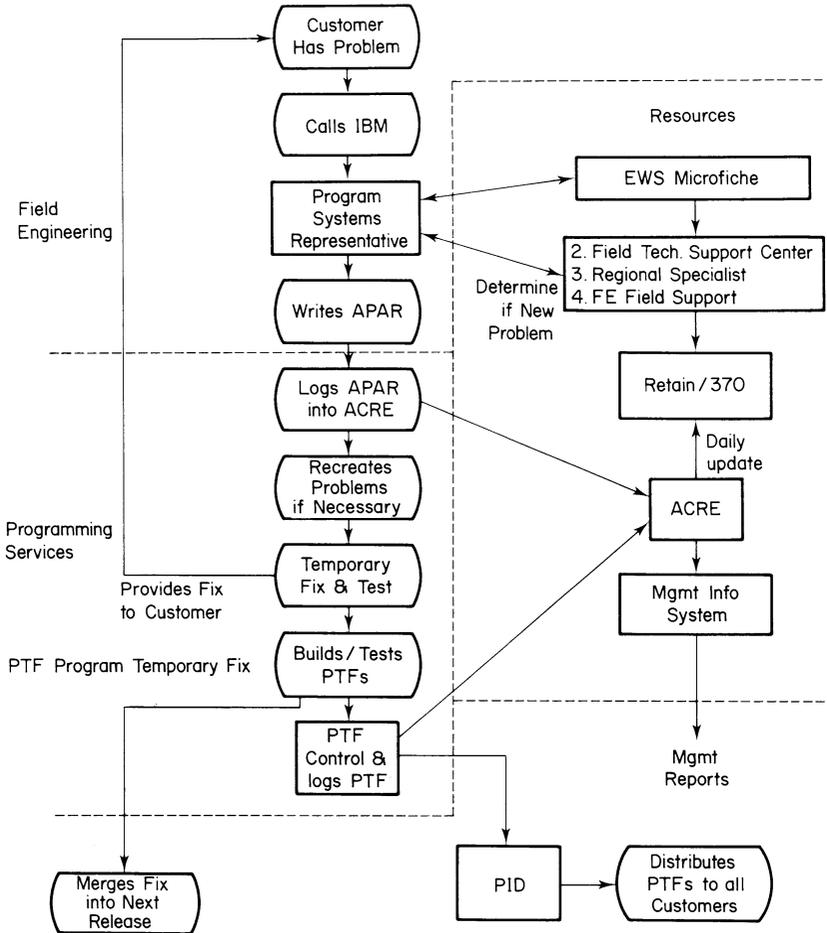


Figure 6 The Software Services Process

the newly assembled system must be tested. The test responsibility will be delegated to one or another of the original groups, to a new group created out of the original groups, or to an entirely new, perhaps specialist, group. The process changes once again. From being in the domain of a single individual or group from start to finish the process transforms into a sequential, assembly-line-like activity.

Of course in the very small 'large' program the analogy is hardly relevant. But when an organisation develops to any size, when systems become really large, expensive and therefore long-lived, division of labour, specialisation and the programming analogy of the industrial assembly line may, superficially at least, appear as the most cost effective design and manufacturing process. Architecture, design, programming and coding, data base management, documentation, component tests, integration and system test, quality assurance, each activity becomes the domain of a group of specialists. A variety of system elements pass sequentially among them. New programmers, taken on as the more experienced are promoted, must gain knowledge of the system and experience of the process. What better way can there be of gaining that knowledge and experience than assigning to them responsibility for fault fixing, clearing up the many problems reported by users and project teams alike from day to day?

6.2 Product vs Process Knowledge (14.3)

There is unfortunately a fundamental fallacy in this approach. The assembly line can work where local processes do not require knowledge of the total product, and the entire process; when individual activities and parts can be totally specified and described, so that elementary operations are essentially independent. Above all a successful line operation requires that total product quality is the summation of the quality with which the individual operations of the process have been performed.

In software, process knowledge relates to methodologies, techniques and tools for specifying, designing, coding, testing, and integration of programs, as well as to the planning and management of these activities. Product knowledge relates to the understanding of program elements, structure, algorithms and operation, individually and

14.3 (*Eds*) See previous chapter [BEL78] for more complete discussion.

collectively, and their interactions. It demands a constant awareness of the major objective and requirements, of the multitude of program interactions that occur through common use of program names of objects, such as procedures, tables, labels, variables, and so on.

Product and process knowledge and awareness can be obtained by the individual only from the documentation and by word of mouth. Total assimilation is impossible. Hence the assembly line approach cannot be fully effective; is in fact highly fallible. Since no individual can have total knowledge or comprehension, errors are unavoidable and some must remain undiscovered till the product is in regular use possibly for all time.

Assigning repair responsibility to the 'greenhorns' is of course the greatest fallacy of all. They cannot, and cannot be expected to have assimilated, the product knowledge, or for that matter the process knowledge, that is essential for effective structural maintenance, performance maintenance, and functional modification. Clearly repair activity should be the responsibility of those with the maximum system overview and insight. But these are generally the most experienced, the most senior, the highest paid individuals. As such they will often have been promoted into, or out of reach of, the project management. Even if still within the project, they will have architectural or design responsibility, will be working on more advanced system elements, will probably have forgotten much of the detail required for repair.

6.3 The Part-Number Explosion

An additional dimension of complexity in repair arises as follows: consider modules as basic building blocks, perhaps thousands in number. In a release scheme, there is at any given time only one single valid version for each module. In a multi-installation system, however, repairs (fixes) will often be rejected by users who decide that they are not affected by the fault being fixed. The result is that *multiple* and valid versions of the same generic module evolve; the most recent version, as well as one or more predecessors which will still be actively used at some sites. The predecessors cannot be invalidated to simplify documentation and bookkeeping, since they still exist in the environment against which further errors could be reported and their repair requested [HOA69].

6.4 Documentation

The key to system control is system comprehension. One cannot hope to understand the purpose, the mode of functioning and the details of operation of a *software* system by visual inspection alone, though this might well be possible in a hardware system. One certainly cannot expect understanding of software systems to be discernible from the exquisite level of detail represented by present-day machine level or micro-code. Comprehension of the system and its parts requires knowledge of total system objectives, of their partitioning into individual capabilities or function and of their mapping onto a systems structure and its structural elements. Equally, at least minimal explanation is required of the algorithms used in implementing the system and its subsystems at various levels.

Clearly then a large software system *must* be accompanied by documentation. Moreover, the documentation must be readily *accessible* according to the particular needs or interests of the inquirer. And the documentation must keep pace with the changing system, remaining correct and complete.

In practice, of course, this is very difficult. The concept of 'self-documentation' of high level programming languages is important but it is insufficient. For machine level languages it does not apply. Thus documentation is necessarily an activity that runs parallel to, and must be interwoven with, the design, system implementation and maintenance activities. As such there is an inherent problem of coordination. When projects begin to lag and to fall behind schedule, when resources run short, the documentation activity, representing as it does a long-term investment that shows no immediate return, being essentially anti-regressive [BEL71b]; [LEH74] in nature, is amongst the first to fall victim to the inevitable axe. Hence divergence between the system, instructional documentation and descriptive documentation is another very typical characteristic of large systems.

6.5 Communication as the Key to Large System Mastery

The analysis so far has identified 'communications' as the key in determining the pattern of development of system characteristics. It must be considered at several levels. System-internal communication links join its separate parts and make it a system. Understanding of the system and effectiveness of execution are both heavily dependent on the

internal structure as determined by these links. One cannot hope to comprehend the system *as a whole* unless one is aware of the dependencies and interactions, static and dynamic as determined by both explicit and implicit internal communications. And comprehension of the system as a whole is essential to its effective application and its effective maintenance.

Application and maintenance are essentially the domain of people. These are themselves involved in three further levels of communication between the people that jointly collaborate to build and maintain the system. There is also the communication between them and the operators and users of the system. Finally, there is the communication between all of these and the executive management of the producer organisation. It is, of course, the latter which controls the ultimate fate of the system [COM76], together with that of many other artifacts and activities controlled in support of organisational objectives and making a call on organisational resources.

The resultant flow of documentation and verbal communication is enormous and, in general, not clearly structured. Yet for total mastery of the system it must be integrated and comprehended.

Why is this total comprehension so vital for successful long-range exploitation and control, for continuing control of a system? In general, any interaction with the system, whether for usage or for modification, requires a view of the system as a whole, as an entity. It demands a knowledge of the reaction of the system in its entirety as well as that of each of the parts. The intending user must know and be aware of the *total* consequence of each system access, and of each of the separate individual responses of which that totality is comprised. Even more strikingly, the individual changing the system in any way must tamper with the code at the lowest level of detail, but be fully aware of the global implication of his action over the entire system.

This need for simultaneous awareness, at both the global and the lowest levels of detail, is brought about by the complex and largely invisible structure of system-communication, the totally unforgiving nature of system execution in the presence of logical error or even imprecision, the rapidity with which the system executes and, therefore, the high probability that any fault, any weakness, will be revealed sooner or later. The need is addressed by system structure

and documentation, by system intelligibility, supported by collective human knowledge and understanding of the system.

It is to these areas that we must look for major advances in software engineering, in the development and maintenance of large systems.

6.6 Structure as a Reflection of the Manufacturing Process

Delineation of function and definition of interfaces must occur before autonomously managed groups can begin design and implementation activity. Since the theory of computing system design has not been adequately developed, these definitions and divisions cannot be perfect or complete. In the presence of an evolving environment they cannot remain near perfect or complete. Thus modification must be made as the work progresses, as the emerging system, its function and its structure, is more clearly understood, as the design coalesces and as the system takes shape. Strictly, each such modification should require a total review of all previous decisions. In practice, many modifications appear to be clearly (sic) localisable, to remain within the judgment and domain of a single group. Sometimes they clearly cannot be. But in the interest of cost effectiveness, review and negotiation is then limited to those groups most clearly and most directly involved. Implementation of modifications is largely forced into the constraints of the initial structure.

Similarly, when new requirements are identified and subsequently when responsibility for the implementation of supporting code is assigned, management decisions must be based on the existing structure, on the availability of resources, on the existence of localised product knowledge and process experience. In current industrial practice it is largely based on inter-managerial negotiation and bargaining. System structure cannot constantly be reviewed and redesigned to take cognisance of the new features that may well cut right across existing divisions.

Thus gradually, as the design and implementation proceeds, as the system ages, its structure will not only degenerate. Increasingly the relationship to requirements and functional structure will be obscured. The system structure will begin to reflect the organisational structure and process sequence that created it. And this is, of course, not helpful from any point of view. It cannot make the system more understandable, more maintainable, more fault-free. Nor can it be expected to improve system performance.

Possibly the clearest example of *arbitrary* structural dichotomy is the division of most organisations into hardware and software groups. Ironically, even with the emergence of new technologies, microprogramming as a replacement for earlier hardware logic design has in many instances been seen as belonging to the hardware dominion. Thus the most basic implementation decision, the selection of an implementation technology, the partitioning of a system into its hard, mushy and soft parts is taken in almost ad hoc fashion at the most primitive stage of system definition.

These approaches *may* have been correct in the early days of computing systems. It cannot be correct in today's world. And the greatest sufferers are likely to be system changeability, growth flexibility and system performance.

Perhaps the most important contributions to the solution of these problems has, however, already been made. We refer to the extension and generalisation of the dual concepts of standard I/O interfaces and of channels. Invention of the latter concepts has made possible the individual design, optimisation and system attachment of several generations of new I/O and storage devices. Performance did not suffer because of the incipient potential for device autonomy and system parallelism. We see the generalisation of these concepts in the form of a functional channel or Funnel [LEH77c], as offering the way to a solution to the problems we have identified as discussed briefly in Section 8.

7 System Behaviour: The Optimisation Problem

Like most other artifacts, software systems are developed and enhanced with particular objectives in mind. The objectives are often formalised into optimisation of system attributes, such as function, capability, cost, reliability, security, size, modifiability, etc. All of these attributes cannot be discussed here, so we single out just one quality indicator: performance. In software this is often considered to require an appropriate balance between execution time of a given sub-program which executes a sequence of functions, and the resource usage required to achieve this execution sequence and speed. The following presents some of the difficulties which software performance optimisers must face.

7.1 System Performance - Execution Dynamics

The factors considered so far have been viewed in relation to the programming process, the development and maintenance

dynamics of the large program, its evolution dynamics. When passing to its execution dynamics, its behaviour during execution, not surprisingly we observe the consequences of the same pressures, reflection of these same characteristics.

System performance objectives can also not be precisely specified, implemented and achieved in the first instance, and for precisely the same reasons. After all a program is, by its very nature, a complex system of interacting subprograms, subprograms executing on and interacting with shared hardware resources. Performance is not simply predictable [LEH76] and the desired characteristics must be approached via an iterative modification procedure. The endless growth in function, size and complexity tends to degenerate performance. Hence further changes must be undertaken to maintain it. In fact, in the face of application, user and device evolution, performance characteristics normally need to be *improved*, not just maintained. Thus the execution dynamics of a system, desired and achieved, is a further factor in establishing and maintaining the characteristics we have identified, and itself comprises a further characteristic attribute, *evolving performance*.

7.2 Local and Global Optimisation

The question thus arises as to the extent to which system performance can be optimised, at least relative to requirements and to the environment or at some identified point in time [BR076].

A problem is immediately apparent. Do we optimise performance under the requirements, environmental conditions and system state as they are now? Then by the time optimisation has been completed and implemented, because of continuing evolution, neither system nor environment will be the same. Optimisation will turn out not to have been optimisation after all. On the other hand, one might attempt to forecast the direction and rate of the various evolutionary processes and optimise to some future expected state. Then the problem may well be that because of changing environmental conditions the expected state is never reached. Alternatively, the forecast may become self-fulfilling, achieving optimum performance relative to an anticipated system state that is itself no longer optimum because of unanticipated environmental changes.

The resultant dilemma may appear worse than it is in practice. It is certainly one that may be largely resolved if the optimisers are conscious of the dynamic environment in which they operate. There is, however, a more fundamental constraint on optimisation that, in the present state of software engineering, is far more difficult to overcome.

The essential nature of a software system as a set of highly interacting parts has been repeatedly stressed. The parts interact structurally in that they use common program objects. Certain aspects of optimisation, storage space minimisation for example, dictate maximisation of sharing objects. Other interactions come about dynamically during execution. Procedures use each other's services, they share information, they sequentially share hardware. All of these interactions should be taken into account during optimisation. In practice, however, such global optimisation is very difficult, to say the least. Moreover, designers and programmers have at any given moment an essentially local view of the system in terms of the flow diagram or the code they have in front of them. They have an essentially static view of program flow. A very intensive intellectual effort would be required to convert this to, and assess, the action in terms of the coexistence of concurrent, interacting processes sharing resources. Thus, optimisation will unfortunately often tend to be local. And it is well known that local optimisation almost invariably leads to global sub-optimisation.

There is also a third aspect to the optimisation problem in a multirequirement, multifunction system: it is unlikely that all capabilities can be simultaneously optimised. Requirements are likely to be contradictory and while an acceptable compromise must be reached, true optimisation may not be meaningful. Even where requirements do not intersect functionally it is difficult if not impossible, as a consequence of data or resource sharing for example, to achieve simultaneous optimisation. A data structure that is best for the execution of one function will be suboptimal from the point of view of the other. Optimising with regard to storage usage is very likely to increase time requirements and vice versa.

Summarising then we find that optimisation for large system performance is a delusion, one that is likely to eat up large amounts of human resources if nevertheless pursued too diligently. In its place, a statement of performance expected and required must form an integral part of the

system statement of requirements and of the system specification at its various levels.

8 The Future

The methodological trends stemming from the movements toward structured programming certainly satisfy many of the process and system desiderata arising from the characteristics we have identified for large program systems. The block structure concept first conceived for ALGOL [BAC60], the undesirability of the GOTO construct [DIJ68] and its replacement by sequence control structures that are related to the semantic structure of algorithms, the more general movement to high level languages, the channeling of communication via parameter-passing rather than global variables, the single entry-single exit subroutine or procedure, all these are ultimately directed towards increasing the clarity of code structure, its initial intelligibility and therefore its veracity.

But, while necessary, they are not sufficient. If all the rules are understood and observed in their spirit as well as their letter, good structure, healthy code will have been created. But structure must not only be created, it must also be maintained. Structural maintenance, which is strictly anti-regressive, bringing no immediate return, must nevertheless form an integral objective and part of the maintenance process.

By and large none of the concepts or techniques mentioned above help in structural maintenance under the conditions encountered in the industrial and commercial world. Even if they can be enforced during program development, the pressured development of fixes for field-discovered faults, or software support for new devices, is very likely to lead to their infringement, and hence system pollution, during maintenance activities.

Moreover, as base systems get ever larger, structure will tend in any case to deteriorate more rapidly during maintenance. Thus the level at which the problem needs to be solved is itself rising dramatically. Solutions which may have appeared adequate just a few years ago, now no longer suffice.

One approach to the solution to these problems, program units or components, has been talked about for many years [MCI72] but is only now becoming technically feasible. Moreover, it

is only now that manufacturers and users alike recognise the necessity to *bypass* the problems created by attempts to build assemble large software systems from self-contained software units much as hardware systems are configured out of a variety of 'black box' units. The recent announcement by IBM [IBM76] of selectable software units highlights the practical emergence of this trend (14.4). *De facto* the announcement implies the abandonment of the large, integrated, system.

The basis of the unitised approach is that defined capabilities or functions are implemented, packaged and offered to users as a unit. Each such unit will (in theory) have been totally tested as an entity against its defining specification. Since in practice the specification will not be absolutely complete, it must also have been tested against other units with which it may interface, against some base system which acts as a central connector for numbers of units or against some standardised and complete interface. Thus we elevate the problem of perception, comprehension and control of the large system to a new and higher level. Software units form the primitives of a new universe of discourse. Their *specifications* and *interface definitions* define the semantics and syntax of the design and implementation process. More naively, the process of software system building is then viewed simply as an extension of the standard practice of hardware configuration.

Unfortunately the analogy to hardware configuration breaks down [LEH77c]. The critical differences, those that lead directly to the problem whose solution would effectively overcome most, if not all, of the large system characteristics we have identified, relate directly to the two unit-descriptors referred to above, the *specification* and the *interface definition*.

For the unitised system to be viable, growable and maintainable over a long period the specification of each and every unit must be *complete*, relative to *stated* requirements,

14.4 (Eds) *But has, as forseen below, only provided a temporary respite. The units and collection of units must themselves evolve, be adopted and all the old problems reappear. Now, in 1985, we recognise that only formalisation of the process and its strict control can hope to contain the problem, to permit the continued development, continuous apaptatin and, hence, reliable usage of 'large programs'.*

correct, ie self-compatible, *accessible* and *compatible* with the specifications of a significant subset of other units, and ideally with all of them. Each unit must be 'pluggable', imposing a known 'loading' in terms of its use of system objects. There may be no unidentified side effects, preferably no side effects at all, as a result of unit connection. In Parnas' terms [PAR72] the assumption made by the unit and its environment about others must be completely correct and completely known.

It is one thing to recognise the need for satisfactory specification. It is quite another to achieve it. For the unit concept to work in practice we require a unit specification which is as complete and as accessible as, say, that of a standard bolt or nut. This need was specifically recognised in the emphasis placed at a recent conference on Software Engineering [ICSE2] on requirements analysis. But it goes further. The need is not purely for the identification of requirements. Nor will all problems be solved with the development of the specification language that has been the main objective of specification technology for so long. A specification must have *structure* as well as content. A limited number of candidate structures can perhaps be deduced by identifying system attribute classes and sub-classes, and the relationships between them, that together may constitute the specific character of a piece of software. These classes must then be structured and their alternative interrelations formatted into a very small number of specification skeletons that form possible frameworks into which a specification may be developed. After initial exploration of alternatives, one framework must be selected and a specification developed to fill and cover the entire skeleton. The resultant specification can and should be as complete as the framework. It can then form the definition against which the unit itself is developed, validated, modified - *with simultaneous modification of the specification* - marketed, taught, used and maintained. The structured specification form, of course, is the first step in the development of software engineering than is structured programming.

The need for firm and total identification of each unit interface is also clear. The problems raised are more pragmatic. The problem arises from the fact that there is no apparent physical limit to the size of the interface analogous to the surface area and pin limitations arising in a hardware interface. Software communication is volumetric unless constrained by such rules as that of block structure. Even the latter, and certainly any 'agreed' interface, can be

violated 'merely' by a programmer changing the point of declaration of an object, more generally simply by referencing some system object external to his unit, using simple and direct communication as agreed between two programmers. Thus we face two problems: software interfaces are likely to come in an enormous variety of shapes and sizes; even if successfully established, they are very likely to be violated. It is extremely difficult to control them.

The inherent flexibility of the software interface, plus the sociological and managerial problems of maintaining their integrity in the face of three or more generations of managers and programmers brought up without the concept of an impregnable interface standard has led recently to the suggestion that the interface between software units, at least in any one environment, be standardised and implemented in hardware [LEH77c]. The Funnel concept is based on a generalisation of the channel concept [PAD64] to the point where it is seen as the interface between any two functional units, not just a central processor and an I/O device. It standardises interfaces by restricting and channeling all communications through hardware links, with the definition of each message on each link being contained within a message header according to a standard grammar.

At first sight such an artificial constraint on inter-software unit communication might appear as a crippling penalty that would seriously degenerate system performance. However, we may recognise that the advancing mini-computer and micro-processor technologies make the whole concept not only feasible but even advantageous. Funnels can be implemented in micro-processor form. Equally, each software unit can execute on its own microprocessor. This in turn leads to a potential for parallel execution which means that the performance limitations of standard hardware interfaces are more than overcome. Thus the large software systems of today will gradually evolve into the distributed systems of tomorrow. But we stress again that such distributed systems cannot become a reality without fundamental solutions, such as those outlined, to the specification and interface problems. Their complete solution on the other hand, provides the potential for further major functional evolution and growth.

9 Concluding Remarks

The characteristic of continuing evolution that involves growth, maintenance and increasing complexity is intrinsic to the very being of large software systems. The resultant

indeterminacies of system state, system function and system capability make them costly to implement, even more costly to maintain and could prove disastrous in certain applications. As a consequence, there is a limit to the size and functional content of software and software controlled systems in their present form. However, the developing technology of micro-processors, together with the concepts of total, structured specification and standard hardware interfaces between self-contained software units is seen as leading to the gradual evolution and future development of large software systems in the form of distributed, highly parallel, systems.

Finally we note that large software systems display all the features characteristic of large systems in general: static and dynamic properties and behaviour patterns that are being increasingly discovered and described by an emerging systems science (eg [SUT75]). But they also display peculiar properties that are a direct consequence of the implementation technology - programming - used in creating, maintaining and enhancing these systems [LEH78].

In discussing the characteristics of large systems in the present context, we do not attempt to specifically identify the more general systems properties. It is, however, well worth drawing attention to the contribution that systematic application of systems thinking, the 'systems approach', can be expected to make to the future development of software engineering concepts: software systems engineering.

CHAPTER 15

ON SOFTWARE COMPLEXITY*

A survey of complexity measures

1 Introduction

In 1974 there was very little known work on complexity in general, and even less on the complexity of programming. At that time Professor Beilner (now at the University of Dortmund) and I shared an office at Imperial College and, according to our mutual interest, discussed problems of 'complex' problems and of large scale programming. Sometimes we asked: what is complexity? Is it possible to come up with a reasonably precise definition? Is the intuitive concept of complexity quantifiable? Since we could not answer these questions, we turned to the literature. And soon, among the many rather philosophical essays on complexity, we hit upon a very interesting doctoral thesis by van Emden [EMD71], 'An Analysis of Complexity'. In addition, we also found a sizable body of work on computational complexity.

The thesis work was based on the concept of conditional probabilities and on the formalism of information theory, and appeared suitable to model complexity of interconnected systems, such as programs built of modules. The computational complexity work was, however, only marginally relevant to the problems of designing, understanding and manipulating programs: program complexity was expressed as the measure of memory space and execution time demand of a coded algorithm. This measure is more or less independent of the program's development history, and of the difficulty to understand its workings, and thus says very little about them.

From these early readings it was clear that we faced a very difficult task (five years later this paper still claims that measuring the complexity of programming is far from being a usable method). But we also learned that complexity can be perceived in at least two different ways: sometimes it appears as a measure of uncertainty or surprise (the

information theoretical formulation), and sometimes it is deterministic and defined as a count or magnitude (as the amount of storage cells or the number of instructions necessary to execute an algorithm).

Since that time the literature search has continued and by now my complexity file, still growing, contains about eighty reasonably relevant papers. The objective of the present paper is to give a brief and a little bit organised survey of this 'complex' repertoire of approaches, which all try to capture the elusive program attribute called complexity.

2 Categories

As the published material became large, classifying it became necessary and fortunately possible. Firstly, a few quite distinguishable approaches to model complexity appeared to emerge, and most papers fell naturally into one of four categories. Secondly, and more or less independently of the approach taken, the objects or processes under study varied from programs to interconnected systems to human perception. These two independent views of classification then led to a two-dimensional tabulation of the collected papers.

The first of the four approach categories is labelled 'informal', contains some delightful but in practice not directly usable philosophy, and some advice as to how to cope with complexity - while the term itself remains undefined. Papers in this category appeal to intuition and their only foundation is the common meaning of the word 'complicated'.

The approaches of the second category are deterministic, with the objective of precise quantification. Typically, some countable property of the studied object or situation is selected and then the count is asserted to be proportional to complexity. Underlying this approach is the belief that meaningful cost and other estimates can be based on counts; analyses can then be performed to predict programming workload, or to aid design or similar selection processes. An example is the mentioned analysis of algorithms.

Perhaps the most exciting approach is that taken by papers of the third category. Here, complexity is perceived as the measure of *uncertainty* - under closer examination a quite reasonable assumption about real life situations. Consider a person involved in a complex intellectual task. You may visualise him sitting at a desk, rapidly scanning available and relevant information in his head, as well as spreading

out notes on the desk in front of him. The larger the desk and the more it is covered, the greater variety of information may be needed at *any* time in his work. It must be uncertain, ie only probabilistically specifiable, which piece of information is needed next; where it certain, then notes and other sources could be produced and ordered in advance, and then put on the desk one by one, with the result of having but a few notes exposed at a time. Notice that in the alter case the required work would still be significant, but less complex, because of its being more predictable and more organised.

A notion closely related to uncertainty is *variety*. If, in doing some task, the variety of information and other necessary equipment is large, then the task is complex. Conversely, a large number of essentially identical items or tools, or their repetitious use, present less problems, and matters are generally simpler. It happens that information theory is founded on a similar notion of uncertainty which is mathematically formalised as *entropy*. Approaches within this third category are therefore strongly influenced by information theoretical concepts: complexity grows with the variety of objects, states of processes under consideration, as well as with the lack of a priori information about the relative relevance of the objects to the next task. For the host the composition of a menu is more complex, the more choices are available and the more uncertainty exists as to availability, custom, taste and other preferences of the guests. Equal, unguided, possibilities present the greatest challenge.

In the most frequently used sense, the word complexity characterises some human activity, such as understanding, producing, putting things together, solving problems, etc. Complexity can thus be measured directly as the time taken by the activity, under the assumption that, with other factors being equal, a complex task takes proportionately longer time. In the fourth category of approaches we collected such empirical work on complexity. Here, in most cases comparative measurements are employed to find relative complexity.

In the second (vertical) dimension of the matrix, papers are lassified by the object or situation under study. Most of the collected papers are, of course, limited to programs and programming, but we have also found quite a few relevant articles which are also applicable to the domain outside of programming. Accordingly, we labelled the two major categories as software and non-software.

SOFTWARE				
	Informal	Counts	Probabilistic	Empirical
Algorithms	Chaitin	Aho Traub	Rabin Chaitin Pippinger	
Control Structure	Chapin	McCabe Farr Myers Cobb Woodward	Chen	Savage
Data		Valiant Yin	McKeeman	Basili Lehman
Composite	Hoare	McClure Rugaber Halstead Knuth Mills Gileadi/ Ledgard		Walston/ Felix Feuer Elshoff Kolence Uhrig Weissman Stucki
NON-SOFTWARE				
Systems	Parnas Simon Myers Jones	Belady/ Lehman Basili Gilb Belady/ Evangelisti	Belady/Beilner Haney Williams van Emden	Ferdinand
Tools		Considine Halstead Lawson	Laemmell/ Shooman	Symes
Under- standing	Weaver	Schorer	Klare McKee	van Gigch Curtis/ Love Weissman

Table 1: Categories

Within the software category we distinguish between four sub-categories: algorithmic complexity (the least related to the programming process), measures based on program control, measures based on data-structure (or flow), and measures of composite program attributes (control and data flow may or may not be included).

In the nonsoftware category, we listed the complexity studies of interconnected systems (often immediately applicable to large software systems), complexity studies of hardware (ie logic circuits), measures of complexity of tools (such as language) and of comprehension, understanding.

Table I is the two-dimensional matrix, with references to major papers appropriately indicated. In the next section we discuss quite briefly some prerepresentative work on the quantification of complexity.

3 Some Examples

In the category of deterministic control flow complexity the best known work is McCabe's [MCC76]. With the control flow graph of the program given, he proposes as complexity measure the number of distinct execution sequences which are possible along the directed graph. The application of this matrix has become quite widespread, because the number of paths is easy to extract automatically from existing (and machine stored) programs. This approach also appeals to intuition: a person reading a program must mentally follow all control paths in order to fully understand the program. Unfortunately the even more complex activity of following data reference paths is completely neglected in this model.

Another control flow based measure was proposed by Woodward et al [W0079]. The basis of their approach is program text, amended by lines with interconnect statements where control may be passed between them. These lines occasionally cross each other and thus create 'knots'. The complexity then is assumed to be proportional to the knot-count. Indeed, well-structured, easy-to-read programs have less knots, but again data references are not included here (although the knot method could include graphs as well).

Also related to the above approaches are Cobb's 'reachability' measure [COB78], Myer's [MYE77] extension to McCabe's model, and an early paper by Farr and Zagorsky [FAR65] proposing the density of IF statement as a measure of complexity.

Significantly different is the approach taken by Yin and Winchester [YIN78]. Here data flow complexity is considered basic, and the associated graph's departure from its spanning tree is defined to be the measure of complexity. The rationale is that in a tree a unique path leads to each node - a sort of minimum complexity.

The most comprehensive of the deterministic approaches based on program object counts is unquestionably Halstead's software science [HAL77]. It is based on the four counts of: distinct operators, operands and total occurrences of operators and operands in the program. From these numbers bounds and estimates of program size, programming effort, etc, are derived. The approach has received considerable attention (15.1). Often, experimentalists summarise their own results in terms of Halstead measures, or test and verify the claims of software science. Such work has been reported for example by Curtis [CUR78].

There are too many other deterministic proposals to mention them all. Due to its originality, however, we cite here Mill's proposal [MIL] to measure complexity of a program by the length of text necessary to prove its correctness. (Will this motivate for simplicity?)

For modular systems an example is the Belady/Lehman [BEL76] model in which complexity of system modification is captured as the ratio of modified to total number of system components (modules). Clearly, if modification gets diffused into a larger portion of the system, then it must have been more intertwined and complex than a system in which modifications remain confined and localised. The approach has been successfully used to predict modification workload of a large operating system which has been evolving over a ten year period.

However interesting and promising, information theory based approaches are rather rare in the literature and appear concentrated either around the study of probabilistic algorithms or of interconnected systems in general. More specifically for program systems Belady and Beilner attempted to capture the complexity of program evolution by introducing distributions over the set of modules of the probability that (a) a change hits a given module and (b) that another module becomes impacted by the change. The scheme is formally quite close to the entropy approach of van Emden's [EMD71]

15.1 (Eds) *But by now (1985) largely discredited.*

mentioned in the introduction. Another, earlier, effort by Haney [HAN72] models the change propagation along a graph of edges labelled by probabilities and spanned by modules as nodes. Unfortunately, very little experience exists with these approaches and at present they are subjects of research.

There are numerous papers in the literature on readability, complexity of comprehension, frequency distributions of words and symbols in natural languages as compared to program symbols, many based on probability theory. But again, experience with respect to their usability is practically nil.

Much more promising are the empirical approaches. In one of the earliest studies L Weissman [WEI74] at the University of Toronto identified a number of program constructs and attributes and ranked them according to the associated relative difficulty which a group of students encountered while programming. By implication, a construct is more complex the more difficult it is to apply and understand in a program. Encouraged by earlier results, recently more professional psychologists turned their attention to the empirical evaluation of programming complexity. They usually conduct joint efforts with computer scientists, with the objective to understand human factors of programming tools and techniques and to test in practice the measures arrived at by speculation, which would otherwise be doomed to oblivion. Some such work is presented in Bill Curtis' [CUR78] accompanying tutorial on complexity.

Surprisingly, many people have something to say about the unsettled subject of complexity. Many thought-provoking papers are listed in the first column of Table 1. Anyone interested should read them since, due to the variety of the ideas, it would be very difficult to give a summary here. However, it is hard to resist the temptation to quote Herbert Simon [SIM69] for conclusion: '... complexity in the computer program was, to a considerable extent, complexity of the environment to which the program was seeking to adapt its behaviour'.

4 Summary

Quantification of programming complexity is far from being mature. Only experimentation will be able to bring about slow but steady progress, eventually enabling us to predict

program quality and programmer productivity, by inference from only a few factors which are available at the start of a software project.

5 Acknowledgement

I collected the material for this paper with H Beilner. His insight was essential for the above presentation.

CHAPTER 16

A MATHEMATICAL MODEL FOR THE EVOLUTION OF SOFTWARE* (16.1)

1 Introduction

Evolution dynamics is a theory describing the growth over time of large computer programs. It has been developed by Lehman and others ([BEL72]; [BEL76]; [LEH76b]; [LEH78a] and other papers) to explain certain features of the evolution of large programs, such as limitations in the average growth rate and difficulties which commonly follow growth spurts. The theory distinguishes between *progressive* work to introduce new features, and *anti-regressive* work to make the program well-structured, documented, understandable, and capable of further development. Overly rapid growth often leads to partial neglect of this latter component. Both types of work are needed together in order to make progress in developing the program.

To apply evolution dynamics, we must be able to make quantitative statements about the balance of the two types of work. That is, we need a mathematical model. Our need is well summarised by Harvey M Wagner [WAG75], page 7:

Constructing a model helps you put the complexities and possible uncertainties attending a decision-making problem into a logical framework amenable to comprehensive analysis. Such a model clarifies the decision alternatives and their anticipated effects, indicates the data that are relevant for analysing the alternatives, and leads to informative conclusions. In short, a model is a vehicle for arriving at a well-structured view of reality.

The purpose of this paper is to find a model for evolution dynamics.

16.1 (Orig) *This work was done while the author was visiting the Department of Computing and Control, Imperial College, London, England.*

The model should:

- 1 embody the qualitative ideas of evolution dynamics (summarised in the next section,
- 2 explain the data observations in the literature, and
- 3 be as small and simple as possible, consistent with (1) and (2); a simpler model is easier to fit, to understand, and to use and thus (paradoxically) may be more powerful than a more complex model.

Starting with the model of Riordon [RIO77], this work has simplified the number of model variables from five to two and reduced the number of parameters as well (though not so greatly), without sacrificing effectiveness under (1) and (2).

The paper is concerned with the mathematical model; some questions of data measurement (such as the measure of the size or power of a program) are not addressed here. Our measure of size is that adopted in references [BEL72]; [BEL76]; [LEH76b]; [LEH78a], namely, the program 'module', which must then be defined in specific cases. We use the same kind of complexity measure as in [RIO77], for which there is as yet no objectively measured variable. This does not invalidate the definition; complexity is fully specified indirectly by its effect on program growth.

The model is not intended as a magic formula. In its present state it requires a good understanding of the application, particularly as regards the consistent definition of program size, the relationship between size and power (this is an *estimation* step - estimation of lines of code, or modules, needed for particular functions), and the particular forms taken by complexity. Allowance must also be made for random variations from the model predictions, and the size of these depends on the case.

2 Program Growth Dynamics

This section describes the types of program growth that an evolution dynamics model must explain, and then defines the model equations. We begin with some of the data on which evolution dynamics is based, shown in Figures 1 and 2.

Notice that:

the average growth rate [the slope of $M(t)$] stays near a certain average value, but declines as the program gets larger;

the growth rate in Figure 2 *oscillates* over the early portion of the curve. This was a series of ambitious releases alternating with 'cleanup' releases. Thus the two types of work identified in the introduction alternated in this period.

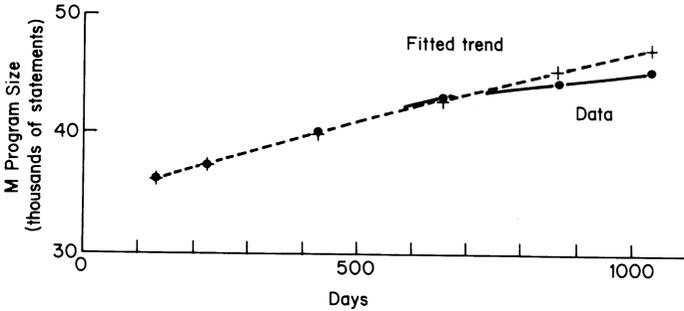


Figure 1 Growth Data for System P

A series of laws of evolution dynamics have been formulated [LEH78a] to explain these and other observations. For our purposes we can summarise them in two observations about growth.

Observation 1: Each project has a certain maintainable average growth rate which is very difficult to alter. Attempts to exceed it lead to problems and subsequent cleanup work, as in system T (Figure 2). As the program becomes larger, the average maintainable rate declines and the release interval tends to increase. (The third, fourth and fifth laws in [LEH78a] elaborate on this observation.)

Observation 2: Growth introduces disorder, entropy, or complexity (mismatched and unmet specifications, undetected errors, inadequate and incorrect documentation, and new internal connections) which reduce the efficiency of further work. The disorder or complexity can be reduced only be effort directly expended on it. This leads to the aforementioned distinction between progressive and antiregressive work. (The first and second laws in [LEH78a] bear on this observation.)

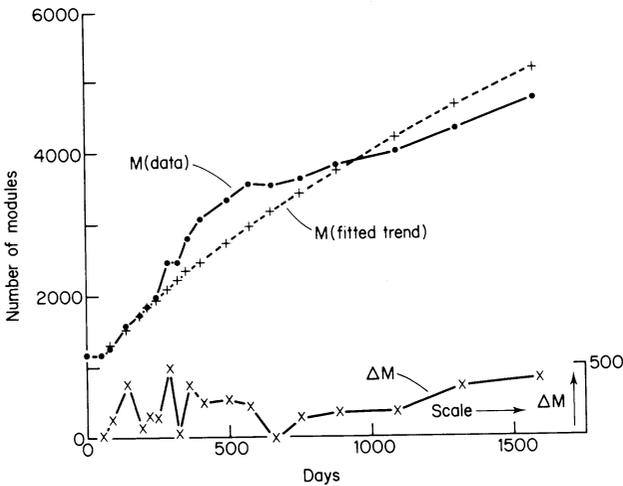


Figure 2 Growth data for system T:

Size M and increment $\Delta M(k) = M(k) - M(k-1)$.
Scale for ΔM at the right

2.1 Model Variables

Program $M(k)$ at release k is measured in 'modules', following [BEL72]; [BEL76]; [LEH76b]; [LEH78a]. We assume that a logically consistent modularity is imposed so that 'more modules' implies 'more functions implemented in the program', and that implementation targets can be converted into 'number of module' targets.

To represent the growth-limiting effects of disorder described in observation 2, we introduce an efficiency function f_E . Relative to a desirable standard level of order, the output rate of the project team is multiplied by f_E , which is less than 1 for disorder above the standard level. The actual disorder level itself will be defined as a variable U such that

efficiency f_E is a function of U , $f_E(U)$;

the 'standard' value is $U = 0$ at 100% efficiency,
 $f_E(0) = 1.0$;

usually $U > 0$; and

$$\partial f_E / \partial U < 0$$

The function f_E is the inverse of Riordon's complexity function [RIO77], and just as he chose an exponential form, for the same reasons we shall suppose that $f_E(U) = \exp(-U/K_E)$. This form meets the requirements above and is mathematically simple. The constant K_E determines the slope of the function $f_E(U)$ as $-1/K_E$ at the standard level $U = 0$.

This paper assumes that the many forms of complexity listed in observation 2 can be meaningfully lumped together in a single index variable U . Small values of U are associated with such terms as 'good structure', 'good software quality', and so on; however, the defining property of our variable U is its productivity-reducing effect on efficiency. Thus our definition of U is precise even though it is not yet obvious how to measure it directly.

The term 'software complexity' usually has a stricter, narrower meaning than our variable U and implies an objectively measurable property of the program code and data. The definitions of McCabe [MCC76], Chen [CHE78], and Benyon-Tinker [BEN84] are in this spirit and are based on graphs of the program and measures of their inter-

connectedness. Chen did an experiment to relate his complexity measure to programmer productivity, but because he studied only novice programmers working on small programs, his results are not directly usable here. Work is continuing in this area and eventually may produce a measureable version of the U variable.

2.2 New vs Structural Work

The effort exerted on the kth release is regulated by a budget $B(k)$. We shall assume that the rate of expenditure per unit time is fixed at B and that the budget for release k is related to the time interval $\Delta t(k)$ spent on it by $B(k) = B \Delta t(k)$. The effort $B(k)$ is divided between new (progressive) work $B_N(k)$ and structural (antiregressive) work $B_S(k)$, where the latter covers effort expended on structuring new code, on old code, and on the overall system structure. The division is made by an allocation function $g_S(k)$, $0 \leq g_S(k) \leq 1.0$; thus

$$B_S(k) = B g_S(k), \quad B_N(k) = B(k)(1 - g_S(k)).$$

The effectiveness of B_S and B_N are both decreased by disorder U through the efficiency factor f_E .

$B_N(k)$ is directed to producing new modules of program, and each unit of $B_N(k)$ leads to K_N modules if $U(k-1) = 0$, $f_E(U(k-1)) = 1$, or to $K_N f_E(U(k-1))$ new modules in general. $B_S(k)$ is directed to controlling $U(k)$.

The time evolution of $U(k)$ is made up of two opposing components at each release: a 'potential increase' $U^+(k)$ and a reduction $U^-(k)$, so

$$U(k) = U(k-1) + \Delta U^+(k) - U^-(k).$$

The potential increase arises whenever new work is done because the new modules affect the rest of the program in ways that are unforeseen, so it is jointly proportional to the amount $B_N(k)$ of new work and to the size $M(k-1)$ of the pre-existing program through a function $f_M(M(k-1))$. It is also greater in a more disordered program and it is proposed here that $\Delta U^+(k)$ is proportional to $U(k-1)$. With K_U as a constant of proportionality,

$$\Delta U^+(k) = K_U U(k-1) f_M(M(k-1)) B_N(k), \quad df_M(M)/dM > 0 \quad (1)$$

This product form is merely a plausible and simple form of the more general form

$$U^+(k) = f(M(k - 1), U(k - 1), b(k)).$$

$B_S(k)$ includes all the extra effort expended on understanding the pre-existing system and properly integrating the new modules, and has its effect through $\Delta U^-(k)$.

2.3 Reduction of Disorder by Structural Work

Potential new disorder ΔU^+ and previously existing disorder $U(k - 1)$ are together reduced by allocating effort to $B_S(k)$. Using similar reasoning, the effectiveness of $B_S(k)$ is proportional to U (since problems are more easily found if there are more of them) but there is a counter-effect due to reduced efficiency, a disordered program being harder to work on. With proportionality constant K_U , these considerations give a reduction.

$$\Delta U^-(k) = K_S U(k - 1) f_E(U(k - 1)) B_S(k). \quad (2)$$

In defining (2), we have for simplicity used the same efficiency factor f_E as for production of new modules; these might in fact be different functions since the activities are quite different. The counterproductive effect often observed, where structural work produces new errors, is implicitly taken in account in (2) by reducing the value of K_S .

2.4 The Model Variables and Equations

As a summary the variables defined above are listed for convenience:

- $M(k)$ = program size in modules
- $U(k)$ = Index of the disorder, or complexity state, of the program (per module) in arbitrary units.
- $f_E(U)$ = relative efficiency of work. We shall use $f_E(U) = \exp(-U/K_E)$. Then K_E is the value of U that reduces efficiency to $e^{-1} \approx 37\%$.

- $f_M(M)$ = Impact of program size on the growth of disorder. Supposing the growth of U to be promoted by interactions between modules, the function $f_M(M) = M$ is used in this paper to model homogeneity of interactions between old modules and new. $f_M = 1$ would model a constant number of interactions per module, independent of program size.
- $B_N(k)$ = budget value for work on new modules (arbitrary units, possibly money).
- $B_S(k)$ = budget value for work on structure.
- $B(k)$ = $B_N(k) + B_S(k) = B \Delta t(k)$, which defines B as a constant rate of expenditure over time.
- $\Delta t(k)$ = time interval between releases $k - 1$ and k .
- K_U = conversion factor, disorder units per unit of UMB_N , if $f_M = M$.
- K_S = conversion factor, disorder units per unit (UB_S).
- K_N = productivity in modules per budget unit at 100% efficiency.

2.5 The Model Equations are now

$$\begin{aligned}
 M(k) &= M(k-1) + K_N f_E(U(k-1)) B_N(k), \\
 U(k) &= U(k-1) [1 + K_U f_M(M(k-1)) B_N(k) \\
 &\quad - K_S f_E(U(k-1)) B_S(k)] \\
 f_E(U(k-1)) &= \exp(-U(k-1)/K_E), \\
 f_M(M(k-1)) &= M(k-1).
 \end{aligned} \tag{3}$$

2.6 Parameter Values

The parameters K_U , K_S , K_N are different in each application, but several commonsense statements can be made about their values. for example, K_N modules per budget unit is the maximum possible productivity of the programmers, given well-structured existing code and minimal effort to maintain the structure, ie, for $f_E = 1.0$ and $g_S = 0$. Release policies will probably limit the amount by which complexity U can change in one release, since any set of quality standards will tend to keep U consistent. Suppose that even with $g_S = 1.0$ and $f_E = 1.0$ U is never decreased by more than 50%; then

$$0 < K_S B(k) < 0.5. \quad (4)$$

On the other hand, suppose it is never increased more than 50% either, even with $g_S = 0$;

$$0 < K_U M(k - 1) B(k) < 0.5. \quad (5)$$

The range of disorder U will be controlled by reasonable standard practices in order to retain efficiency f_E might be

$$14\% < f_E < 100\%,$$

which in our exponential function gives

$$0 < U/K_E < 2. \quad (6)$$

Although some cases may differ, these ranges give a rough guide to the parameter magnitudes. Deviations, if they occur, will have some obvious symptoms.

This section has left the function g_S undefined except for the limits $0 \leq g_S \leq 1$, since g_S is a management factor, reflecting the importance attached to good structure and to complaints about structure.

3 Behaviour and Validation

Validation of the above model depends on the reasonableness of its behaviour (sometimes called 'face validity') because the data are too limited for definitive statistical studies. Within this limitation, the model passes three tests:

- 1 It bears out observations 1 and 2, in predicting growth paths with ultimately decreasing growth rate if $B(k)$ is constant.
- 2 Parameter values satisfying the reasonable ranges in the previous section can be found for systems T and P in Figures 1 and 2; the resulting model prediction curves (dashed lines in the figures) are close to the **average** paths of the data.
- 3 A management strategy g_S can be found which explains the oscillation in the growth rate in system T. The strategy is described in the section on responsive management.

To examine the model behaviour, we shall first consider some artificial parameter values $K_N = 1$ module/budget unit, $K_U = 10^{-6}$, $K_S = 10^{-3}$, $M(0) = 1000$ modules to start, and $f_E(U(0)) = 0.5$, so that $U(0) = 0.693K_E$.

3.1 Decreasing Growth Rate: Model Predictions

Figure 3 shows some model trajectories with constant budget and g_s , that is, a fixed amount of effort on new and structural work over time. Many other sets of parameters also give similar declining growth rates. The upper curve represents going all out for rapid implementation, so that after five releases further growth is totally impeded by bad structure (remember this rapid growth includes poor documentation, badly planned code, and all sorts of 'quick and dirty' work). The lower curve represents overkill on structure, with very careful work, but a low growth rate; the middle curve is a compromise at a 50-50 division of effort, and is still capable of further development although its structure is somewhat degraded after five releases. A constant release interval $\Delta t(k) = 100$ is used in Figure 3. Varying it changes the curves very little.

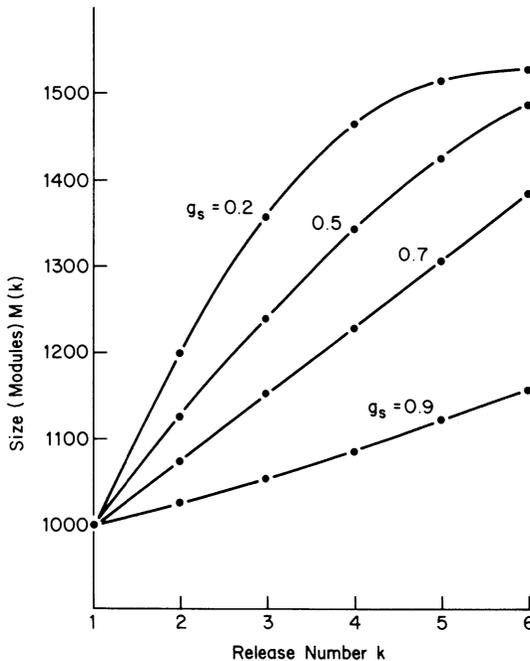


Figure 3: The effect of the allocation function $g_s = \text{const.}$

Model trajectories for parameter values: $M(1) = 1000$ modules, $U(1) = 0.693$, $K_N = 1.0$, $B = 5.0$, $\Delta t(k) = 100$ days/release, $B(k) = 500$, $K_U = 10^{-6}$, and $K_S = 10^{-3}$.

A second reasonable management strategy is to maintain a constant U , representing some standard of software quality. It can easily be shown that this implies

$$g_S(k) = \frac{K_U M(k-1)}{K_U M(k-1) + K_S \exp(-U/K_E)}, \quad (7)$$

$$M(k) - M(k-1) = \frac{b(k)K_N \exp(-U/K_E)}{1 + K_U M(k-1)/(K_S \exp(-U/K_E))}. \quad (8)$$

Figure 4 shows some trajectories using (8) for three values of U ; smaller U gives enhanced productivity but the decline in growth rate is still evident. That is, even with constant U the effect of a larger program is to absorb resources and reduce the growth rate.

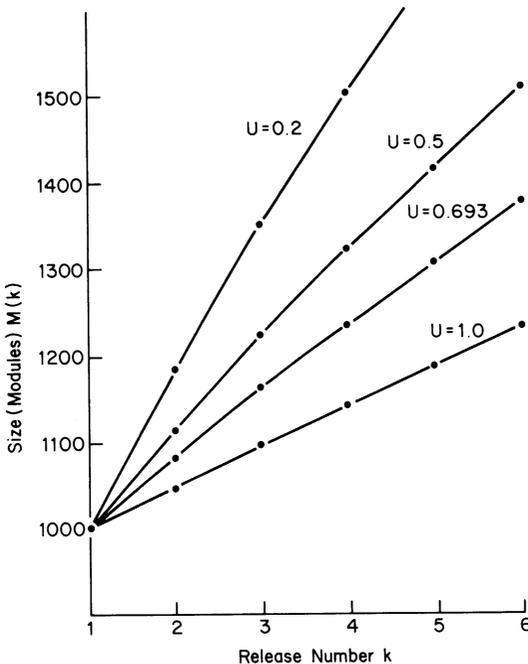


Figure 4: The effect of various constant values of U . Model trajectories for g_S chosen to keep $U = \text{const}$ and other parameters as in Figure 3.

3.3 Parameter Values

Equation (8) is equivalent to the following curve for the program size:

$$M(k) = M(k - 1) + a \Delta t(k)/(1 + bM(k - 1)). \quad (9)$$

When $U = \text{const}$ is a reasonable assumption, this could be fitted as a regression curve. This was done for systems P and T, giving the dotted lines in Figures 1 and 2 and the following equations:

$$\begin{aligned} \text{system P } \hat{M}(k) &= \hat{M}(k - 1) \\ &+ 36.4 \Delta t(k)/(1 + 5 \times 10^{-5}M(k - 1)), \end{aligned} \quad (10)$$

$$\begin{aligned} \text{system T } \hat{M}(k) &= \hat{M}(k - 1) \\ &+ 9 \Delta t(k)/(1 + 8 \times 10^{-4}M(k - 1)). \end{aligned} \quad (11)$$

These fits were made by eye, to see how closely they matched the data; with these few points, statistical fitting would not be meaningful. We conclude that a good average fit can be obtained, but the oscillations in growth rate of system T are unexplained. They arise due to manipulation of $g_S(k)$ by feedback effects, as explained in the next section.

Notice that constant U (which is related to consistent and successful software management) simplifies the model to one equation and two parameters; this seems eminently reasonable. The data on system P did not justify further investigation of parameters beyond the condensed form of (10), but system T was investigated further.

3.4 Responsive Management

The system T data shows an oscillation in the growth rate of releases 4 - 11 in Figure 2, which is not explained by the g_S policies of the previous section. This section relates such oscillations to the sensitivity of management feedback.

First, we roughly calibrate the model (3) for system T, consistent with the simpler model function (11) and with the commonsense ranges of values (4) - (6). The budget unit is chosen so $B(k) = 1$ and the complexity unit so $K_E = 1$.

Over releases 1 - 13, system T grows by an average of about 230 modules per release, so on average $K_N f_E B_N(k) \approx 230$. We shall take average values of $f_E = 0.5$, $g_S = 0.5$, making $K_N(0.5)(0.5) \approx 230$ and K_N close to 1000. With M in the range

1000 - 4000, (4) and (5) suggest $K_U \approx 10^{-4}$ and $K_S \approx 0.25$. These values are consistent with (11). The model is now

$$\begin{aligned} M(k) &= M(k-1) + 1000 \exp(-U(k-1))(1 - g_S(k)), \\ U(k) &= U(k-1)[1 + 10^{-4}M(k-1)(1 - g_S(k)) \\ &\quad - 0.25 \exp(-U(k-1))g_S(k)]. \end{aligned} \quad (12)$$

From the data, it is clear that $g_S(k)$ varies widely over releases 4 - 11. Our explanation, like that in [RIO77], is that the variation is due to *responsive management* reacting to users' and programmers' complaints about bad structure when $U(k)$ exceeds a satisfactory level \bar{U} . The manager increases $g_S(k)$ when $U(k-1) > \bar{U}$, but decreases it when $U(k-1) < \bar{U}$; if managerial response is sharp enough, the one-release delay in information can cause an oscillation as seen in Figure 2. In control systems terminology, the 'gain' is too high.

Mathematically we express g_S in the feedback form

$$g_S(k) = g_S(U(k-1), M(k-1)).$$

At $U = \bar{U}$, g_S is given by (7) to keep $U = \bar{U}$, but its slope $\partial g_S / \partial U$ is determined by the manager's reactions. Further, the equation for $U(k)$ in (3) will be *linearised* about average values for \bar{M} for $M(k)$, $\bar{f}_M = f_M(\bar{M})$, $\bar{f}_E = f_E(\bar{U})$, \bar{B} for $B(k)$, and g_S for $g_S(U, M)$ to give the following linearised model for $U(k)$:

$$U(k) \approx \bar{U} + \partial U(k), \quad (13)$$

$$\begin{aligned} \partial U(k) &= a_U \partial U(k-1), \\ a_U &= 1 - \bar{U}B[(\bar{f}_M K_U + \bar{f}_E K_S)(\partial g_S / \partial U) - \bar{f}_E K_S \bar{g}_S]. \end{aligned} \quad (14)$$

For system T with $f_M = M \approx 2000$ modules over releases 1 - 13,

$$a_U \approx 1 - 0.693[0.325(\partial g_S / \partial U) - 0.125g_S]. \quad (15)$$

The linearised coefficient a_U includes the manager's sensitivity $\partial g_S / \partial U$. Equation (13) will give an approximation to (3) in a neighbourhood of the average values, and will give damped oscillations (such as observed in Figure 2) if

$$-1 < a_U < 0. \quad (16)$$

The actual oscillations are probably initiated by components in the planning process not included in the model, such as changes in specifications or demands for more productivity, perturbing the underdamped closed loop.

Using (15) and ignoring the small term $0.125\bar{g}_S$ (since we do not know \bar{g}_S), we have for system T

$$\begin{aligned} a_U = 0 & \quad \text{implies} \quad \partial g_S / \partial U \approx 4.4; \\ a_U = -1 & \quad \text{implies} \quad \partial g_S / \partial U \approx 8.8. \end{aligned}$$

The reality over releases 4 - 11 is in between; reflection shows that this is a *very steep slope* for the reaction function. The range of values taken by U [see (6) above] is expected to be about 0 - 2, and the range of g_S is strictly 0 - 1 (or less, more probably less), so a value of $\partial g_S / \partial U$ as high as 5 indicates that $g_S(u)$ is almost a switching function.

Thus we have demonstrated that the model can plausibly explain the behaviour of system T. At the same time we have derived a caution not to respond too sharply to feedback information from users or higher management. To avoid these unsettling oscillations, $\partial g_S / \partial U$ should be small enough to make $a_U > 0$. Here the condition is $\partial g_S / \partial U < 4.4$, and for other systems, by taking the extremes of the ranges (4) and (5) and using $\bar{U} = 0.693$ again, we obtain $\partial g_S / \partial U < 1.9$ (approximately) for avoiding oscillations.

4 Applications of the Model

Although this paper is mainly concerned with determining the model and testing it, it is informative to see what light the model can cast on planning questions.

4.1 Release Timing

The content and timing of releases determine a growth path. The model will predict that a certain addition to the program (implying a certain number of additional modules) requires a certain length of time. Figure 4 shows how, if the increment in M is the same from release to release and a certain standard U is maintained, the release interval must lengthen as the program grows larger (the curves follow the same path over time for different release timings). In broad terms this bears out part of the observation 1 that were mentioned earlier.

4.2 Planning Two Releases

A sophisticated question may be asked about two consecutive releases: supposing the initial U and M to be known and the final U to be specified, and budgets for both releases to be

known, how is the work divided over the releases? In particular, is it better to go for growth and new modules in one release and to clean them up in the other, or is one to follow a plan that keeps the quality as constant as possible?

Figure 5 shows a state-plane with trajectories of (M,U) for $k = 1,2,3$ and $U(1) = U(3) = 0.693$, $B(2) = B(3)$, and $M(1) = 2000$. In choosing $B_N(2)$, $B_S(2)$, $B_N(3)$, $B_S(3)$ there turns out to be only one free variable, which is chosen to be $U(2)$ to simplify the equations. The lower part of Figure 5 displays $M(3)$ as a figure of merit against $U(2)$, since in our formulation more modules mean more functions programmed, so a larger $M(3)$ is better.

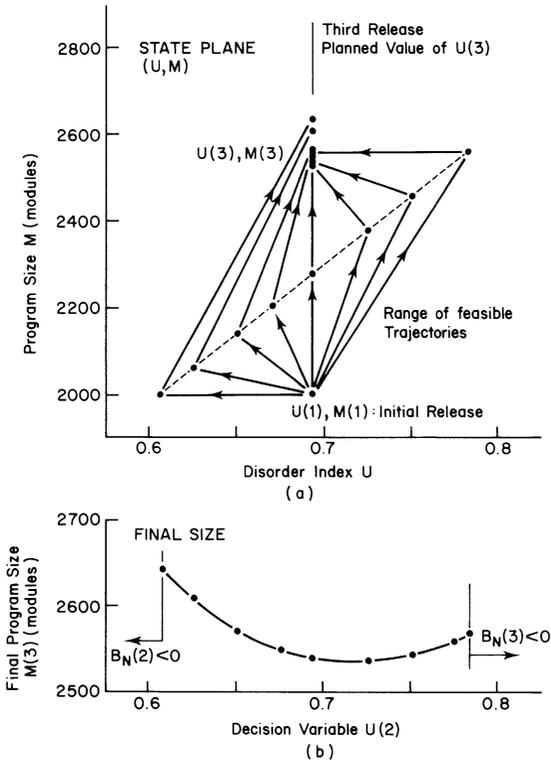


Figure 5: Optimal management of structure (disorder = U) over time. For two successive releases with equal budget, what growth pattern gives the greatest final size? (a) Stateplane trajectories $M(3)$ over two releases; (b) optimality - final size $M(3)$ vs decision variable $U(2)$

The results, however are contrary to the author's expectations. Consistently well-structured growth with $U(2)$ equal to $U(1)$ and $U(3)$ is nearly the worst case, and either of the feasible extremes is preferable. The same result was found for several other sets of parameter values, so the conclusion is not sensitive to the model parameters.

The model therefore shows a clear preference for alternating growth and cleanup. It places the sequence 'cleanup, growth' above the sequence 'growth, cleanup' but that seems less significant. Note that 'release' in the model can mean any milestone, so each release could be divided into two phases and planned this way, to give actual releases of consistent quality.

This is a tantalising result (16.1), but it would be premature to build too much on it. Altogether different lines of research may be needed to tell finally whether software development should be 'smooth' or 'jerky'. The author has found on reflection that personally he preaches the former and practices the latter.

5 Conclusions

The model described above represents a theory for software evolution at a very macroscopic level. It has satisfied various tests against the data, and fits the observations of evolution dynamics. It is a simpler model than previous attempts, which makes it a more useful conceptual tool.

Using the model, two management issues have been examined: the sensitivity of a responsive manager to feedback (which may cause oscillating behaviour over several releases) and the allocation of effort between progressive and cleanup work over two successive releases. In the latter issue, the mode favours an alternating allocation over a smooth and consistent one, which may be a significant result in software planning.

16.1 (*Eds*) *But conforms to the conclusion reached in other chapters of this book, and common de facto industrial practice, based on practical experience.*

CHAPTER 17

MODIFIABILITY OF LARGE SOFTWARE SYSTEMS*

1 Introduction

Modifiability of procedures and artifacts is an increasing problem in modern civilization. On one hand, social progress brings about an increase in freedom of choice and action. This would mean that, as individuals or members of communities, we are free to change that part of the world over which we have legitimate control. Yet, as we all know, it has become increasingly difficult to exercise this right for many of the changes have unpredictable or predictably unpleasant side-effects.

Changes are often results of choices made of alternatives. The availability of alternatives is of course made possible by the recent massive increase in productivity, due to social cooperation in modern societies. But this is unfortunately coupled with a growing interdependence of all participants - producers and consumers. It is this interdependence which makes choices risky and the often unavoidable changes painfully expensive. Let us look at a few examples.

The woman scientist with a heavy publication record gets married, and if she chooses to change her family name to that of her husband's she may lose, at least for a while, the continuity of her professional reputation. (We don't even mention problems of mailing address, driver's license, bank accounts, etc., which almost all marrying women must face). This problem is that of 'information *investment*' in the minds of fellow scientists.

Or take the architect of the beautiful medieval cities, for example the ones which surround the Mediterranean. The streets are too narrow for automobile traffic - whom can we blame for not having foreseen the 20th century? Modification of the streets would be difficult because we want to *preserve*, not destroy and replace them just to accommodate a new function for which the original setting was not designed.

My final example is the current changeover to the metric system in the U.S. In this case the consequences are predictable but, as most of us know quite well, the *cost* of

First published in Proceedings—The 14th IBM Computer Science Symposium, October 1980. Reprinted with kind permission of International Business Machines Corporation.

replacing the many tools and instruments of one of the most advanced industries in the world is horrendous, not to mention the *inertia* in people's mind, which acts against the development of a new 'feel' for guessing, comparable to the one within the old system.

Investment, preservation, cost - these are some of the issues which must be considered whenever we contemplate modification. The main question is then how to reduce the impact on the different issues of a change. Clearly, the smaller the domain and the shorter the expected time of the impact the better we are off. *Locality*, in space and time, of the impact of modification seems to be the central concept. The narrower the field and the shorter lived the scientific results, the less unpleasant is the name change; the more primitive the society the easier the changeover to new standards.

We are also interested, as mentioned earlier, in the predictability of the effects of change: we would like to easily *locate* the consequences of planned modifications. It is intuitive to accept that modifiability increases with the ease of indentifying and locating its consequences, and increases with locality, with its 'impact size'.

It seems that large software too displays the mentioned characteristics of modifiability. After all, programs are representations of real life procedures and artifacts: banking, manufacturing, flight dynamics, airplanes. And since they are models, ie abstractions of reality, they are better bounded than parts of the complex real world and thus easier to study. They too must be constantly modified, as was shown almost a decade ago in *Evolution Dynamics* [BEL71B] [BEL76] the study of ever changing large programs.

These studies distinguish between two entirely different dynamics of computer programs. The first and traditional subject is that of *execution* which takes place when a computer, guided by a program, processes data by changing the state of variables stored in the memory. This dynamics is the domain of computational complexity and of the many efforts collectively called performance modelling and evaluation.

The other, and the perhaps more challenging, dynamics is that of *program evolution* which takes place when the program is not running on the machine at all - in fact is itself subjected to changes, enhancements and modifications

continually performed by developers and maintainers. There is not enough space here to give much detail on program evolution dynamics and only one of its major observations is reviewed herewith.

Large operating systems, such as OS-360/370, whose evolution was first studied, are built of modules - components small enough to understand and worked upon by one programmer, relatively independently of, and with little need to communicate with, others. OS-360/370 consists of several thousands of these modules.

During the first ten years of its history the system had about twenty releases. The amount of modification - change and enhancement in each release - was more or less the same over the entire period. At release changeover, some modules remained identical, some others were new versions of old modules, while a third category consisted of brand new modules. It was interesting to observe that for the entire observation period the fraction of modified modules was monotonically and faster than linearly increasing, and towards the end of this twenty-release period actually approaching saturation point at which all modules must be modified to make the next release (Figure 1).

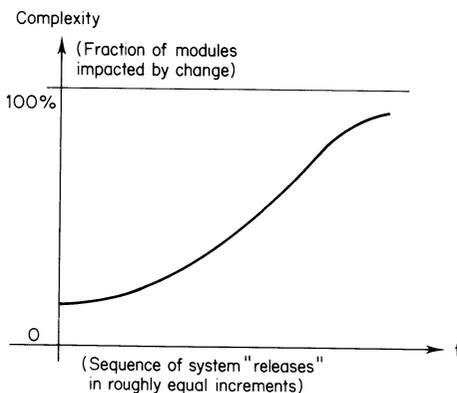


Figure 1

This observation was interpreted as clear indication that repeated modification of software systems is increasingly difficult - each modification creates an extra bit of obstacle to the next one. The reasons for this is that the original structure which was deduced from and well matched to the original requirements is gradually deteriorating as more recent requirements induce changes which do not fit the old structure. Since one must live with an ever changing environment, one must explore methods which at least reduce this deterioration of structure during unavoidable program evolution.

Another way of looking at modifiability is to observe what kind of work is being done by programmers. Following many surveys [GOL73], [BOE76], it is safe to say that significantly more than half of the world's programmer population works with existing programs, and only the minority is involved in developing programs from scratch. It appears that 'modify we must', and our paper is founded on this premise. For the time being we do not question the wisdom of modifying rather than replacing the old program in response to new requirements, nor do we consider the development of techniques for deciding whether modification or replacement is more economical.

2 The Modification Process

First, we must examine what classes of modifications are usually performed on software. The obvious number one category is that of repair, ie, the elimination of faults found during development and subsequent operation in the field. The important issue here is that while hardware repair is *not* modification but replacement of parts or sub-assemblies to get the product back to its original designed state, software repair is always changing *away* from the original state because it is found faulty.

This leads us to the important observation that the so-called maintenance activity is actually *redesign*, corresponding roughly to engineering changes in hardware. If we assume that in product development even low level detail decisions must have been deduced from requirements, then we must also expect that changes at the same level, but now under the name of maintenance, may cause perturbation of higher level choices, or even of the initial requirements.

But in reality maintenance ignores the redesign aspects of its task and actually much less skilled people than software designers are employed for performing modifications. The maintainers' knowledge of the original design is marginal. In addition to this, documentation of design is not only too sketchy, but usually rapidly loses its validity, thanks to long series of earlier (sic - later?) modifications. Moreover, maintainers focus on individual building blocks, the modules, of the system instead of studying the interaction of the components before effecting a change. The top-down approach, if practiced at all in the first place, is replaced by low level patching, whose consequences, uncontrolled and unpredictable, propagate into the system, and thus affect clarity of the entire structure.

But repair is not the whole story. Driven by expanding market requirements for new function and, in the case of operating systems, by the introduction of new, advanced, pieces of hardware, enhancement becomes the other large category of continuing changes. And if performance is critical and the desire to save machine resources is great, system tuning becomes another source of modification because, in practice, modifications introduced by tuning rarely observe clarity of structure or understandability of program text.

But what is usually modified in software? After all, software is a fiction, simply information to guide a machine's operation. At the same time it appears to humans in form of documents - either printed text or projection on a VDU. In fact, several levels of documents must be prepared for developing and maintaining software if it is large enough and employs many people organised into specialist groups: architects, designers, testers, etc. In fact, one way of viewing the process or life cycle is as a sequence of documents, that of requirements, specifications, programs, etc, and transformations from one document into the next. It happens that only the last of these transformations - program text into machine code - is fully automatic and done by a compiler. The other transformations - design out of specifications, or programs out of design - are manual.

We can then make two important observations. The first is related to the labour intensiveness of the process. This leads to high cost which is further amplified by low reliability of human beings in contrast to reliability of machines. Moreover, iteration in the process is also often necessary to iron out errors introduced by fuzzy and noisy

communication between development and maintenance groups, who work under schedule and cost pressure. In this environment only machine processed documents (compiled text) are modified and be promptly updated, namely the ones which comprise the end-product to be delivered, while the associated higher level design documents re left unchanged, reflecting the state in which the software was, and not is. It would cost extra money to keep these documents aligned with code. Of course, this neglect makes the next generation of modifications again a bit more difficult and forces the crew to work with the only trustworthy document, namely the one which runs on the machine: the low level code itself.

3 Research in operating systems

Up to about three years ago the problems of software modifiability did not seem to be appreciated much by the software engineering community. This was indicated by the fact that no exploration - research or other - was conducted at all on maintainability or modifiability. Rather, most efforts were directed to the problems of designing software from scratch. Our software engineering group in Yorktown thus found it first necessary to invent *how* one should do research in software modifiability.

It was felt that modifiability would most directly impact the world of maintenance - excluding major enhancements. We learned from IBM's maintenance organisation that the major cost contributor was labour, and most of this was spent on the actual definition and implementation of the modification. From this and other observations made earlier in this paper we reached the following conclusions:

- There would be little chance for success if we just speculated and theorised about maintenance. Hence we decided to make our hands dirty by working on an *existing software* system, in order to build a more modifiable version of it.
- We should have a double objective:
 - Improved methods and tools for modifying *existing systems* - with good or bad structure.
 - Methods and tools to construct new software which is *easier* and safer to *modify*

- We should not invent and introduce new tools; rather select and, in the process of exploration, test and *refine the already proposed* and promising *ideas* which help achieve the above objectives.

Let us elaborate a little bit on the last point. For a decade or so the concept of *abstraction* has been in the focus of programming methodology. Recently this concept has gone well beyond, and become broader than, the traditional procedural abstraction implemented by a subroutine. Abstraction has been manifested for example by information hiding [PAR72], or abstract data types [LIS74]. Common to them is the following rationale:

the task of working with complex systems is significantly simpler if one explicitly separates the relevant from the irrelevant while modifying or maintaining the system.

This fits well the many formal or intuitive models of complexity, which is regarded as synonymous to uncertainty or variety (variety, since repetitive occurrence of the same thing does not add to complexity) (17.1). Following this idea, simplification in working with software can be achieved by decreasing the uncertainty, either by reducing the information (ie, the number of items to examine) necessary to correctly perform the modification, or by prescribing, as a recipe, with certainty, the steps to be taken (or objects to consider) when modification is performed.

Let us illustrate this by examples. It seems intuitively obvious that modification is simpler if we know that

- a) it does not impact the interface, ie, the calling pattern and associated parameter definitions are left unchanged, and
- b) there is no way but through call invocation that the module is ever accessed from other parts of the system.

In the same vein, of one finds on a single page or at least on adjacent pages in the documentation all data structures which are potentially accessed by a single module, the verification of the impact of a changing module is much simpler than performing a frantic search through the entire program documentation for direct or indirect references issued by the module.

Let us now put the pieces together. Our reading into then current technical literature and discussions with members of the software engineering community led us to the adoption of *data abstraction*, a concept which intergrates information hiding, structural simplification and advanced language ideas into a tool. This tool we thought could have the potential to enforce a particular design in which locating and localizing changes are easier to perform.

Relatively soon we opted for VTAM as a research vehicle. It is an OS370 component of reasonable complexity, a good balance between a perhaps impossibly large operating system (too large for a small research team) and a small 'toy' example which would not help us to validate the experiment. The result was a firm plan of experimentally redesigning parts of VTAM *specifically* for modifiability, using data abstraction [LIS74].

Soon we learned also that our project would have two major phases: the first to work with the existing, official, version of VTAM to extract its functional content, and the second to redesign it anew. We found later that these two phases already predetermined the final output of the effort: on one hand we would develop tools and techniques of 'scoping', i.e. studying, existing systems, and on the other construct facilities to aid the design of new, more modifiable systems.

Based on our discussions and cooperation with several development and maintenance organizations, we have become convinced that current conditions in both domains - work with the old and constructing the new - demand a great deal of substantial improvement. In the following we will describe the tools developed so far for working with existing systems, then turn our attention to problems of constructing modifiable software.

3.1 Work with existing systems

The first obstacle which we encountered was the great difficulty to understand and accurately redocument the actual functional content of VTAM. But this difficulty created just the right milieu: we were in fact playing the role of maintainers who were given a piece of software to work with - debug or update. As already mentioned, if you *really* want to know what the program does during evolution, you cannot trust manually produced documents, which are not automatically coupled to the compilable program text. (17.2)

We found the reading and understanding of the PL/I (17.3) written VTAM code quite slow, in spite of the ample comments supplied in the program text. We therefore started experimenting with different, mostly graphical, program representations, as a possible enhancement of the text proper. After long considerations we rejected the number one contender, the Nassi-Sneiderman (NS) diagram, mostly because of the complications of displaying it on a, say IBM- 3270, terminal. We remind the reader that NS-charts are nested representations of program segments which then appear to shrink indefinitely if surrounded by repeatedly nested constructs. Obviously, a simple alphanumeric terminal, or a line printer, cannot easily cope with this, and the increased complexity of image construction would severely limit widespread applicability.

We were, therefore, literally forced to invent another program notation which we later designated GREENPRINT (in contrast to engineering 'blueprints', programmers use VDU terminals with green phosphorous images). At present GREENPRINTs are generated as follows: a program, developed by us, takes the (PL/1) text as its input and then produces as output an enhanced listing. In the process the program text and the order of statements are left unperturbed, while along the left margin the program tree, ie the control flow, is displayed as indicated in Figure 2. Procedures, loops and decisions are distinguished by the different vertical column types. Detailed information on GREENPRINT can be obtained from [BEL79a].

In addition to our own positive experience with GREENPRINT - aided program 'scoping', an increasing number of co-workers have become interested in using this inexpensive and machine generated scheme. Current work is in the direction of making GREENPRINT interactive, the idea being to permit editing the graph and thus using it as a design tool for efforts from scratch.

The second major obstacle to working with VTAM was the small size of our research team: the already mentioned difficulty of rapidly learning this operating system component, and its size (in excess of 100K lines of code) made a complete redesign prohibitive and we had to satisfy ourselves with re-

17.2 (Eds) *Hence the importance of the new concepts of integrated programming support environments.*

17.3 (Orig) *Actually it is based on a dialect of PL/I used internally by IBM to write system programs.*

A GREENPRINT with indented text

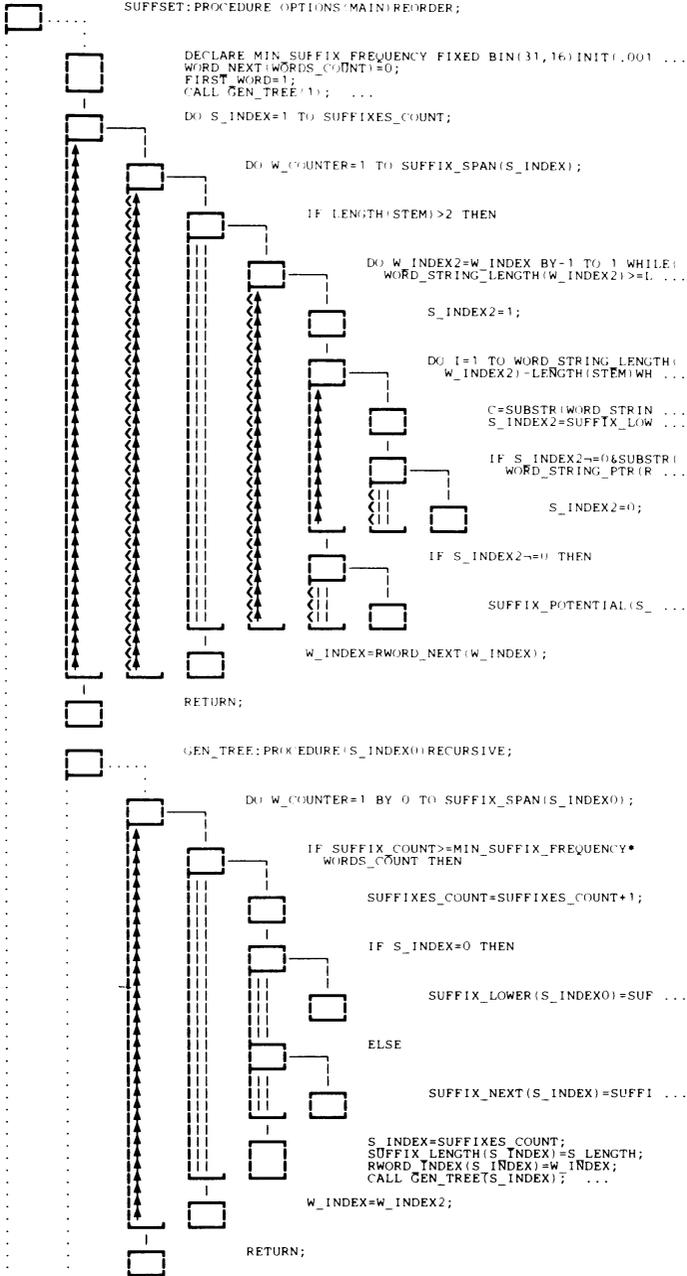


Figure 2

designing only selected sub-components of VTAM. But the sub-components, the ones identified by the original design documents, appeared quite intertwined at the code level.

Again, due to size, it seemed impractical to search manually for reasonably self-contained chunks, namely those having relatively sparse communication or little shared information with the rest of VTAM. To do this automatically, we had, once again, to find or invent a new tool.

At this point we must describe the notion of interconnected control blocks and modules which span the basic structure of most operating systems. Control blocks hold the state information - of the machine and its resources, relating to jobs, tasks, users, messages and other data stored for processing - while modules are the 'actors' - programs which induce state transitions. After having gathered information out of some control blocks, a small set of modules typically updates one or more control blocks. The operating system functions as an aggregate of these individual actions, hence a module has access to many control blocks, while a control block is accessible by several modules and thus becomes a shared object serving as the main communication link between modules.

In the original design the set of modules and the set of control blocks are neatly laid out: modules grouped by function, control blocks by meaning. But during subsequent maintenance and enhancement, clarity of interfaces suffers by ad hoc extensions of the crisply defined control blocks. Extensions become necessary to accommodate new and originally unforeseen functions. The long chain of these modifications results in the already mentioned and observed increase in change penetration into the system, as the number of modules accessing a control block, and the number of control blocks accessed by a module, increase during program evolution.

Figure 3 displays this interconnectedness by a two-dimensional matrix. A matrix provides for the most general, indeed most generous, pattern with which elements of two sets can be connected: m modules and n control blocks may have up to $(m \times n)$ two-way connections. Clearly only a sparse sub-set of this is desirable for a neatly structured system, if we expect not only the reasonably independent development of individual components, but also predictable maintenance during which the number of potentially impacted elements is not too large.

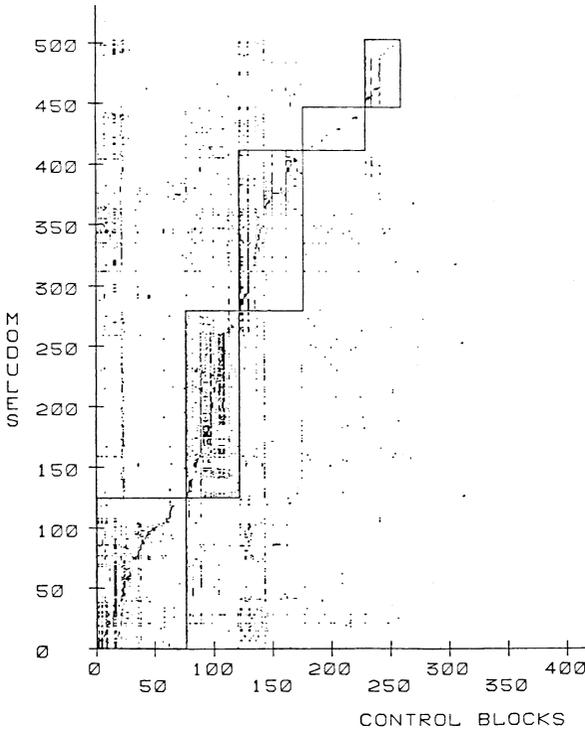


Figure 3

In order to carve out well isolated sub-components, we first machine-stored the connectivity matrix of VTAM. After that we were fortunate enough to find within our building an already existing and immediately usable heuristic program which, although originally developed for the placement of circuit elements on chips, was directly applicable to our problem of identifying *relatively independent clusters of modules and control blocks*. The detail of this work was reported in [BEL79]. This is how our short term action of looking for a tool of this type to master the complexity of VTAM, led to the supply of a generally useful tool for others to work with evolving software and for control of its complexity. (Incidentally, we also made some attempts to formalise complexity on the basis of the connectivity matrix as given in [BEL79].

3.2 Design for Modifiability

As mentioned earlier, we selected data abstraction as our approach to software modifiability. Data abstraction can be applied immediately as a design principle: structure programs (modules) and data (control blocks) such that the access code is packaged with the data rather than interspersed with the actual logic of the programs. Paraphrased, data should interface programs in behavioural, operational terms, rather than in terms of implementation. This would result in the much desired hiding of the detail of the chosen data structure implementation.

A simple analogy is a team of several office workers sharing a file cabinet of several drawers. The particular arrangement of the files in the cabinet is a matter of implementation; what is really wanted is to store certain documents such that they be easy to retrieve later. This means just two fundamental operations: store and retrieve. Yet, in order to perform them, each person in the office must know the rituals of how to get access to any required file. If someone, perhaps in order to optimise, perturbs the order in which the drawers, and the files within, are organised, the others must also be retrained - their knowledge 'modified' - if efficient office operation is to be retained. The alternative is chaos.

A possible solution is to hire another agent who then personally stores and retrieves the requested files. He can optimise the process by using any desired implementation or reshuffling of the files, without the need to reveal this to the actual users who will be quite happy as long as the files become stored and retrieved rapidly and as requested. This is actually the case of isolating the implementation (physical storage) from the usage of the documents.

Similarly, while writing or reading a program, we usually try to concentrate on the program's real function - perhaps some algorithmic procedure or mimicking a physical process - and do not like to be distracted by occasionally having to read unimportant yet complex data-access code. It would be like reading a book intensely, but getting occasionally interrupted with the description of the type setting processes which were used in the production of the book.

If several programs share the same data, then it is even more important that data be associated with its own access code, otherwise each program using these data must contain the

access code individually. Thus, if the implementation of the data structure changes, this must be reflected in each program - a certainly error prone process.

Experience with large software indicates that new design and programming principles are rarely followed readily and rapidly, due to inertia in the programmers' mind, and to the usual schedule and cost pressures which force the choice of methods offering the shortest time to put the program together, and not of schemes making modification easier in the future. Indeed, optimising development and optimising the entire life-cycle (development *plus* the long period of maintenance and enhancement) are far from being coincidental. This is why we thought that short of a tool which *enforces* the use of data abstraction, we would probably not improve modifiability too much.

Accordingly, we decided to develop a language which encourages, in fact enforces, the use of data abstraction. The language was made to be strongly typed, one which permits the user to define abstract data types in addition to the usual built-in types such as integers, characters, Boolean, etc, along with operations associated with each type (as the four arithmetic operations defined on integers). This we did by designing a language called XPL/I, an extended PL/I (see footnote earlier). The design was guided by the existing language CLU developed at MIT. In order to avoid the burden of writing a new compiler, we wrote a preprocessor which accepts XPL/I programs and generates PL/I code for a run-of-the-mill compiler, which in turn produces executable machine code.

Two kinds of *modules* can be written in XPL/I: *procedure* (a la PL/I) and *capsule* which is an abstract data type. Figure 4 displays an example for procedure and capsule in the syntax of the language.

Having a new language, we were ready to write new, more modifiable operating systems - or were we? The new language indeed helped us only to construct individual modules, and the compiler helped to establish the internal consistency of a single module, but left open the even more important issues of how the modules are interconnected, and of machine aided checking and intermodule consistency. So we put forward an effort to develop a tool for *machine aided design* based on data abstraction.

```

PROC MSG_QUEUE_BUILD(BUFFER<<MESSAGE>>, MOD QUEUE<<MESSAGE>>)

  USES ( BUFFER<<MESSAGE>> <OBTAIN, GETVAL> ,
        QUEUE<<MESSAGE>> <OBTAIN, ADD>,
        MESSAGE < > ,
        LOCK < > )

TYPE QUEUE<<T:TYPE>>

  DEFINES
    (
      •
      OBTAIN(QUEUE<<T>>, MOD LOCK) -> BOOL ,
      ADD(MOD QUEUE<<T>>, T, LOCK) ,
      •
    )

  USES (
    •
    BOOL,
    LOCK,
    •
  )

TYPE BUFFER<<T:TYPE>>

  DEFINES
    (
      •
      OBTAIN(BUFFER<<T>>, MOD LOCK) -> BOOL,
      GETVAL(BUFFER<<T>>, LOCK) -> T,
      •
    )

  USES (
    •
    BOOL,
    LOCK,
    •
  )

TYPE MESSAGE

  •
  •

```

Figure 4

On closer examination it became clear that the structure of an operating system, redesigned with data abstraction in mind, could become significantly different than that resulting from a traditional composition by modules and control blocks. Indeed, the experimental redesign of VTAM resulted in a single set of quite similar objects: procedures and capsules, which are implemented in terms of each other.

In order to explain this better, let us examine Figure 5 which is a VTAM function after redesign. Arranged from 'northwest to southeast' are the modules: procedures and capsules. Lines between them indicate the direction of 'uses' relations. For example the capsule 'RPH' (module A) uses for its implementation 'DVT', 'QUEUE', 'MEL' and some primitive types. We call the interconnected set on Figure 5 the *external structure* (ES) of the software design.

There are several important attributes of ES which we would like to emphasise. First, a line represents the set of operations used by the module originating this line, and is performed on the module forming the end point of the line. The operations in the set must be consistent with the 'type rules', and can thus be checked by the compiler, which is now extended into the world of intermodule communication. This world is that of 'programming in the large', in contrast to the implementation of individual modules - 'programming in the small' [DER75].

Visualise a community of software designers - or maintainers - who share and view at their individual display terminals the machine stored single copy of the evolving design, the interconnected objects. They can recall at will the detail specification of the operations defined on capsules. In the future the semantics of individual modules will also be added such that the participants understand rapidly *what* the programmed function is and *how* it is implemented. When a function must be extended, the designers may easily become familiar with the already implemented components, some of which could be included in the extension; *reuse of parts* is thus encouraged. Once, for example, a generic queue is built, it can be used as a template at other places in the system, or even in other systems, by properly 'wiring it' into the external structure. preservation of consistency will of course be checked by the extended compiler.

Or consider maintenance. Each module has two faces: its 'left profile' is the specification, ie, its functional capabilities, while the 'right profile' is the implementation. If one modifies the specification of say module A, the lines to the left (and up) clearly identify the domain of impact within the set of higher level modules - the modules which rely on module A's definition. Changing module B's implementation, on the other hand, can be followed along the lines leading to its 'building blocks'. Figure 5 illustrates the 'limited domain' impact sets and clearly contrasts the new structure's simplicity against the

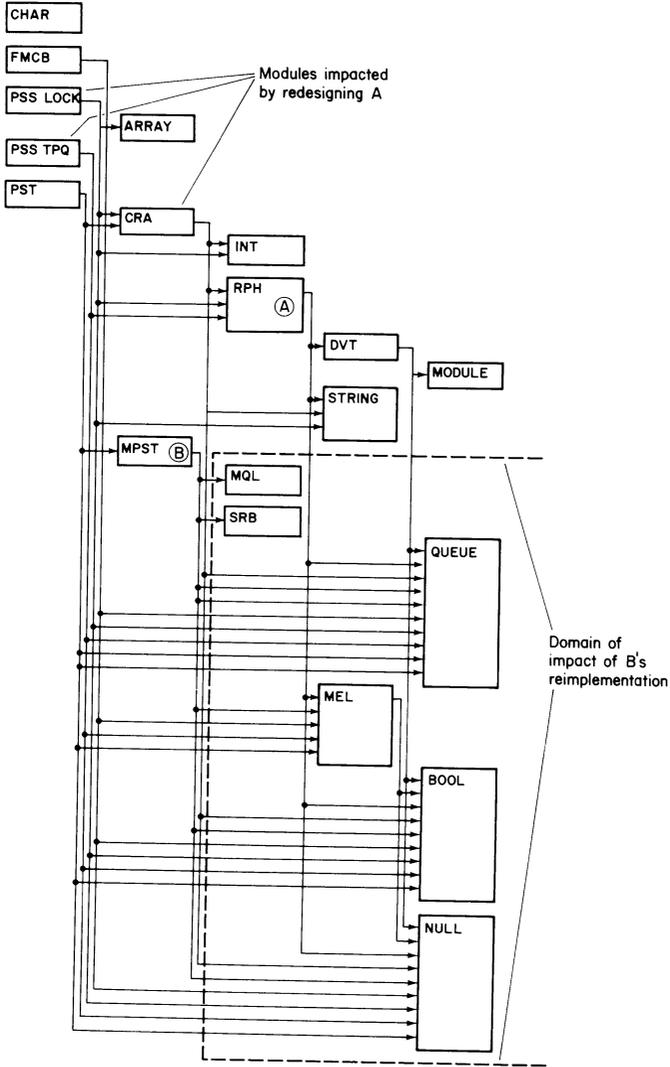


Figure 5

generality of the two-dimensional matrix with its rich connectivity.

We consider the external structure a machine aided design tool which is particularly well suited to the construction of large software systems, for which a human organisation is employed, in contrast to the informal development and modification process by small teams. In our own experience in redesigning VTAM, consequences of changes, even during the design process, are easier to follow, exploring alternatives is more rapid, and the compiler's typechecking superior to manual testing in catching bugs early. Our current efforts are now in the direction of enriching the external structure to contain formal specifications of capsules wherever possible (17.4).

4 Conclusions

Tools are useful for both the maintenance of existing systems and the design of new modifiable software. The experimental approach taken by our research team turned out to be productive since certain problems of design and maintenance can be understood and appreciated only by experiencing them. Also, the actual use of the newly developed tools led to their further refinement and consequent increase in usability and credibility for successful technology transfer.

The project's current emphasis is two-fold (17.4). First, modifiability must be evaluated by comparing the new version to the old. This we plan to do by employing two groups of programmers performing the same set of modifications, one group on the new and the other on the old version. Difficulty of the modification, the time it takes, and the performance of the resulting programs, will all be used in quantitative comparison. Second, we will extend both the language and the external structure, the former to facilitate concurrent processing and the latter to accommodate all formal and informal specification for the entire duration of a software's life cycle.

17.4 (Eds) *All references in this chapter to work in progress or planned wer valid at the time of writing in 1981, but should now be interpreted as work still needing to be undertaken.*

5 Acknowledgments

The project mentioned here was headed by Hamed Ellozy. He and the other members: Jerry Archibald, Carlo Evangelisti, Burt Leavenworth and Leigh Power, as well as many visitors from the United States and other universities abroad, have been doing the creative work which lead to the presented results.

CHAPTER 18

ON UNDERSTANDING LAWS, EVOLUTION AND CONSERVATION IN THE LARGE-PROGRAM LIFE CYCLE*

1 Large Programs

Various published papers ([LEH79] and its bibliography), have discussed the characteristics and dynamics of evolution of large programs and the laws that have emerged from the studies of Belady, Lehman, and others over the past 7 years. The main objective of the present contribution is to discuss one specific aspect of these laws - conservation. However, some general introductory remarks are desirable.

In the first place it must be stressed that the discussion here is limited to large programs. Until recently these were defined as programs of which at least some part has been concurrently implemented and/or maintained by at least two separately managed groups [LEH78a]. Such programs will certainly display the characteristics of largeness [BEL78]. They will, for example, inevitably have the property of variety; they will also be outside the intellectual grasp of any individual; above all, they will undergo a continuous process of maintenance and evolution, generally in somewhat undisciplined fashion.

The above definition is, however, not very satisfying. For one thing, programs not satisfying it may also display some or all of the other characteristics of largeness. Moreover, the definition tends to focus attention on management or sociological issues, whereas the fundamental concern here is with programming methodology and the engineering of software. In particular, one must seek to recognise and learn to control the circumstances that lead to the ill effects so often associated with largeness. Will not the adoption of appropriate attitudes, algorithms, methods, and programming techniques yield large programs that are well disciplined? Such questions are being addressed by our current research, which includes attempts to formulate more acceptable and useful definitions.

2 The Nature of Laws of Program Evolution

2.1 Their place within the spectrum of scientific laws

The evolution of large programs, software systems, is clearly not a natural process governed by immutable laws of nature. Changes to a program are neither initiated nor occur spontaneously. People do the work: amend or emend the requirements, the specification, the code, the documentation; repair the system; improve and enhance it. They do this in response to fault reports, user requests, business or legal requirements, managerial directives, or their own inspiration. Human thought and judgement play a decisive role in driving and executing the process that results from (and in) a seemingly continuous sequence of exogenous inputs (18.1).

Thus, we should not expect to discover laws of program evolution that yield the precision and predictability of the laws of physics [LEH76]. If laws governing large-program evolution can be formulated at all, they must certainly be weaker than even those formulated in the biological sciences, where regularity stems from the collective behaviour of

cellular systems that, even though alive, are nonintelligent, that is, are not influenced by conscious thought processes, at least at the level of human understanding. Program evolution should not even be expected to display the regularity that has been abstracted into laws in the social and economic sciences, for example, the Law of Supply and Demand. After all, the programming process is planned and controlled by an organisational and management structure that is sensitive and reactive to the demands, pressures, circumstances, and contingencies of each moment. Thus, superficially, it would seem reasonable to expect the program evolution process to be totally irregular, a reflection at each instance in time of the pressures of the moment.

One of the first and most surprising, yet most fundamental, results of our observations and of the resultant analysis of the dynamics of evolution of some eight programs ranging over a wide spectrum of implementation and usage environments, has been that this is not so. Regularities, trends, and patterns appear and dominate large-program evolution. The common

18.1 (*Orig*) We may regard the inputs as exogenous even though some of them will have been generated while, or as a result of, working on, or using, the system.

features and patterns of behaviour reflect common characteristics [BEL78] from which laws can be deduced; laws that, within the spectrum outlined, lie closer to those that describe the time behaviour of biological organisms than those that emerge from the study of socio-economic systems. Moreover, these laws find very practical application. They provide a basis for large-program life cycle management tools, as well as insight and understanding for improvement of the programming process. As we increasingly rely on the laws for guidance in the development of programming methodology and on a software engineering discipline with its techniques and tools, it becomes vital to develop also a deeper understanding of these laws and the fundamental phenomena or truths that they reflect.

2.2 The Underlying Cause of Regularity

Once the phenomena have been recognised, the mechanisms underlying them are not difficult to understand. As a totally unintelligent machine, the computer executing a program impacts its environment in a way that is precisely and completely determined by the code in association with any input data. The code is unforgiving; there is no room for logical error or imprecision. Thus any deviation from the required semantic and syntactic structure creates a need for corrective action. Good intentions, hopes of correctness, wishful thinking, even managerial edict cannot change the semantics of the code as written or its effect when executed. Nor can they a posteriori affect the relationship between the desires, needs, and requirements of users and the program specification as presented to the programmers; nor that between the specification and its implementation; nor between any of these and operational circumstances - the real world.

Additionally, the program and its documentation in all their versions - the system - has a damping effect analogous to an ever-increasing mass. It is precisely the freedom to implement changes or additions required for obtaining desired program behaviour which is increasingly constrained by existing accumulated code and documentation, past program application and behaviour, acquired habits, and implementer and user practices.

The development and implementation of change and of any subsequent corrective action is strongly influenced by the fact that, in itself, program code is also not malleable. Internal coupling, interconnections, and dependencies cause even changes that, superficially, appear localised, to impact

and modify the semantic consequences of code elsewhere in the program. Thus when changes are made to the code, deviations from absolute correctness will occur and unexpected side effects will appear; these are very likely to lead to a need for further corrective action. The more intensive the pressure for change, the higher the rate of its implementation. the larger the group of people involved, the more likely it is that maintenance must subsequently be diverted from progressive enhancement to repair and clean-up (that is to redesign, restructure and re-implement).

These dependencies may be viewed as feedback connections over the entire system and application processes and organisations. The resultant interplay of forces for change and expansion of the code on the one hand and the inertia of accumulated code, documentation, and habits on the other. and the interplay over the various processes and the organisational structures implementing them, are believed to be major factors in causing the observed regularity, determining its statistical characteristics and parameters.

These facts alone suffice to explain the consistency of the observations. Recognition of other factors merely strengthens belief in the reality of the phenomena. In particular. large programs are. as a rule, created within large organisations and for large numbers of users; otherwise they could not be economically justified or maintained. But size causes inertia, and inertia smoothes behaviour that might otherwise prove highly irregular. Moreover. the size and complexity of both the program and the application for which the program is intended mean that decisions take time (sometimes considerable) and large numbers of people to implement. Resultant delays provide exogenous pressures and endogenous opportunities for change. The overall circumstances and environment act as a filter. smoothing out the global consequences of individual decisions but, paradoxically. also adding the occasional random disturbance. They also act as an economic and social brake that inhibits or softens decisions that would have too drastic an impact. For example, large budgets can, in general, be neither suddenly terminated nor drastically increased; in practice they can only be changed by fractional amounts. Similarly, a work force cannot be instantaneously hired, retrained, relocated, or dismissed; at best a task force can be sent in, and can cause a local perturbation.

In summary. large-program creation and maintenance occur in an environment with many levels of arbitration, correction,

smoothing and feedback control. A large number of superficially independent (ie, almost random) inputs are concurrently and successively superimposed to yield time behaviour that may be statistically modelled (eg described by parameters that have normal distributions). Many, if not all, of the inputs arise from organisational checks and balances, from feedback often also involving users of the system. The feedbacks in general ensure long-term stability; negative feedback dominates. The alternative, of course, would be instability and disintegration of the system. The existence of regularity, and therefore of laws abstracting that regularity, becomes reasonable and understandable.

2.3 The Gross Nature of the Laws

The detailed behaviour of the programming process and of the system that is the object of process activity is the consequence of human decision and action. Specific individual events in the lifecycle of the system, the system development and maintenance process, cannot therefore be predicted more precisely than can the specific acts of participating or interacting individuals [LEH76]. Any laws can only relate to the gross (statistical) dynamics of large-program systems over a period of time, but as such they yield insight and understanding that should permit improvement of the programming process and advance the development of software engineering science and practice.

2.4 Feedback Consequences of Increasing Understanding of the Process

Increasing understanding of the dynamics of the large-program life cycle raises another problem: to what extent will the discovery and acquisition of knowledge and understanding of the laws that regulate the programming process, by an environment previously unaware of or insensitive to their existence, lead to changed behaviour and thus invalidation of the laws? How will managerial awareness of and conscious reaction to the laws affect the very nature of these laws? Since they reflect the joint behaviour of people, the laws are unlikely to immutable. Surely they may be expected to change as understanding of system behaviour increases [LEH76].

Space does not permit detailed discussion of this question. We merely assert that the present laws reflect deeply rooted aspects of human and organisational behaviour. Associated with the mechanistic forces that define, control, and execute

the automatic computational process. they are sufficiently fundamental to be treated as absolute, at least in our generation. As knowledge of them is permitted to impact the programming process, and as programming technology advances, the laws may require restatement or revision. Perhaps they will become irrelevant or obsolete. But for the time being, we must accept and learn to use them. To ignore them is foolish and costly.

3 The Laws

The following brief comments on the laws summarised in Table 1, are intended to expose some of the more fundamental truths that they reflect. These laws have been fully discussed in earlier publications ([LEH69] and its bibliography).

I *The Law of Continuing Change*

This first law reflects a phenomenon intrinsic to the very being of large programs. It arises, at least in part, from the fact that the world (in this case the computing environment) undergoes continuing change.

All programs are models of some part or aspect of, or process in the world. They must therefore be changed to keep pace with the needs and the potential of a changing environment. If they are not, the programs become progressively less relevant, useful, and cost effective.

Of course, all complex systems evolve. Living, social and artificial systems [SIM69] all respond to reactions and pressures from their environments by changes in operational pattern, function, and structure. Software is distinguished not by the fact that evolution occurs, but by the way in which it occurs.

The pressure for change with respect to any large program is felt almost daily. A widely held view is that the details of the desired change need 'only' be written down and then applied without further real effort (or so it would seem) to all instances of the system. As a consequence, changes are super-imposed (change upon change) in a *current embodiment*. This contrasts strongly with normal industrial practice where conceptual changes are inputs to a redesign and recreation process that ultimately produces a *new instance* of the system. Moreover, any repairs to software are a departure from the original conceived design and/or implementation rather than the replacement of a worn-out part. In addition

Table 1: Five Laws of Program Evolution

I CONTINUING CHANGE

A program that is used and that, as an implementation of its specification, reflects some other reality, undergoes continuing change or becomes progressively less useful. The change or decay process continues until it is judged more cost effective to replace the program with a recreated version.

II INCREASING COMPLEXITY

As an evolving program is continuously changed, its complexity, reflecting deteriorating structure, increases unless work is done to maintain it or reduce it.

III THE FUNDAMENTAL LAW
(of Program Evolution)

Program evolution is subject to a dynamics which makes the programming process, and hence measures of global project and system attributes, self-regulating with statistically determinable trends and invariances.

IV CONSERVATION OF ORGANISATION STABILITY
(Invariant Work Rated)

The global activity rate in a project supporting an evolving program is statistically invariant.

V CONSERVATION OF FAMILIARITY
(Perceived complexity)

The release content (changes, additions, deletions) of the successive releases of an evolving program is statistically invariant.

there is, in software. absolutely no decay or death process through which older parts of the system wear out and are replaced, or disintegrate and disappear out of the system. Removals, with replacements and additions, occur as a result of system-extraneous pressures and effort, and then *only* as the result of conscious and directed effort on the part of people.

The evolution of software differs from that of other systems in many other ways, but it is not our concern here to prove that software is different to or to state in detail how it is different. We ask the reader to accept that difference and then to ponder the practical implications.

These implications are, we assert, strongly influenced by the fact of continuing evolution, recognised and formalised by the first law. The causes of continuing change are seen as stemming, at least in part, from the continuing evolution of the operational environments, in combination with the 'soft' nature of programming technology. Hence changeability and all it implies must be accepted as a basic requirement for software systems. The degree to which it is achieved and maintained may make all the difference, in the development, application, and cost effectiveness of a system between success and failure, profitability and loss.

II The Law of Increasing Complexity

Our second law may be seen as an analogue of the second law of thermodynamics. More correctly, both of these laws should perhaps be viewed as descriptions of instances of a still more fundamental and natural phenomenon. In the case of software, the law is a consequence of the fact that a system is changed to improve its capabilities and to do so in a cost-effective manner. Specific change objectives develop from a consideration of factors that indicate immediate or measurable benefit. They are expressed in terms of performance targets, system resources required during execution, implementation resources, completion dates, fiscal objectives and constraints, and so on.

In cases with multiple objectives, it is generally impossible to meet all of them optimally. Hence the completed project and system must represent a compromise that results from judgments and decisions taken during the planning and implementation processes, often on the basis of time and group or management-local optimisation.

Structural maintenance is rarely mentioned in objectives. Being anti-regressive [LEH74], it yields no immediate or visible benefit but *merely (sic)* prevents deterioration. Thus structure, being excluded from stated project objectives, will inevitably suffer; each change will degrade the system a little more. The resultant accumulation of gradual degradation ultimately leads to the point where the system can no longer be cost effectively maintained and enhanced unless and until redesign and clean-up or re-implementation is undertaken and successfully completed.

The law suggests that large-program structure must not only be created but must also be maintained if decay is to be avoided or postponed. Planning and control of the maintenance and change process should seek to ensure the most cost-effective balance between functional and structural maintenance over the lifetime of the program. Models, methods and tools are required to facilitate achieving such balance.

III The Fundamental Law of Large-Program Evolution

This was previously called *the Law of Statistically Smooth Growth* [LEH78]. It expresses the observation already made above that large-program evolution does not simply reflect, at each instant and in each period, the decisions and actions of the people in the environment in which it is maintained and in which it is used. The law states that, at least in the current state of the art, there *exists* a dynamics whose characteristics are largely determined during the conception and early life of the system, of the maintenance process and of the maintenance organisation. The characteristics of this dynamics increasingly determines the gross trends of the maintenance and enhancement process. System, project and organisational *history* play an important role in the program evolution process, while feedback provides a self-stabilising control process, itself evolving. Thus cyclic effects emerge, though not necessarily with pure periods.

This law is particularly important in guiding our understanding of the software creation and maintenance process. However its tacit acceptance (for the time being) also helps the manager and the planner to remain realistic. We are not free to set and achieve arbitrary design, performance, and work targets [BR075]. Project constraints are at present not all under our control. Thus one must accept any limits they imply until they can be or have been changed. Moreover, the law implies that models of large-program evolution can be created and exploited as planning and control tools.

IV *The Law Of Organisational Stability*

This was previously referred to as the Law of Invariant Work Rate [LEH78]. It reflects the fact that, in general, human organisations seek to achieve and maintain stability with stable growth. As suggested above, sudden substantial changes in such managerial parameters as staffing, budget allocations, manufacturing levels and product types are avoided; as a rule, such changes are not even possible. A variety of managerial, union, and governmental checks, balances, and controls ensure smooth overall progress to the ever changing, ever-distant objective of the organisation (or its eventual collapse). In addition, the fourth law also reflects the organisational response to the limitation that will be shown underlies the fifth law.

With hindsight it becomes clear that the discovery of an invariant activity measure (statistically invariant, as when its parameters are always normally distributed with constant mean and variance) could have been anticipated. What is not really understood is why, in large-program maintenance projects, measures of work input rate should be the quantities to display such invariance (18.2). However, the fact remains that for the systems observed, the count of modules changed (handled) or changes made per unit of time, as averaged over each release interval, has been statistically invariant over the period of observation. The limitations implied by this invariance can only be temporarily overcome. If they need to be overstepped, the consequences should be identified and must be accepted.

V *The Law of Conservation of Familiarity (Perceived Complexity)*

In [LEH78] this law was referred to as the Law of Incremental Growth Limits. Its discovery was based on data from three systems, each of which was made available to users release by release. In each case the incremental growth of the program varied widely from one release to the next, but the average

18.2 (Eds) *It really is not that difficult to explain. At the organisational level feedback with production and activity targets and budget control is guided by organisational perceptions and goals and by executive decision. These, in turn, are couched in terms of human and other implementation resources, production capability which is reflected in work targets and work rates. QED.*

over a relatively large number of releases remained remarkably constant; that is, a high-growth release would tend to be followed by one with little or not growth, or even by system shrinkage; or two releases, each displaying near average growth, would be followed by one with only slight growth. Moreover, releases for which the net growth exceeded about twice the average proved to be minor disasters (or major ones, depending on the degree of excess) with poor performance, poor reliability, high fault rates, and cost and time over-runs.

The evidence suggests that initial release quality is a nonlinear function of the incremental growth. From a more complete phenomenological analysis along the lines outlined below, it is hypothesised that quality is exponentially related to the *release content*, that is, to the amount of change implemented in the release.

It should perhaps be added that at this time no *precise* way is known how to define or measure release content per se, or how to take into account the size, complexity, and inter-relationships of system and code changes, additions, and deletions. It is not even clear that a metric can be found. If it can, then such a universal measure must also be sensitive to the characteristics of the systems and the environments involved in or affected by the changes.

The absence of adequate definitions and measures is no reason for ignoring observed phenomena and their implications. The gradual clarification and evolution of concepts, definitions, and measures is fundamental to the very nature of the phenomenological approach we have adopted, an approach that is considered essential for significant progress in mastering the problems of software engineering. One first observes and measures some phenomenon, then seeks models, interpretations, and explanations in more fundamental terms; subsequently, one can seek measures and devise experiments that confirm, reject, modify, and/or extend the original hypotheses, interpretations, and explanations; and so on.

4 Interpretation of the Fifth Law

4.1 Change and Refamiliarisation

The phenomenon abstracted by the fifth law was detected at a very early stage of the evolution dynamics studies and was featured in the earliest model [BEL71b]. It has been applied as a planning and control parameter for a number of years.

The explanation, however, has only recently become apparent. The release process has always been understood as fulfilling a stabilisation role [BEL71b]. Once a large program is in general use, its code and documentation are normally in a state of flux. A fault is fixed locally; in other installations it is perhaps fixed differently or not at all. Minor or major changes and local adaptations are made. Code is changed without a corresponding change to documentation. Documentation is changed to correspond to observed behaviour without a full and detailed analysis of the precise semantics of the code, within the context of the total system, under all possible environmental conditions. Only at the moment of release does there exist an authoritative version of the program, the code, and its documentation. Even this may include multiple versions of modules, say, for more or less clearly defined alternative situations.

Some time after the release of a program or program version, each designer, implementer, tester, salesman, and user that has been exposed to or worked with the system will have become thoroughly conversant with, at least, those of its attributes and characteristics that are considered at all relevant. The resultant familiarity will have bred some degree of relaxation, of ability to work with the program in order to accomplish specific objectives. The program will be manipulated without uncertainty or concern and used without (apparent) need for concentrated thought. External perception of a program's intrinsic complexity will be at a minimum. For people working consistently on or with the program, its perceived complexity may be said to approach zero.

As changes are introduced, as the new release is gradually created and becomes available, new and unfamiliar code appears. The program behaves differently in execution, in its interaction with and impact on the environment. Pagination in the previously familiar documentation has changed and any need for reference entails a major search. The system has become uncomfortably unfamiliar, the degree of unfamiliarity depending on the magnitude and extent of the change.

A major intellectual effort is now required by each person involved before any completely successful and cost-effective interaction with the new system can occur. The system has suddenly become strange. Its perceived complexity is high.

Even those who participated in the preparation of the new release will normally have been included directly with only a small part of the changes, a small portion of the system. They too must now learn to understand the new system in its totality. Moreover, until the complete system is available, all acquisition of knowledge and understanding of the changes and of the new system must be based on reading of code and documentation text, or on partial execution of system components on test cases or system models. At least some part of system-internal interactions or dependencies will be absent in such an environment. Only with final integration of the new system does the full executable program become available. Therefore, when the release content exceeds some critical amount, only operational experience with the complete system can restore the degree of knowledge and familiarity, the global viewpoint, that is essential for subsequent cost-effective maintenance, enhancement, and exploitation of the large program.

Thus, in general, at the moment of release or shortly before that time a major learning effort will begin. This must involve all those associated with the system, not just the users. All changes and additions must be identified, understood, and *experienced*, their significance appreciated within the operational context of the total system. Once this has been done, the old degree of comfort with the system will return and its perceived complexity once again will approach zero; the level of familiarity has been restored.

The amount of work that must be invested, the intellectual effort required to achieve this, depends among other factors, on the attitudes of people, on the organisation, and on the number, magnitude, and complexity of changes introduced. Because changes to the system interact with one another, because changes implemented in the same release must be understood in the context of all other changes being concurrently implemented as well as in the context of the unchanged parts of the system and of past and future applications, the relationship between the release content and the amount of intellectual effort needed to absorb the changes introduced by a new release fully is at least quadratic, probably exponential. But whatever the precise relationship between the difficulty of restoring familiarity with the program and the magnitude of the release content, it will be of the general form indicate in in Figure 1.

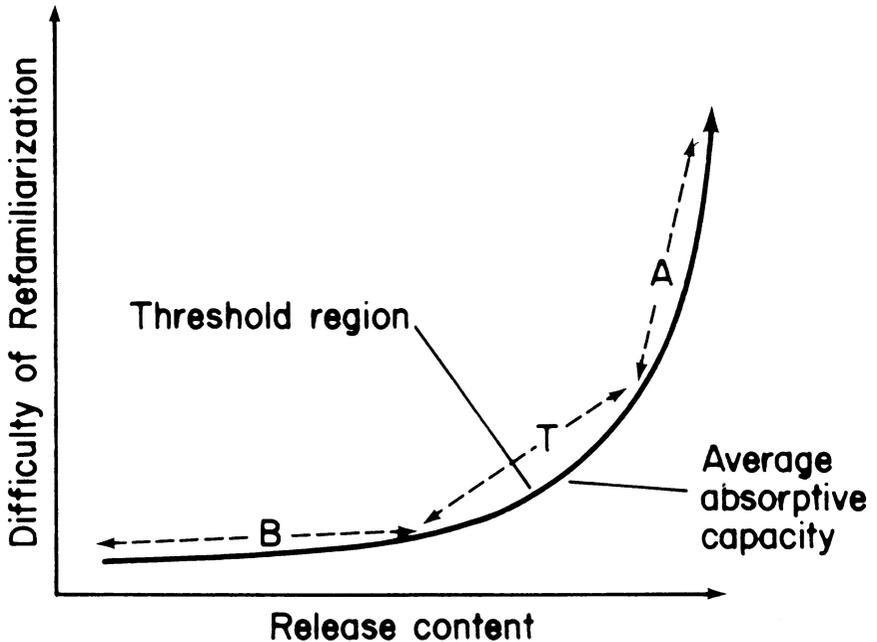


Figure 1: Difficulty - Release-content relationship.
A, above threshold; B, below threshold

The axes of the curve are not calibrated since at present neither concept nor suitable measures are well defined. The concept of 'difficulty' introduced here, relates to that of Norden and Putnam [NOR77]; [PUT77]. They, however, are concerned with difficulty of implementation, whereas the present concern is with understanding the changes within the context of the total system; their implications with regard to its operational behaviour. Although related, since one cannot (should not?) implement without understanding, the concepts are clearly not identical.

4.2 The Averaging of Ability Through Human Interactions

It must be left to the future to identify or define measures and provide an improved formulation of the fifth law. Meanwhile the basic concepts must be clarified, and understanding of, at least the phenomenology increased. A basis for ultimate formalisation is thereby provided.

Everyone's ability to master a new or changed object is limited, though people clearly differ in their ability to absorb new knowledge (eg to achieve full understanding of the changed program). Thus the impact of changes will vary from person to person according to many factors that will include, but are not limited to, their learning ability and absorptive capacity. For a given large program with which many people are inevitably involved, the direct and indirect costs of familiarisation (delays incurred, mistakes made, destructuring, etc) relate to the average ability of all the people involved. This average will not change significantly with time, or, with the detailed composition of the group.

In the implementation environment. for example, although the above-average person will regain mastery more quickly, make fewer mistakes, and achieve a temporary advantage (which might lead to promotion or transferral to another project), the below average person will fall behind, perhaps lose contact, make more mistakes, and do more damage. This person may well be reassigned to a less demanding role, one with less impact, or even be fired. But the damage will have been done; others will have to do additional work or apply corrective action. Since hiring policies are related to the already established make-up of the project, the average capacity to understand will, at best, remain unchanged; more probably it will decline [LEH78].

In the application and user environment. the capable individual will master the changes relatively quickly and carry on with assigned responsibilities, experience minimal perturbation. and cause no impact on others. The less capable, on the other hand, will have to discuss with others their difficulties in fully appreciating the changes. They will even misinterpret documentation or system behaviour and report difficulties or faults that are in fact, non-existent. Such discussions or reports will cause delays or disruptions in the project, and may even lead to erroneous repair. Once again the presence of persons with greater than average difficulty in refamiliarisation has an impact that ranges beyond the immediate bound of responsibility of these

individuals. It results in an *overall* slow-down in the return to normalcy after change, the time required being determined by *average* ability.

For different organisations, systems, structures, methodologies, and processes, the average level will, of course, be different. This implies that any model will contain parameters reflecting exogenous variables. These variables point to a potential for improving the average absorption level. Once the phenomenon, the organisation, and the programming process are understood.

4.3 Conservation of Familiarity and Statistically Invariant Release Content

Given the above insights firstly into the increased difficulty of understanding changes and their implications as release content increases and secondly into the mechanism of the slowdown of both utilisation and further evolution as system structure deteriorates, the number of faults increases, documentation lags, and performance declines, we are now in a position to appreciate the fifth law.

If the release content, the magnitude of changes and/or the incremental growth, is less than some threshold region T (Figure 1), the integration and operational installation of the new system should be fairly straightforward. No major problems should be experienced in mastering the new release; it may well be that the change may be absorbed and familiarity restored without actual operational disturbance.

The very ease of the re-familiarisation process in conjunction with the never-ending search for productivity growth will, however, create a managerial climate in which more ambitious releases that will challenge organisational capability and may flout its natural parameters, will be attempted. Pressures are created that tend to move subsequent releases from B region into the threshold T region.

When the release content lies in T (which may not be precisely delineable), quality, performance, completion, and installation problems are to be expected. Slippage and cost overrun will probably occur. A subsequent release may be required to clean up the system and restore it to a state that permits further cost-effective evolution. This experience will certainly not encourage management to demand an increase in release content. The next release will tend to be in the same threshold region or even below it.

of the T region and moves into A, serious problems will be encountered. Slippage and cost overrun must occur unless plans take account of the greatly increased difficulties that will be experienced. If not properly planned, such an attempt may lead to the effective collapse of the system or, as observed in at least two instances, to an effect that we have termed *system fission*. Since only release of the system to end users and to the developers provides full exposure, even when adequate resources and time have been provided, such a release will still have to be followed by a restoration or clean-up release. This results in one or more successor releases in the B region of the characteristic curve of Figure 1.

It was the repeated observation of the above patterns of release behaviour that suggested the analysis and led to the insights summarised in the preceding paragraphs. Our analysis suggests that the consequences of feedback in the process, in conjunction with nonlinear characteristics indicated in Figure 1, lead (over several releases) to stabilisation of release content in or just below the threshold region. No attempt has yet been made to create an analytic model of this phenomenon, but it should not prove too difficult to build and validate [W0079a].

The fifth law abstracts both the observations and their interpretation, including the emergence of invariant average incremental growth of release content. The latter is also a consequence of the additional exogenous pressure for accelerated functional growth that is characteristic of large-program applications and, in general, of organisational environments. Once again the law suggests that managers and planners take note of project and system invariances; when formulating plans, they must respect the limitations the invariances imply or accept the inevitable consequences.

5 Final Comments

The first recognition of the laws discussed was based entirely on an examination and analysis of data from a variety of programs and systems, both large and not so large. To make the transition from phenomenology to science, however, the laws, once formulated, must be examined in their own right. The laws of large-program development and evolution are now beginning to be understood in this way. They are seen to express very basic attributes of computing, of the programming development, maintenance and usage

processes. of programs themselves, and of the organisations and environments in which these activities are carried out.

Once this interpretation of the laws in terms of more fundamental phenomena has been achieved, the old data must be re-examined and new information examined in the light of the laws *as understood*. Deviations must be explained and interpreted; contradictions may require reformulation and re-interpretation of a law, or even its rejection.

There is, of course, nothing new in these comments: they form the very basis of the scientific method. They are added here, however, to assert the belief that the laws as formulated have been substantiated by experience and by experimental data to the point where they can stand in their own right until accumulating evidence and developing insight and understanding demand their change - or until we can so change system structure, process methodology and characteristics, and programmer and user practice and habits, that the laws as formulated no longer apply.

CHAPTER 19

PROGRAMS, LIFE CYCLES AND LAWS OF SOFTWARE EVOLUTION*

1 Background

1.1 The nature of the problem

The 1977 US expenditure on programming is estimated to have exceeded \$50 billion and may have been as high as \$100 billion. This figure, more than 3% of the US GNP for that year, is itself an awesome figure. It has increased ever since in real terms and will continue to do so as the micro-processor finds ever wider application. Programming effectiveness is clearly a significant contributor to national economic health. Even small percentage improvements in productivity can make significant financial impact. The potential for saving is large.

Economic considerations are, however, not necessarily the main cause of widespread concern. As computers play an ever larger role in society and the life of the individual, it becomes more and more critical to be able to create and maintain effective, cost-effective and timely software. For more than two decades, however, the programming fraternity, and through them the computer-user community, has faced serious problems in achieving this [GOL73]. As the application of micro-processors extends ever deeper into the fabric of society the problems will be compounded unless very basic solutions are found and developed.

1.2 Programming

The early fifties had been a pioneering period in programming. The sheer ecstasy of instructing a machine, step by step, to achieve automatic computation at speeds previously undreamed of completely hid the intellectually unsatisfying aspects of programming; the lack of a *guiding theory* and *discipline*; the largely *hit* or *miss* nature of the process through which an acceptable program was finally achieved; the ever present uncertainty about the *accuracy*, even the *validity*, of the final result.

More immediately, penetration of the computer into the academic, industrial and commercial worlds led to serious problems in the provision and upkeep of satisfactory programs. It also yielded new insights. Programming as then practised required breakdown of a problem to be solved into steps far more detailed than those in terms of which people thought about it and its solution. The manual generation of programs at this low level was tedious and error prone for those whose primary concern was the result; for whom programming was a means to an end and not an end in itself. This could not be the basis for widespread computer application.

It was, however, observed that such programs contained instruction sequences that occurred again and again. These sequences represented primitive concepts in the programmer's view of the problem. Surely these at least could be mechanically generated. Thus there was born the concept of high level, problem oriented, languages and automatic program compilation. Starting with Fortran, then Cobol, Algol, Algol 68, PL/1 and countless non-mainstream languages, most recently Pascal and Ada, languages were created to simplify the development of computer applications.

These languages did not just raise the level of detail to which programmers had to develop their view of the automated problem-solving process. They also removed some of the burden of procedural organisation, resource allocation and scheduling, burdens which were further reduced through the development of operating systems and job control languages.

Above all, however, the high-level language trend permitted a fundamental shift in attitude, a shift whose potential was not always recognised or exploited. To the discerning, at least, it became clear that it was not the programmer's main responsibility to instruct a machine by defining a step-by-step computational process. His task was to state an algorithm that correctly and unambiguously defines a mechanical procedure for obtaining a solution to a given problem [LEH80], [LEH79]. The transformation of this into executable and efficient code sequences could be more safely entrusted to automatic mechanisms. The objective of language design was to facilitate that task.

Languages had become a major tool in the hands of the programmer. Like all tools they sought to reduce the manual effort of the worker and at the same time improve the quality of his work. They permitted and encouraged concentration on the intellectual tasks which are the real province of the

human mind and skill; in this case defining both the problem and an algorithm for its solution. Thus ever since the search for better languages and for improving methods for their use, has continued [WUL77].

There are those who believe that the development of *programming* methodology, high level languages and associated concepts, is by far the most important step for successful computer usage. That may well be, but it is by no means sufficiently. There exists a need for additional methods and tools, one that arises primarily from program maintenance.

1.3 Program Maintenance

The sheer level of programming and programming-related activity makes its disciplining important. But a second statistic carries an equally significant message. Of the total US expenditure for 1977, some 70% was spent on program *maintenance* and only about 30% of program *development*. Even higher ratios of maintenance to development costs have been reported for a variety of environments and the ratio may be accepted as characteristic of the state of the art in the software community, at least in the developed countries of the western world.

Some clarification is, however, necessary. For software the term *maintenance* is generally used to describe *all* changes made to a program after its first installation. It therefore differs significantly from more general usage which is concerned with *restoration* of a system or system component to its *former* state. Deterioration that has occurred as a result of usage or with the passage of time, is corrected by repair or replacement. But software does not deteriorate spontaneously or by interaction with its operational environment. Programs do *not* suffer from wear and tear, corrosion or pollution. They do not change unless and until *people* change them, and this is done whenever the current behaviour of a program in execution is found to be wrong, inappropriate or too restricted. So called repair actually involves changes *away* from the previous implementation. Faults being corrected during maintenance can originate in any phase of the program *life-cycle*.

Moreover, in hardware systems major changes to a product are generally achieved by redesign, retooling and the construction of a new instance. With programs, the code itself is continually being altered and extended to implement a sequence of *improvements* and *adaptations* to a changing

environment. New capability, often not recognised during the earlier life of the system is superimposed on an existing structure without redesign of the system as a whole. The program becomes a quicksand of ever-changing code, documentation and structure.

Since the term software maintenance covers such a wide range of activities, the very high ratio of maintenance to development cost does not necessarily have to be deprecated. It will, in fact, be suggested that the need for continuing change is *intrinsic* to the nature of computer usage. Thus the question raised by the high cost of maintenance is not exclusively how to control and reduce that cost by avoiding errors or by detecting them earlier in the development and usage cycle. The *unit cost of change* must initially be made as low as possible, and its growth, as the system ages, minimised. Programs must be made more *alterable*, and the alterability *maintained* throughout their life-time. The change process itself must be planned and controlled. Assessment of the economic viability of a program must include total life-time costs and their life-cycle *distribution*, and not based exclusively on the initial development costs. We must be concerned with the cost and effectiveness of the life-cycle process itself and not just that of its product.

The remainder of this paper examines the nature of this continuing change process and the support it demands. Methods and techniques that have evolved to cope with the associated problems are discussed elsewhere [BEL80].

The opening paragraph highlighted the high cost of software and software maintenance. The economic benefit and potential of the application of computers is, however, so high that present expenditure levels may well be acceptable, at least for certain classes of programs. But we must be concerned with the fact that *correctness, performance, capability, quality in general*, cannot at present be designed and built into a program *ab initio*. They can only be achieved by gradual refinement. Moreover, when needed or desirable changes are authorised, they can usually not be implemented on a time scale fixed by external need. *Responsiveness* is poor. And as mankind relies more and more, collectively and individually, on software that controls computers that in turn guide society, it becomes crucial that people remain in ultimate control of the change and adaptation process. To achieve this requires insight, theory, models, methods, tools; a discipline. That is what software engineering is all about [BOE76], [TUR78], [BOE78].

2 Programs as Models

2.1 Programs

Program Evolution Dynamics [BEL78 and its bibliography] and the laws [LEH74], [LEH80], [LEH79], [LEH80a], discussed in the next section have always been associated with a concept of *largeness*, implying a classification into large and non-large programs. Great difficulty has, however, been experienced in defining these classes. Even the definition that was finally adopted [LEH80], [LEH79] does not satisfy since it is based on management rather than programmatic concepts and on sufficiency but not necessity.

Recent discussions [TUR79], have produced a more satisfying classification. This follows from recognition of the fact that, at the very least, any program is a model of a model of .. a model of some portion of the world or of some universe of discourse. The classification categorises programs into three classes, S, P and E. Since programs considered large by our previous definition will generally be of class E, the new classification represents a broadening and firming of the previous viewpoint.

2.2 S-Programs

S-programs are programs whose function is formally defined by and derivable from a *specification*. It is the programming form from which most advanced programming methodology and related techniques derive, and to which they directly relate. We shall suggest that as programming methodology evolves still further, all large programs (software systems) will be constructed as structures of S-Programs.

A specific problem is stated: lowest common multiple of two integers; function evaluation in a specified domain; Eight Queens; Dining Philosophers; generation of a rectangle of a size within given limits on a specific VDU type. Each such problem relates to its universe of discourse. It may even relate directly and primarily to the external world, but be completely definable as is the travelling salesman problem.

As suggested by Figure 1, the specification, as a formal definition of the problem, directs and controls the programmer in his creation of the program that defines the desired solution. Correct solution of the problem as *stated* in terms of the programming language being used, becomes the programmer's sole concern. At most, questions of elegance or efficiency may also creep in.

The problem statement, the program and the solution when obtained may relate to an external world. But it is a casual, non-causal, relationship that defines the area of intrest. If that changes the problem may be re-defined. But then it requires a *new program* for its solution. It may be possible and time saving to derive the new program from the old. But it is a *different* program that defines a solution to a *different* problem.

When this view can be legitimately taken the resultant program is conceptually static. One may change it to improve its clarity or its elegance, to decrease resource usage when the program is executed, even to increase confidence in its correctness. But any such changes must not effect the mapping between input and output that the program defines and that it achieves in execution. The output remains invariant even under program transformation [BAU77], [DAR79]. Whenever program text has been changed it must be shown that either the input-output relationship remains unchanged, or that the new program satisfies a new specification defining a solution to a new problem. We return to the problem of correctness-proving in section 2.4.

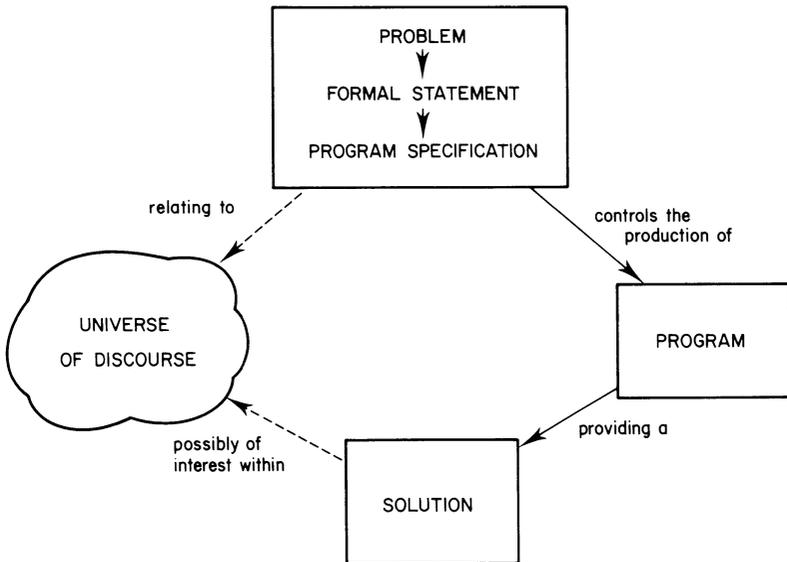


Figure 1: S-Programs

2.3 P-Programs

Consider a program to play chess. Its universe of discourse is largely defined and the program may, in theory, be completely specified in terms of the rules of chess and procedure rules. The latter must indicate how the program is to analyse the state of the game and determine possible moves. It must also provide a decision rule to select a next move. The procedure might, for example, be to form the tree of all games that may develop from any current state and adopt an evaluation strategy to select the next move. Such a definition, while complete, is naive, since the tree structure at any given stage is simply too large, by many orders of magnitude, to be developed or to be scanned in feasible time. Thus the chess program must introduce approximation to achieve practicality, judged as it begins to be used, by its performance in actual games.

A further example of a problem that can be precisely formulated but whose solution must inevitably reflect an approximation of the real world is found in weather prediction. In theory, global weather can be modelled as accurately as desired, by a set of hydrodynamic equations. In the actual world of weather prediction, approximate solutions of modified equations are compared with the weather events and patterns that occur. The results of such comparisons are interpreted and used to improve the technology of prediction to yield ever more usable programs.

Finally consider the travelling salesman problem as it arises in practice, for example from a desire to optimise continuously in some vaguely defined fashion the travel schedule of salesmen picking up goods from warehouses and visiting clients. The required solution can be based on known solutions to the classical problem. It must also involve value judgements relating, for example, to cost, time, schedules, timetables, and even salesmens' idiosyncracies.

The problem statement can now, in general, not be precise. It is an abstraction of a real world situation, containing uncertainties, arbitrary criteria, continuous variables. To some extent it must reflect the personal viewpoint of the analyst. The problem statement and outline solution approximate to the real world.

Programs such as these we term P-programs (real world *problem* solution). The process of creating such programs is modelled by Figure 2. Despite the fact that the problem to

be solved can be precisely defined, the acceptability of a solution is determined by the environment in which it is embedded. The solution obtained will be evaluated by comparison with the real environment. That is, the critical difference between S and P-programs is expressed by the comparison cloud in Figure 2. In S-programs judgements about the correctness, and therefore the value, of the programs relate by definition *only to its specification*, the problem statement that the latter reflects. In P-programs the concern is not centered on the problem statement, but on the value and validity of the solution obtained *in its real-world* context. Differences between data derived from observation and from computation may cause changes in the world view, the problem perception, its formulation, the model, the program specification and/or the program implementation. Whatever the source of the difference, ultimately it causes the program, its documentation or both to be changed. And the effect or impact of such change cannot be eliminated by declaring the problem a *new* problem, for the real problem has always been as now perceived. It is the perception of users, analysts and/or programmers that has changed.

The model of Figure 2 shows the intrinsic feedback loop that is present in the P-situation. Program changes stem from an explicit or implicit comparison of program output with real-world data that directly or indirectly form part of the input to the system-defining abstraction. Feedback implies a possibility of instability experienced as a loss of effective management control. Control may be retained by controlling the 'Gain/Delay' characteristics of the change-analysis, development and application process. Amongst other factors, gain is here related to user and organisational benefit deriving from change implementation in relation to the cost of the *total* work involved over the system life. The meaning and significance of delay is self evident. Release of the program to the end users and its execution in the user environment plays a vital role. It facilitates control by offering opportunities for discrepancy evaluation, change design, benefit-assessment and the planning of content and timing (section 4). Given an awareness of the situation and good control of the process, stability with ultimate availability of a satisfactory program, can be achieved, though several cycles through the loops may well be required.

Unfortunately, however, there is another fact of life that needs to be considered. Dissatisfaction will arise not only because information received from the program is incomplete or incorrect, or because the original model was less than

perfect. These are imperfections that can be overcome given time and care. But the world too changes and such changes result in additional pressure for change. Thus P-programs are very likely to undergo never-ending change or to become steadily less and less effective and cost effective.

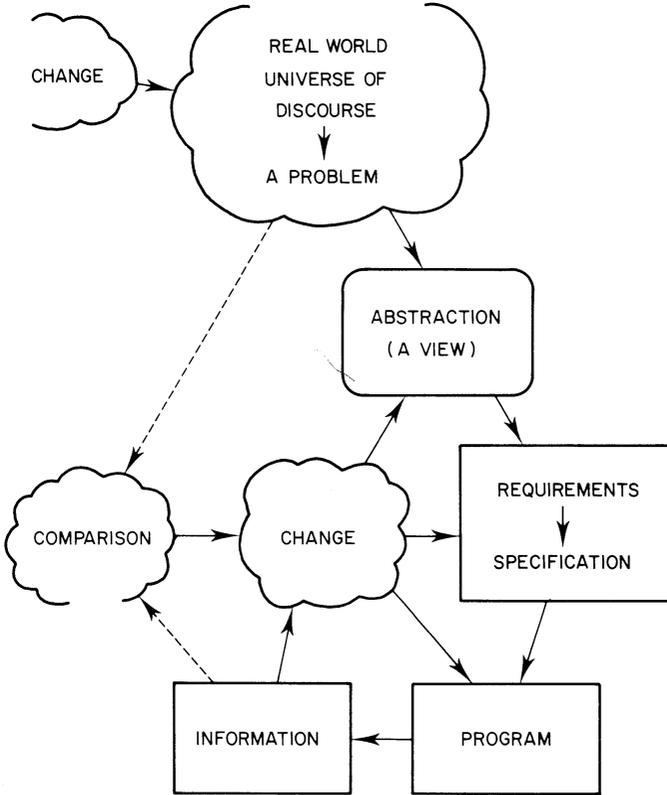


Figure 2: P-Programs

2.4 E-Programs

The third class, E-programs, are inherently even more change prone. They are programs that mechanise a human or societal activity and that are, therefore, embedded in the application.

Consider again the travelling salesman problem but in a situation where several persons are continuously en route, carrying products that change rapidly in value as a function of both time and location, and with the pattern of demand also changing continuously. One will inevitably be tempted to see this situation as an application in which the system is to act as a continuous dispatcher, dynamically controlling journeys and calls for each individual. The objective will be to maximise profit, minimise loss, expedite deliveries, maintain customer satisfaction or achieve some combination of such success criteria. How does this situation differ from that discussed previously?

Activation of the program and its associated system, radio links to the salesmen for example, changes the very nature of the problem to be solved. *The program has become a part of the world it models*; it is embedded in it. Conceptually, at least the program as a model contains elements that model itself, the consequences of its execution.

The situation is depicted in Figures 3 and 4. Even before program execution and its in the operational environment, the E-situation contains an intrinsic feedback loop as in Figure 3. Analysis of the application to determine requirements, specification, design, implementation now all involve extrapolation, prediction of the consequences of system introduction and the resultant ever increasing potential for application and system evolution. Prediction inevitably involves opinion and judgement. In general, several views of the situation will be combined to yield the model, the system specification and, ultimately, a program. Once the program is completed and begins to be used, questions of correctness, appropriateness and satisfaction arise as in Figure 4 and inevitably lead to additional pressure for change.

Examples of E-programs abound, computer operating systems, air-traffic control, stock control. In all cases the behaviour of the application system, the demands on the user and the support required will depend on program characteristics as experienced by the users. As they become familiar with a system whose design and attributes depend, at

least in part, on user attitudes and practice before system installation, users will modify their behaviour to minimise effort or maximise effectiveness. Inevitably this leads to pressure for system change. In addition, system exogenous pressures will also cause changes in the application environment within which the system operates and the program executes. New hardware will be introduced, traffic patterns and demand change, technology advance and society itself evolve. Moreover the nature and rate of this evolution will be markedly influenced by program characteristics, with a new release at intervals ranging from one month to two years, say. Unlike other artificial systems [SIM69] where, relative to the life-cycle of process participants, change is occasional, here it appears continually. The pressure for change is built in. It is due to the feedback-linked process that converts system concept into an operating application. It is intrinsic to the nature of computing systems and the way they are used.

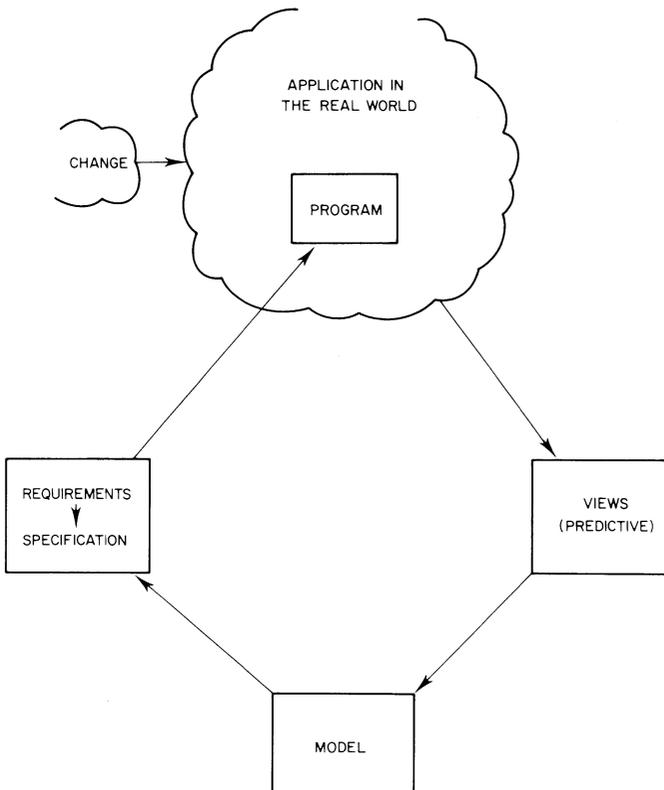


Figure 3: E-Programs - the basic cycle

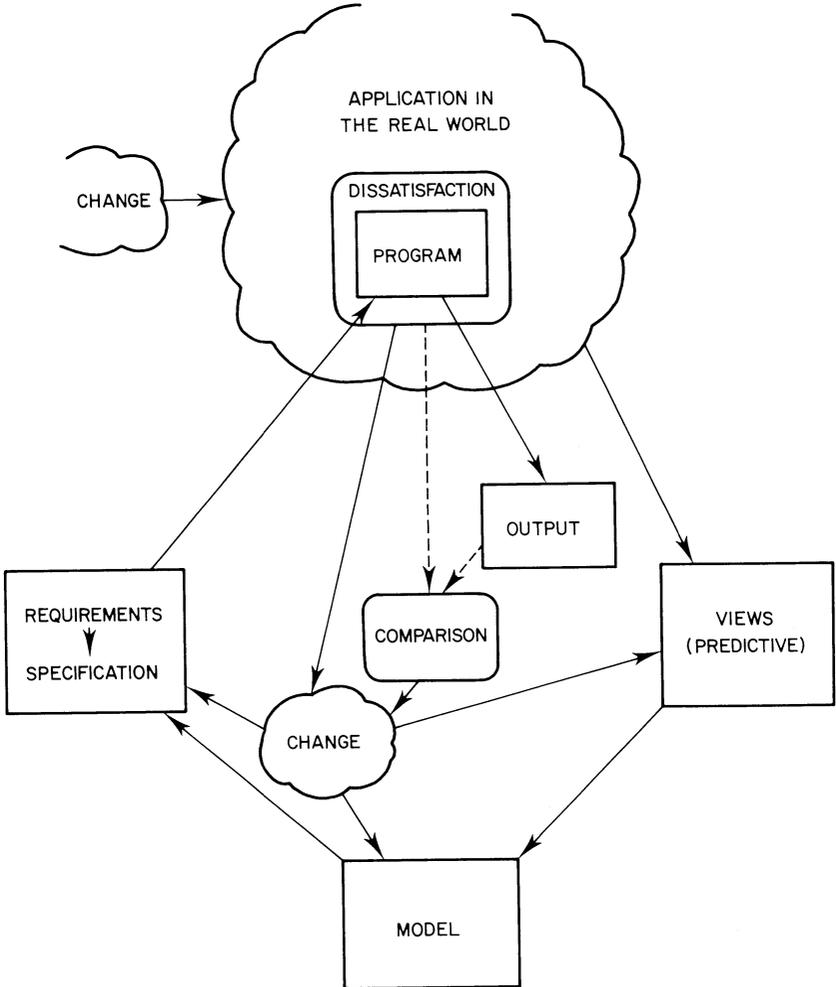


Figure 4: E-Programs

P and E programs are clearly closely related. They differ from S-programs in that they represent a computer *application* in the real world. We shall refer to members of the union of the P and E classes as A-type programs.

2.5 Program Correctness

The first consequence of the SPE program classification is a clarification of the concepts of program correctness. The meaning, reality, and significance of related concepts has recently been examined at great length [DEM79]; [HOA79]. Many of the viewpoints and differences expressed in that discussion become reconcilable or irrelevant under an adequate program classification scheme.

For the SPE scheme, the concept of verification takes on significantly different meanings for the S and A classes. If a completely specified problem is computable, its specification may be taken as the starting point for the creation of an S-program. In principle a logically connected sequence of statements that demonstrates the validity of the program as a solution of the specified problem can always be found. Detailed inspection of and reasoning about the code may itself produce the conviction that the program satisfies the specification completely. A true proof must satisfy the accepted standards of mathematics. Even when the correctness argument is expressed in mathematical terms, a lengthy or complex chain of reasoning may be difficult to understand, the proof sequence may even contain an error. But that does not invalidate the concept of program correctness proving, merely this instance of its application.

We cannot discuss here the range of S-programs for which proving is a practical or a valuable technique, the range of applicability of constructive methods for simultaneous construction of a program and its proof [DIJ68b]; [HOA69]; whether confidence in the validity of an S-program can always be increased by a proof. We simply note that since, by definition, the sole criterion of correctness of an S-program is the satisfaction of its specification, (correct) S-programs are always demonstrably correct.

This is not purely philosophical observation. Many important components of a large program, mathematical procedures for example, in conjunction with specified interface rules (calling and output) are certainly S-type. It becomes part of the design process to recognise such potential constituents during the partitioning process and to specify and

implement them accordingly. In fact it will be postulated in section 2.6 that an A-program may always be partitioned and structured so that *all* its elements are S-programs. If this is indeed true, no *individual* programmer should *ever* be permitted to begin programming until his task has been defined and delimited by a complete specification, against which his completed program can be validated.

In summary, for S-programs it is meaningful to talk about correctness. There is a program specification that totally determines the accepted requirements. We may verify that the program satisfies the specification and thereby prove the program correct. And once correct, always correct, since by definition the program cannot change.

For an A-program on the other hand, validity depends on human assessment of its effectiveness in the intended application. Correctness and proof of correctness of the program as a whole, are, in general, irrelevant in that it may be formally correct but useless or incorrect in not satisfying some stated specification, yet quite usable. Formal techniques of representation and proof have a place in the universe of A-programs but their role changes. It is the detailed *behaviour* of the program *under operational conditions* that is of concern.

Parts of the program that can be completely specified should be demonstrably correct. But the environment cannot be completely described without abstraction, and therefore, approximation. Hence absolute correctness of the program *as a whole* is not the real issue. It is the usability of the program and the *relevance* of its output in a changing world that must be the main concern.

2.6 Program Structures and Structural Elements

The classification created above relates to program entities. A program will, in general, consist of many parts variously referred to as sub-systems, components, modules, procedures, routines. The terms are, of course, not synonymous but carry imputations of functional identity, level, size and so on.

The literature discusses criteria [PAR72] and techniques [WIR71], [DIJ72], [JAC75] for partitioning systems into such elements. Related design methods and techniques seek to achieve optimum assignment, in some sense, of element content and overall system structure. In the present context we consider only one aspect of partitioning using the term

module for convenience. The discussion completes the presentation of the SPE classification and provides a link to other current methodological thinking [WIR79].

Consider the end result of the design process for an A-program to be constructed of primitive elements we term modules. The analysis and partitioning process will identify functional elements that can be fully specified and therefore developed as S-program modules. Any specification may of course be less than fully satisfactory. It may even prove to be wrong in relation to the remainder of the design. For example the specification may not mention input validity checks, the specified output accuracy may be insufficient or the specified range of an input variable may be wrong. But each of these represents an omission from or an error in the *specification*. Thus it is rectified by first correcting the *specification*, and then creating, by one means or another, a new program that satisfies the new specification.

At any stage of the decomposition process there will remain required system functions that are at least partly heuristic or behavioural in nature and therefore define A-elements. It is suggested that it is *always* possible to continue the system partitioning process until *all* modules are implementable as S-programs. That is, any imprecision or uncertainty emanating from model reflections of incomplete world views will be implicit or, if recognised when the specification is formulated, explicit in the specification statement. The final modules will all be derived from and associated with precise specifications, which *for the moment*, may be treated as complete and correct.

The design may now be viewed and constructed as a data flow structure with the inputs of one module being the outputs of others (unless emanating from outside the system). Each module will be defined as an abstract data type [LIS77], [JON80], [SHA80] defining in turn, one or more input-to-output transformations. Module specifications include those of the individual interfaces, but for the system as a whole, the latter should in some sense be standardised [LEH77c]. Given appropriate system and interface architecture and module design, each module could be implemented as a program running on its own micro-processor and the system as a distributed system. The potential advantages for both execution (parallelism) and maintainability (localisation of change) [LEH77c], [BEL78], [WIR79] cannot be discussed here.

For the system as a whole, of course, some uncertainty or incompleteness remains as the root cause of further evolution (19.1). The methodology will have constrained that uncertainty to lie *entirely* and *visibly* in the module specifications. When the program proves unsatisfactory the analysis may in the first instance concentrate exclusively on them. When any must be changed to correct a fault or to enhance the system, modules that implement affected specifications will have to be replaced. Once the specification-based change-analysis has been completed, implementation of each detail of the change will have been restricted to lie within individual modules.

Many problems in connection with the design and construction of such systems need still to be solved. Adequate solutions will represent a major advance in the development of a process methodology (Section 5.3). We observe, however, that the concepts presented follow directly from our brief analysis and classification of program types. Interestingly the conclusions are completely compatible with those of the programming methodologists [DAH72], [LIN77], [WIR79].

3 Laws of Program Evolution

3.1 Evolution

The meta-system within which a program evolves contains many more feedback relationships than those identified above. Primitive instincts of survival and growth result in the evolution of stabilising mechanisms, implemented as checks, balances and controls, in response to needs, events and changing objectives. The resulting pseudo-hierarchical structure of self-stabilising systems includes the products, the processes, the environments and the organisations involved. The interactions between and within the various constituents, and the overall pattern of behaviour must be understood if a program product and its usage are to be effectively planned and maintained.

The organisational and environmental feedback, links, focusses, and transmits the evolutionary pressure to yield the continuing change process. A similar situation holds, of course, for any human organised activity, any artificial system. But some significant differences are operative in the case of software. In the first instance there is no room

19.1 (Eds) *Due to the inherent permissiveness of the application* [MAI84].

in programming for imprecision, no malleability to accommodate uncertainty or error. Programming is a mathematical discipline. In relation to a *specific* objective, a program is either right or wrong. Once an instruction sequence has been fixed and unless and until it is manually changed, its behaviour in execution on a given machine is determined solely by its inputs.

Secondly a software system is soft. Changes can be implemented using a pencil and/or keyboard. Moreover once a change has been designed and implemented on a development system its installation in the field does not require welding guns, sledge hammers or heavy machinery. In theory it can be applied mechanically to any number of instances of the same system without further significant physical or intellectual effort using only computing resources. Thus the temptation - economic and in the interest of speed - is to implement changes in the existing system, change upon change upon change, rather than to collect changes into groups and implement them in a totally new instance. As the number of superimposed changes increases, the system and the meta-system become more complex, stiffer, more resistant to change. The cost, the time required and the probability of an erroneous or unsatisfactory change all increase.

Thirdly, the rate at which a program executes, the frequency of usage, usage interaction with the operating environment, economic and social dependence of external process on program execution, all cause deficiencies to be exposed. The resultant pressure for correction and improvement leads to a system rate of change with a time scale measured in days and months rather than in the years and decades that separate hardware generations. To study and model system evolution, whether biological or artificial, requires the passage of many generations. For artificial systems implemented with physical components, hardware, the life time of a generation is of the order of magnitude of the professional life of the observer. He has no opportunity to measure, model or experiment with the evolutionary process. In fact, he does not normally perceive or experience it as a process. The rate of change in software, however, causes the individual to be exposed to many versions of a system. He experiences it as a sequence whilst also coping with individual instances. Program evolution can be studied as a dynamic process. Programs play the role of *fruit flies* in the study of artificial system evolution.

With fruit fly, however, the evolutionary process is, at least at the level of human perception, purely mechanistic. The changes, perturbed at most by very rare and transient events (mutations) show a statistical regularity that permits quantitative modelling and statistically significant prediction. Software evolution on the other hand would appear to be driven and controlled by human decision, managerial edict and programmer judgement. The frequency of random transient events, of mutation, is high.

Measures of software evolution and the associated process could therefore be expected to be erratic, reflecting at each instant and for each release, the pressures of the moment. The Program Evolution Dynamics studies, however, have shown that this is not so [LEH69], [BEL71b], [LEH74a], [BEL76], [RIO77], [LEH77e], [WOO79a], [CH080]. Instead such measures display patterns, regularity and trends that suggest an underlying dynamics that may be modelled and used for planning, for process control and for process improvement.

3.2 Program Evolution and Dynamics

Once observed the reasons for this unexpected regularity is easily understood. Individual decisions in the life cycle of a software system generally *appear* localised in the system and in time. The considerations on which they are based *appear* independent. Managerial decisions are largely taken in relative isolation, concerned to achieve local control and optimisation, concentrated on some aspect of the process, some phase of system evolution. But their aggregation, moderated by the many feedback relationships, produces overall systems response which is regular and often normally distributed.

In its early stages of development a system is more or less under the control of those involved in its analysis, design and implementation. As it ages those working on or with the system become increasingly constrained by earlier decisions, by existing code, by established practices and habits of users and implementors alike. Local control remains with people. But process and system-internal links, dependencies and interactions cause the global characteristics of system evolution to be determined by organisation, process and system parameters. At the global level the meta-system dynamics have largely taken over.

3.3 Measures of Program Evolution

The discussion, so far, has been entirely phenomenological in nature. The conclusions have, however, not been reached just by philosophising about the nature of software and the programming process. They have followed from the measurement of a number of programs and programming projects. The resultant observations have led to modelling, interpretation, prediction and the development and refinement of theories that describe the programming process and program evolution. This in turn has led to a deeper understanding of the nature and implication of computing itself.

Since the original observation [LEH69], studies of program evolution have continued based on measurements obtained from a variety of systems. Program and process characteristics such as size, growth-rate, work rate, fault content and fault rates were obtained for successive releases of various systems. The data was then analysed and, surprisingly, produced statistically regular patterns and trends. Typical examples of the resultant models have been reported in the literature [BEL71b], [LEH74a], [BEL76], [LEH77e], [CH080].

Examples of the data, though not of the statistical analysis, will be presented as part of the case study given in Section 4.4.

It was repeated observation of such phenomenologically similar behaviour and the common interpretation of independent phenomena, that led to the conclusions presented here; to a set of five laws, that have themselves evolved as insight and understanding has increased. The laws as currently formulated are given in Table 1 and briefly discussed in the next sections; their practical implications in the section following. It is perhaps important to stress that the laws are abstractions of observed behaviour based on statistical models. They have no meaning until a system, a project and the organisational meta-system are well established.

Table 1: Laws of Program Evolution

I Continuing Change

A program that is used and that as an implementation of its specification reflects some other reality, undergoes continual change or becomes progressively less useful. The change or decay process continues until it is judged more cost-effective to replace the system with a re-created version.

II Increasing Complexity

As an evolving program is continually changed its complexity, reflecting deteriorating structure, increases unless work is done to maintain or reduce it.

III The Fundamental Law of Program Evolution

Program evolution is subject to a dynamics which makes the programming process, and hence measures of global project and system attributes, self-regulating with statistically determinable trends and invariances.

*IV Conservation of Organisational Stability
(Invariant Work Rate)*

During the active life of a program the global activity rate in the associated programming project is statistically invariant.

*V Conservation of Familiarity
(Perceived Complexity)*

During the active life of a program the release content (changes, additions, deletions) of the successive releases of the evolving program is statistically invariant.

3.4 Laws of Program Evolution Dynamics

The first law, *Continuing Change*, originally [LEH74], [LEH78], [LEH79] expressed the universally observed fact that large programs are never completed. They just continue to evolve. Following our new insight, however, reference to *largeness* is now replaced by the phrase ... 'that reflect some other reality ...'. In theory this modification changes the scope of the law. In practice it widens it since with current methods large programs are, in general, economically feasible only if they relate to and have value in the outside world. Large programs can, in general, be expected to be *evolving* programs.

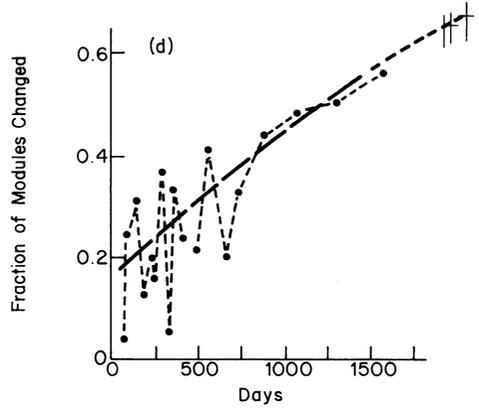
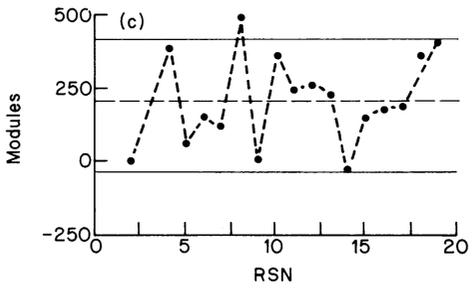
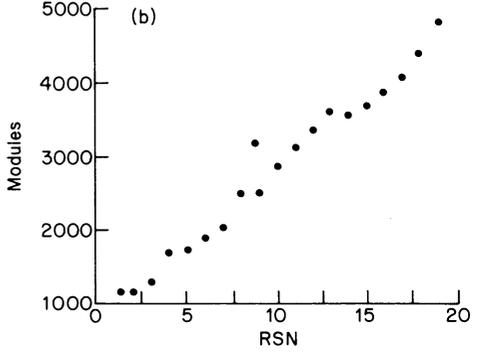
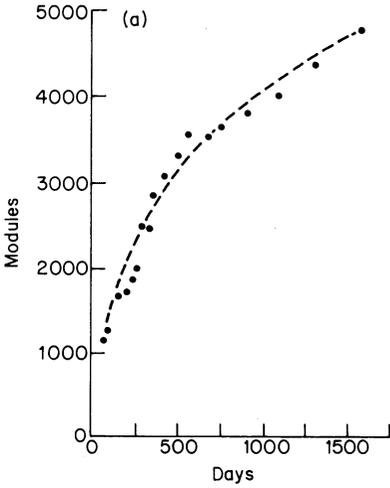
The law must not be interpreted as defining evolving programs. It points to the link with reality as a sufficient condition for evolution and therefore includes both P and E programs. Evolution of the former can, however, be more easily controlled since it is not a part of the external reality; merely compared with, and therefore influenced by, it.

The second law, *Increasing Complexity*, could be seen as an instance of the second law of thermodynamics. It would seem more reasonable to regard both as instances of some more fundamental natural truth. But from either viewpoint its message is clear.

The third law, the *Fundamental Law of Program Evolution*, is in the nature of an existence rule. It abstracts the observed fact that the number of decisions driving the process of evolution, the many feedback paths, the checks and balances of organisations, human interactions in the process, reactions to usage, the rigidity of program code, all combine to yield statistically regular behaviour such as that observed and measured in the systems studied.

The fourth law, *Conservation of Organisational Stability*, and the fifth *Conservation of Familiarity*, represent instances of the observations whose generalisation led to the third law. They express truths about the evolution process, that in turn are interpreted as reflecting more basic truths about human organisations and about individuals in those organisations, respectively.

The fourth law and the phenomenon from which it derives (exemplified by Figure 5a), is interpreted as a reflection of organisational aspirations for stability. The management of



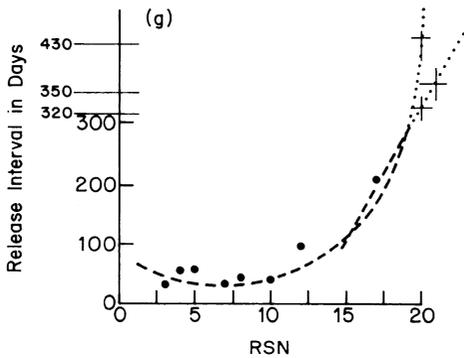
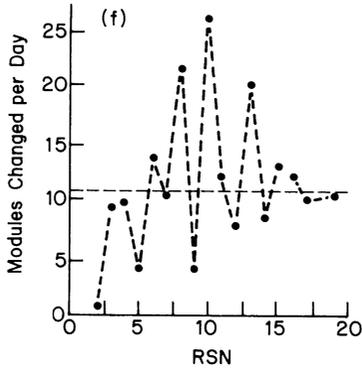
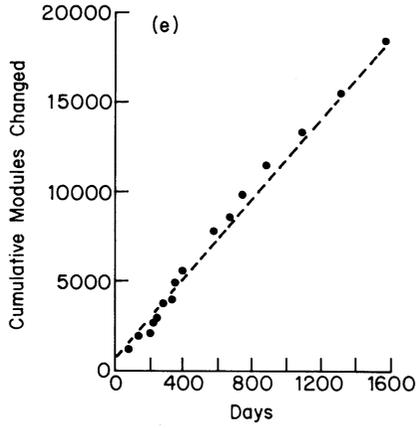
Figures 5a - 5d: System Characteristics

well established organisations avoid dramatic change and particularly discontinuities in growth rates. Moreover, the number of people, the investment involved, the unions, the time delays in implementing decisions all operate together to prevent sudden or drastic change. Wide fluctuations may lead to instability and the break-up of an organisation.

Why the particular measure (changed modules per day) plotted in Figure 5e should prove to be statistically invariant is not well understood. There can be little doubt that it represents the steadiness of an external variable in a multi-loop self-stabilising system.

The reader may find it difficult to accept the implication that the work output of a project is independent of the amount of resources employed, though the same observation has also been recorded by others [BRO75]. The underlying truth is that activities of the type considered, though initiated with minimal resources, rapidly attract more and more as commitment to the project, and therefore the consequences of success or failure, increase. Observations as formalised in the fourth law imply that the resources that can be productively applied becomes limited as a software project ages. The magnitude of the limit depends on many factors including attributes of the total environment. But the pressure for success leads to investment to the point where it is exceeded. The project reaches the stage of resource saturation - further changes have no visible effect on overall output.

The fourth law springs from a pattern of organisational behaviour. The fifth, on the other hand, reflects the collective consequences of the characteristics of the many individuals within the organisation. It is discussed at length in [LEH80a]. Suffice it to say here that the law arises from the non-linear relationship between the magnitude of a system change and the intellectual effort and time required to absorb that change. A system cannot be used or further maintained effectively until there is full awareness of all system interactions and responses, and each of the current changes is fully understood in the context of all the others. Until full familiarity is restored any involvement with the system will produce a feeling of strangeness and discomfort, further changes will inevitably lead to problems. Moreover, our observations indicate that the consequences of the (at least) quadratic relationship between the content of a release and the effort required to master it leads in the feedback environment of the programming and usage processes to the statistical invariance of the release content (Fig 5c).



Figures 5e - 5g: System Characteristics

In summary, the complex hierarchical system in and through which a software system is created, used and maintained itself evolves to be self-stabilising. If it did not, it would disintegrate and disappear - as so many software organisations and software systems have done. Reviewing the rapid development of the software industry and the software process over the last thirty years, it should be no surprise to discover that the survivors are highly stable.

5.3 The Nature of the Laws

The laws, as discussed above, were derived from observations on systems as they evolved over a series of releases. They therefore represent an abstraction and approximation, with measures at the point of release taken as representative of the entire release. A program may, equally, be viewed as having evolved over a time period terminated by its release, from initial identification of content or properties to the final usable program. Equally one may consider (Section 5.2) the evolution of sequences of release sequences. These become significant when one accepts the need for periodic *total system replacement*.

Measures at the two latter levels would provide quite different data. We cannot consider here the form of laws - if any - that might emerge from such studies. Clearly, however, the laws based on release-related phenomena, also reflect phenomena more meaningful at the other levels. They must be viewed as an approximation to more detailed models that might be developed.

There is also a further more direct approximation in the present formulation. The laws have been presented without supporting data or statistical assessment [CHO80]. We have, however, clearly indicated (eg Section 3.2) that the dynamics and its parameters evolve as a system with its supporting organisation and process, develops. Equally, when a series of releases approaches the point where a total replacement - the next generation - begins to emerge, there must be transfer effects that impact the release oriented dynamics and its parameters. Consider, for example, the fourth law. As formulated it implies a linear fit to the data of figure 5f. This brief analysis indicates, however, that strictly speaking, the curve must be S-shaped, modelled by at least a cubic. Thus, for greater precision the laws should be qualified to indicate that they describe phenomena that, from the point of view of successive releases, dominates much of the life cycle of a release generation.

Whatever the precision of the laws, as stated, our main conclusion is clear. As it ages, a software project - at whatever level observed - is increasingly constrained by its internal characteristics and those of the meta-system. The freedom of choice and manoeuvrability of the individual manager is progressively restricted.

The arguments presented here may not convince the reader that he is not in total control of his program development; that to some degree the process controls him. This despite the fact that the same phenomenon can also be observed in, say, economic or social systems. One additional fact must, however, be considered. We are examining man-made systems, processes designed, directed and largely controlled by human decision and activity. The present laws are derived from data emanating from environments managed and conducted in ignorance of the laws. *Recognition of laws itself changes the environment.* It should lead, for example, to methodological and technological advances that invalidate the laws as presently formulated. That is the laws, as presently formulated, may become irrelevant to the way programs are created and maintained [LEH76], [LEH80a]. From our viewpoint, the trend towards distributed microprocessor systems as described briefly in Section 2.6 is an example of this, stemming as it does at least partly, from our analysis of programs and programming and the properties of 'large' programs [BEL78].

It remains to be seen whether with new methodologies and system structures, regularities such as those observed will once again appear. For the present, however, in their representation of the programming process, the set of laws reflects the very basic attributes and attitudes of people and organisations. As such, they are likely to remain significant for some time, certainly as long as large software systems structured, as those currently widespread, are created and maintained. For the present generation of planners, managers, programmers and users, the laws carry a message which in the long run proves costly to ignore. The practical implications are discussed in the next section.

4 Applied Dynamics

4.1 Introduction

The previous sections have emphasised the phenomenological basis for the laws of program evolution, indicating how they are rooted in phenomena underlying the activity of programming itself.

The origin of the laws in individual and societal behaviour makes their impact on the construction and maintenance of software more than just descriptions of the evolutionary process. The laws represent principles in Software Engineering. They are, however, clearly not immutable as are, for example, the laws of physics or chemistry. Since they arise from the habits and practices of people and organisations, their modification or change requires one to go outside the discipline of computer science into the realms of sociology, economics and management. The laws therefore form an environment within which the effectiveness of programming methodologies and management strategies and techniques can be evaluated, a backdrop against which better methods and techniques can be developed.

4.2 Methodological Implications

The development of programming methodology over the last twenty years has been largely experimental in nature, guided by experience of the actual process. Where thought has been given to the nature of the act of programming it tended to concentrate on specific aspects, specification, programming per se or testing for example, rather than the *process* of converting an applications concept into the executable binary code sequence that controls its implementation. Nor, until recently, has much consideration been given to the maintenance process that seeks to ensure that the implementation remains effective and cost effective as operational experience is gained and as the operational environment changes.

What are the methodological implications of the laws? The inevitability of continuing change asserted by the first law implies that, like correctness, alterability is an essential attribute of program quality. To be alterable a program must be comprehensible. This demands accessible, complete and consistent specification, design records and code. All must be well structured and express clear concepts in intelligible and unambiguous language. The meaning and domain of all terms must be explicitly declared and preferably suggested by the allocation of appropriate descriptors. Since understanding of a program depends not only on its static description but also on the order of execution, each sequencing control mechanism must provide the human reader with a clear indication of its function. In short, all unnecessary complexity must be avoided. Moreover partitioning of the programs into small elements and controlling the interfaces between them is also essential [STE74],

[MYE78], [YOU68]. As far as possible the system must be designed so that changes relating to a specific function can be localised, with changes to code and/or documentation restricted to, at most, a small group of related elements. Insofar as a change must be more widespread its flow and impact should be discernable. There must be no invisible links between elements, for example through access to a global name. There should be no side effects, no incidental consequences of the execution of any code sequence. Assumptions should not have to be made about individual elements or relationships between pairs. In general, it must be possible to localise change within the system. When more widespread impact is indicated, a procedure must be definable so that *all* necessary changes are readily identified.

All of the above needs are recognised and addressed by current developments in programming methodology; high level language, structured programming, system and module specification, information hiding, procedure protocol, scope rules, declaration, strong typing, parameter passing, functional sequence control (if ... then ... else ..., while ... do ... etc.) and so on. What has been achieved in this paper is to associate these techniques through the concept of alterability, a concept viewed as complementary to, and essential for, dynamic correctness.

The second law reinforces the conclusion that unnecessary complexity must be avoided from the start. A program can never be simpler than the abstraction of the application or problem that it models. It should not be allowed to exceed that complexity by more than is absolutely necessary. Even then complexity will increase with time, unless work is expended to control or reduce it. Therefore it is always profitable to start from a point of minimum complexity. As the system is changed, the designer and the programmer must, subject to other constraints, minimise complexity growth over the period of concern. It is therefore essential to be able to define and establish measures of complexity so as to be able to make quantitative assessments [MCC76], [LEH77g], [BEL77a], [CHE78], [MUS80], [BEL77c].

The third and fourth laws are most usefully discussed in the management context and are therefore considered in greater detail in the next section. However, in the case of the fifth law there are also aspects that are specifically methodological. System release permits all those associated with a program that has been modified or extended to adapt to its new content and to its new behaviour in execution.

Through the intellectual effort involved in the process of adaptation, developers and users alike achieve once again that degree of unconscious appreciation of all aspects of the program's attributes that enables them to develop it further or to use it successfully as a tool in their real pursuit. Thus each system release should provide a base from which further system evolution can reliably occur. Releases play a stabilisation role and must be planned and managed accordingly.

There is a further interesting interpretation of the fifth law that requires the anthropomorphic extension of the concept of familiarisation. The new program must 'become familiar' with its operational environment. In the relative isolation of the development and maintenance environment, it will have been validated by a series of tests. But these cannot be all-embracing or cover all operational eventualities. It is impossible to foresee or create even all likely environmental conditions in the test laboratory. Thus, inevitably when a program is released to the user environment, new situations will be encountered and new faults discovered. Only adequate exposure to actual usage by a range of users can determine the extent to which a program version represents a meta-stable plateau in the evolutionary growth process. If this is to be achieved smoothly and without major upheaval, the change selection and test process must be designed to expose the entire system to execution under conditions of actual usage and to stress it in every way. The release process must be designed to minimise the chance that a defect can survive undetected beyond the release in whose implementation it was embedded. The set of changes to be implemented over an extended time period must be clustered and allocated to individual releases so as to achieve optimum early actual usage pattern and exposure. This point is illustrated by aspects of the case study of section 4.4.

4.3 Management Implications

The modern industrial manager generally regards himself as totally in control of the activity under him, subject only to management edicts emanating from his superiors and physical limitations arising from the laws of nature. The latter fall into two classes. There are basic invariants expressed by conservation laws, for example, and there are the properties of the materials, the machines and the people through which he achieves project objectives.

Not unnaturally such physical limitations have not been recognised as being operative in software development or maintenance activity. Physical and material limitations do not, in general, impress themselves directly or explicitly on a manager in his management of software project planning and execution, except insofar as budget, people and computing resources are concerned. But the laws assert that constraints do exist. They arise partly from purely physical limitations but mainly from the feedback properties of the software processes and organisations. Their consequences limit the software manager just as purely physical constraints limit his colleagues.

It is difficult enough to convey the existence of limitations and constraints in the software production process. To obtain general acceptance of the implications has so far not proved possible. Yet the existence of constraints on software projects and on the power of software managers is the first and most fundamental management implication of the five laws.

The concept of alterability arising from the first law implies the need to dedicate a part of project resources to the creation and maintenance of system and code structure to ensure continuing comprehensibility and localisation of change [STE74], [MYE78], [BEL78]. It also requires the maintenance of a complete and accessible history of the system, its design and implementations and of the concepts, assumptions and decisions that underlie them. The record must also include the same detailed information about all subsequent changes so that any series of events in system maintenance, and the reasons and reasoning behind them, can be reconstructed.

The amount of information to be maintained is no more than exists, *de facto*, in the set of blueprints that are maintained in any well managed organisation to control the manufacture and marketing of almost any product that is itself an assembly of more than a few separate parts. Only if such information is readily available can a system be maintained correctly, efficiently and responsively. If this is so in an environment in which physical constraints limit the maintainer in his work on the system and in which physical properties and interconnections are visible, how much more must it apply where the structure, dependencies and constraints are intangible; where the system is inflexible, with zero tolerance relative to the unanticipated event?

Blueprint technology in association with micro-filming has just about proved able to cope with the needs of modern industry, though projects which involve many millions of prints stretch the concept somewhat. Such a paper-based, technology is completely inadequate for coping with any but the smallest software projects. The volume of records needed is simply too great. Furthermore the technology for generating written records is, to all intents and purposes, precisely the same as that required to record the code but often requires equal or even greater effort if completeness and clarity is to be achieved. Yet the code is seen as the *real*, immediately useful and therefore valuable, product. From the point of view of the developer who must generate both code and long-term documentation, the latter, containing information well understood at the time of design and coding, has only nuisance value. No wonder it is so often neglected.

The short-term psychological pressure for neglecting documentation is significant. Only the manager can have the long-term view that accepts complete documentation as a necessity in maximising the lifetime benefit from a system. It becomes a major element in the *manager's responsibility* to ensure that *motivation* is provided and that *facilities* are available in every project to maintain the documentation at the same state of readiness as the code, at all times. Once documentation has fallen behind, it can never catch up if only because the facts needing to be recorded will have been forgotten.

Thus it is insufficient to provide a programmer with an adequate programming language [BUX80], even if that language is so readable that programs written in it can be considered self-documented. The first law implies that if the problem to be solved is of any size, is of continuing interest or is connected to some external reality to which it must relate satisfactorily over an extended life period, a programming support environment must be provided. The environment can be seen as a tool kit composed as a set of programs implementing many different support capabilities. Its nucleus must be a recording system that automatically maintains all the information that may eventually be required if further changes in the future are to be made correctly and expeditiously. The speed at which the volume of information is generated during the programming process necessitates an automatic, computer based, data collection facility to guarantee the completeness, correctness and accessibility of the information. That is why a language design project like Ada is so closely coupled to the Stoneman programming support environment

project [BUX80]. Top management must accept the need to provide resources for the development or acquisition and maintenance of methods and tools as an essential part of cost-effective computer usage, even if the related investment shows no immediate and no measurable return. In fact the cost figures quoted in Section 1 suggest that initial expenditure for non-progressive [BAU67] investment in technique development and support tools may pay off handsomely over the product life cycle.

This need to invest in anti-regressive [LEH74] activities follows also directly from the second law. Complexity control and complexity reduction yields no immediate benefit. Thus the manager whose performance is judged by his profit yielding output has no incentive to concern himself with any aspect of complexity. It is a *management responsibility* to provide such incentive. After all, over the life time of any evolving system the direct and indirect cost of insidious complexity growth can exceed the alternative cost of controlling it, many times.

In practice, of course, the decision as to the fraction of a development or maintenance budget to be dedicated to complexity control and reduction cannot be based on cost comparison. If adequate control is exercised, the subsequent cost does not arise and the saving cannot be assessed. If it is not, it will not, in general, be possible to determine how much of the subsequent effort and cost of maintenance could have been avoided, had minimal complexity been maintained. This is the inherent paradox of all anti-regressive activity. Its value lies in that it forestalls subsequent problems and costs. So its return can never be evaluated.

The second law implies a management responsibility to dedicate resources to the management of complexity and to the development of methods to achieve this. Furthermore definitions of complexity must be created to permit quantitative assessment and the development of appropriate models that predict complexity growth. Thus activity and success in any of these areas must attract the same management encouragement, support and recognition as does the development of a marketable product.

The implications of the third law are clear. Management must learn to accept the same sort of limitations for software projects as it does in other areas. For example, the rate at which a system can be changed, depends on many factors other than just the resources applied [BRO75]. It is therefore

necessary to measure characteristic parameters and to construct process models that provide planning parameters. The example of the next section will demonstrate what can be achieved.

The fourth law provides a specific instance of the third. Though observed on *all* projects monitored by us, the invariance can at the moment only be interpreted in broad phenomenological terms. That is the maintainable rate can only be determined by extrapolation from measurements of an activity over a period. It cannot yet be established or even estimated from other project or organisational parameters.

Nonetheless, it is very real and implies that one should not plan for long periods of activity at work rates exceeding the average achieved over the life time of the system to that point. If the nature of external requirements and the changes to be made are such that the average work rate *must* be exceeded for a release or two, plans should be so formulated that the following releases are achieved at a lower work rate. The decline will occur in any case and it is always better that it should happen as a consequence of formulated plans, rather than forced by events.

Questions of productivity are not specifically addressed in this paper. We may, however, just note that invariances such as those implied by the fourth and fifth laws, provide measures that over a sufficiently long period can indicate whether productivity improvements have really been achieved.

The implications of the fourth law may be summarised as follows:-

- * Do not plan long periods for work rates exceeding the average
- * If the work rate must exceed the average over one or two releases, plan to reduce it for the next one or two.
- * Use measures of the work rate to test for genuine (maintainable) productivity growths.

Analogous comments apply also to the implications of the fifth law, except that the latter relates to release content rather than work rate. That is the following management principles may be established:-

- * Plan content of successive releases so that average system growth fluctuation is minimised.

- * If growth has to exceed the average substantially for one or two releases, plan for a clean-up or minimal growth release.
- * Fix release dates (intervals) to yield constant average growth.
- * Use measured average to check for genuinely improved methods, productivity and release control.

4.4 A Case Study - System X

4.4.1 The System and its Characteristics

System X is a general purpose batch operating system running on a range of machines. The eighteenth release (R18) of the system is operational in some tens of installations running a variety of work loads. The nineteenth release (R19) is about to be shipped.

Table 2 and figures 5a to 5g present the system and release data available for the purposes of the present exercise. Details of statistical analysis and model validation, based on this data and that from other systems, that provides confidence in the conclusions and predictions cannot however be provided here.

Examining the system dynamics as implied by models derived from the data and as illustrated by the figures, Figure 5a shows the continuing growth of the system (first law) albeit at a declining rate (demonstrably due to increasing difficulty of change, growing complexity - second law).

Figure 5b indicates that, as a function of release sequence number (RAN) the system growth (measured in modules) has been linear but with a superimposed ripple (a strong indicator of feedback stabilisation).

Figure 5c shows the net incremental growth per release (fifth law).

For system architectures such as that of system X, the fraction of system modules that are changed during a release may be taken as a gross indicator of system complexity. Figure 5d shows that system X complexity, as measured in this way, shows an increasing trend (second law).

Figure 5e is an example of the repeatedly observed constant average work rate (fourth law).

Table 2: System X Statistics

Release 19 Statistics

Size	4800 Modules 1.3M Assembly Statements
Incremental Growth	410 Modules
Modules Changed*	2650 Modules
Fraction of Modules Changed	.55
Release Interval	275 Days

System Statistics

Age	4.3 Years
Change Rate	10.7 Modules/Day
Average Incremental Growth	200 Modules/Release
Maximum Safe Growth Rate	400 Modules/Release

Most Recent Releases

Release	15	16	17	18	19
Incremental Growth (Mod)	135	171	183	354	410
Fraction Changed	.33	.43	.48	.50	.56
Change Rate	12.5	.12	9.6	9.9	9.6
Interval (Days)	96	137	201	221	275
Old Mods. Changed/Mod	7.9	8.6	10	5.1	5.4

* Modules that are changed in any way in release $i + 1$ relative to release i are counted as one changed module, independently of the number of changes or their magnitude.

Figure 5f illustrates how the average work rate achieved in individual releases, as measured by the rate of module change (changed modules per release interval day, m/d) oscillates, a period of high rate activity being followed by one or more in which the activity rate is much lower (third law).

Finally, figure 5g plots the release interval against release sequence number. It might be argued that release interval depends purely on management decision that is itself based on market considerations and technical aspects of the release content and environment. Data such as that of Figure 5g indicates, however, that the feedback mechanisms that amongst other process attributes also control the release interval, while including human decision taking processes are apparently not dominated by them. As a consequence, the release interval pattern is sufficiently regular to be modellable, and is statistically predictable once enough data points have been established.

4.4.2 The Problem

Already prior to the completion (and release) of R19, work has begun on a further version, R20, whose main component is to be the addition of interactive access to complement current batch facilities. This new facility 'ITS' together with other changes and additions summarised in Table 3, are to be made available in R20 to be shipped eighteen months after first customer installation of R19.

For each major planned functional change the table lists the number of new modules to be added (NM), the number of R19 modules that are to be changed in any way in the course of creating R20 (OMC), the total number of modules changed (NM + OMC), and the ratio of OMC to NM (the interconnectivity ratio (IR), an indicator of complexity). No modules are planned for removal in the creation of R20 hence the planned net system growth is 1021 modules.

Management has also accepted that a further release R21 will follow twelve months after R20, to include any leftovers from R20. It may also include additional changes for which a demand develops over the next two years. The current exercise is to endorse the overall plan, or if it can be shown to be defective, to prepare an alternative recommendation.

Table 3: Release 20 - Planned Content

Functional Enhancement					
No	Description	New Mods (NM)	Old Mods Chgd (OMC)	Mods Chgd (NM+OMC)	OMC/NM (IR)
1	Identified Faults (Pre Rls 19)	2	380	382	-
2	Expected Faults (Rls 19)	0	600	600	-
3	Interactive Terminal Support (ITS)	750	1783	2533	2.4
4	Dynamic Storage Management (DSM)	170	1500	1670	8.8
5	Remote Job Entry (RJE)	57	462	519	8.1
6	New Disc Support (NDS)	17	124	141	7.3
7	Batch Scheduler Improvements (BSI)	3	29	32	9.7
8	File Access System (FAS)	8	74	82	9.3
9	Paper Tape Support (PTS)	12	80	92	6.7
10	Performance Improvements	2	157	159	-
		1021	5189	6210	

'ITS' Detail

No	Description	New Mods	Old Mods Chgd	OMC/NM
3a	Terminal Support	444	1032	2.3
3b	Scheduling	127	293	2.3
3c	Telecom support	58	232	4.0
3d	Misc	121	226	1.9
		750	1783	

4.4.3 Process Dynamics

4.4.3.1 Work Rate

From Figure 5e the work rate has averaged to 10.4 m/d (19.2) over the life time of the system. Figure 5f indicates that the maximum rate achieved so far has been 27 m/d. Evidence that cannot be detailed here reveals, however, that that data point is misleading and that a peak rate of about 20 m/d is a better indicator of the maximum achievable with current methods and tools. Moreover, there is strong circumstantial evidence that releases achieved with such high work rates were extremely troublesome and had to be followed by considerable clean-up in a follow-up release, as also implied by figure 5c. Thus if R20 is planned so as to require a work rate in the region of 20 m/d, it would be wise to limit R20 to at most 10 m/d, the system average. If on the other hand the process is further stabilised by working on R20 at near average rate, one could then, with a high degree of confidence, approach R21 with a higher work rate plan.

4.4.3.2 Incremental Growth

The maintained average incremental growth for system X has been around 200 modules per release. Once again circumstantial evidence indicates that releases for which, in this case the growth rate (incremental growth per release) has exceeded double the average, have slipped delivery dates, a poor quality record and a subsequent need for drastic corrective activity. Figure 5c and Table 2 indicate that R19 will lie in this region and that R18 had high incremental growth. That is, R19, once released, is likely to prove a poor quality base. The first evidence emerges that maybe R20 should be a clean up release.

4.4.3.3 Growth Rate Cyclicity

The same indication follows from figures 5a and 5b where the ripple periods are seen to be three, four and five intervals respectively over the first three cycles. In the fourth cycle, six intervals of increasing growth rate have passed with the R18 - R19 growth the largest ever. Without even considering the planned growth to R20 (point X), it seems apparent that a clean-up release is due.

19.1 (Orig)

$$\text{modules per day} = \frac{\text{number of modules changed in release}}{\text{release interval in days}}$$

4.4.4 R20 Plan Analysis

4.4.4.1 Initial Analysis

The first observation on the plan as summarised by Table 3 stems from the column (6) of IR factors. It has not been calculated for items 1, 2 and 10 since these represent activities that only rarely require the provision of entirely new (non-replacement modules). For items 4 through 9 the ratio lies in the range 8.2 ± 1.5 , a remarkably small range for widely varying functional changes. Yet the predicted ratio for ITS is only 2.4. One must ask whether it is reasonable to suppose that the code implementing an interactive facility is far more loosely coupled to the remaining system than, for example, a specialist facility such as paper tape support? Is it not far more likely that ITS has been inadequately designed; viewed perhaps as an independent facility that requires only loose coupling into the existing system? Thus when it is integrated with the remainder of the system to form R20, may it not require many more changes to obtain correct and adequate performance? From the evidence before us the question is undecidable. Experience based intuition, however, suggests that it is rather likely that the number of changes required has been under-estimated. Thus a high-priority design re-appraisal is appropriate. If the suspicion of incomplete planning proves to be correct it would suggest delaying R20, so that the planning and design processes may be completed. An alternative strategy of delaying at least ITS to R21 should also be evaluated.

4.4.4.2 Number of Modules to be Changed

The situation may of course not be quite as bad as direct comparison of the present estimate of the ITS interconnection ratio IR with that of the other items, suggests. In view of the 750 new modules involved, its IR factor could not exceed 6.4 even if all 4800 modules of R19 were affected by the ITS addition. Such a 100% change is, in fact, very unlikely, but the IR factor of 2.4 remains suspect.

Moreover even with the low ratio for ITS the sum of the individual OMC estimates for the entire plan exceeds the number of modules in R19. This suggests that there is a new situation. Multiple changes applied to the same module have become a significant occurrence. Even ignoring the fact that even independent changes applied in the same release to the *same* module generally demand significantly more effort than

changes applied to independent modules, the total effort and time required must clearly increase with both the number of changes implemented and the number of modules changed. The presently defined measure 'modules changed' is inadequate. The new situation demands consideration of more sensitive measures such as 'number of module changes' and 'average number of changes per module'.

These cannot be derived from the available data. One may, however, proceed by considering a model based on the data of Figure 5d. Extrapolating the fraction changed trend, reveals that R20 may be expected to require a change of, say, 64% or 3725 changed modules (19.3). Comparing this estimate with the total of 6210 obtained if the estimates for individual items are summed it appears that the average number of changes to be applied to R19 modules according to the present plan is at least of order two. We have already observed that multiple changes cause additional complications. Hence any prognosis made under the implied assumption of single changes (or of a somewhat lower interconnection ratio) will lead to an optimistic assessment.

4.4.4.3 Rate of Work

The current plan calls for R20 with its 3725 module changes to be available in 18 months, that is 548 days. This implies a change rate of less than 6.8 m/d. This relatively low rate, following a period of average rate activity suggests that *work rate* pressures are unlikely to prove a source of trouble, even with multiple changes to many of the modules.

4.4.4.4 Growth Rate

In Figures 5a and 5b the position of R20 as per plan has been indicated with an x. Both models indicate that the planned growth represents a major deviation from the previous history. Thus confirmation that the plan is realistic requires a demonstration that the special nature of the release or changes in methods makes it reasonable to expect a

19.3 (*Orig*) *Historical Note: In the system on which this example is based the release including the interactive facility ultimately involved some 58% of modules changed. Moreover the first release was significantly delayed, and was of limited quality and performance. More than 70% of its modules had subsequently to be changed again to attain an acceptable product. Our estimate is clearly good.*

significant change in the system dynamics. In the absence of such a demonstration, the suspicion that all is not well is strengthened.

4.4.4.5 Incremental Growth

The current R20 plan calls for system growth of over 1000 modules. This figure which is five times the average and two and a half times the recommended maximum, must be interpreted as a danger signal.

It has already been suggested that the low interconnection ratio for ITS suggests that the planners saw the new component as a stand alone mechanism that interfaces with the remainder of the system via a narrow and restricted interface. If this view proves justified, the large incremental growth need not be disturbing. But it seems reasonable to question it. With the architecture and structure that system X is known to have, such a relatively narrow interface is unlikely to be able to provide the communication and control bandwidth that safe, effective and high capacity operation must demand. This is apparent from comparisons with, say the paper tape or disc support changes or the RJE addition. The onus must be put onto the ITS designers to demonstrate the completeness of their analysis, design and implementation.

Without such a demonstration one must conclude that the present plan is not technically viable. *Marketing* or other considerations may, of course, make it desirable to stay with the present plan even if this implies slipped delivery dates, poor and unreliable performance of the new release, limited facilities and so on. But if such considerations force adoption of the plan, the implications must be noted and corrective action planned. Ways and means will have to be created to enable users to cope with the resultant system and usage problems and the inevitable need for a major clean-up release. It might, for example, be wise to set up specialised customer support teams to assist in the installation, local adaptation and tuning of the system.

4.4.4.6 Release Interval

Figure 5g indicates two possible models for the prediction of the most likely (desirable?) release interval for R20 and R21. Linear extrapolation suggests a release period of under one year for each of the two releases. If this is valid, the apparent desire for a release after the 18 months is of itself unlikely to prove a source of problems. On the basis

of evidence not reproduced here, however, the exponential extrapolation is likely to be more realistic and this yields an R20 release interval forecast of about 15 months and an R21 interval of some 3 years.

4.4.4.7 Recommendation - Summary

On the basis of the available data it has been concluded that:

- 1 To proceed with the plan as it stands is courting serious delivery and quality problems for R20.
- 2 A clean-up release appears due in any case.
- 3 Failure to provide it will leave a weak base for the next release. At the very least the number of expected faults (Table 3 - item 2) is likely to prove an underestimate.
- 4 The absolute size of the ITS component and the related incremental system growth represents a major challenge even on a clean base.
- 5 There are indications that the ITS aspect of the release design is incomplete.
- 6 Change rate needs for R20 are not likely to prove a source of problems.
- 7 Nor is the demand for attainment of a next release in eighteen months.

The following recommendations follow:-

- 8 Initiate immediately an intensive and detailed re-examination of the ITS design and its interaction with the remainder of system X.
- 9 From the integration records of R19 and by comparison with the records of earlier releases, make quality and error rate models and obtain a prognosis for R19 and an improved estimate for R20 correction activity. (Integration and error rate models have not been considered in the present paper but have been extensively studied by the present author and by others [MUS80].

- 10 Assess the business consequences of, on the one hand, a slippage of one or two years in the release of ITS and on the other, a poor quality, poor performance release with a slippage of, say some months (due to acceptable work rate but excessive growth).
- 11 In the absence of positive indication of a potential for major deviations from previous dynamic characteristics or the existence of a genuine business need that is more pressing than the losses that could arise from a poor quality product, abandon the present plan.
- 12 Instead redesign release 20 to yield R20; a clean, well-structured, base on which to build an ITS release, R21'.
- 13 Tentatively release intervals of 9 months and 15 months are proposed for R20' and R21' respectively.
- 14 R21' should be a restricted release for installation in selected sites.
- 15 It would be followed after one year by a general release R22'.

4.4.4.8 Recommendations - Details

Assuming that the further investigation as paragraphs 8 to 10 of 4.4.4.7 re-inforces the conclusions reached, three releases would have to be defined. Proposals for R20' & R21' are outlined here. The third, R22' will be a clean-up but its content cannot be identified in detail until a feel for the performance and general quality of R21' has developed. The detailed analysis is left as an exercise to the reader.

The inherent problem in the design of the ITS release is the fact that the component has a size almost twice the maximum recommended incremental growth. Moreover, with the possible exception of its telecommunications support (Table 3, item 3c), none of the component sub-systems would receive usage exposure in the absence of the others. Thus a clean ITS release cannot be achieved except by releasing the component in one fell swoop. Similarly, Dynamic Storage Management (DSM) is exposed to user testing only when the ITS facility is operational. One may, however, consider whether the telecommunication facility (3c) will be usable in conjunction with the RJE facility, item 5. If it is, there will be some advantage to be gained by releasing 3c and 5 before the remainder of ITS and DSM.

Strictly speaking, Figure 5c suggests that R20' should be a very low content release dedicated to system clean-up and restructuring. But the six preceding releases were achieved with average change rates and, from that point of view, did not stress the process. Thus if R20' is also an average rate release it should not cause problems and it would seem a low risk strategy to include in R20' all those items as in Table 4, that will simplify the subsequent creation of the excessively large ITS release.

The list, in priority order, of the new proposal shows a maximum incremental growth (159) well under average. It is a matter of some judgement and experience whether it would be wiser to delay item 3c with 58 new modules and item 5 with 57 to R21' thereby achieving the very low content release mentioned above. With the information before the reader it is not possible to resolve this question since additional information, at the very least answers to the questions raised in 4.4.4.7, would be required. However, the desire to minimise R21' problems suggests the adoption of the complete plan as in Tables 4 and 5.

Table 4: Modified Release 20 Content

Class	Reason	Items
Fault Repair	Clean-up of base	1, 2
Hardware Support	Revenue Producing	6, 9
Performance Improvement	Prove - but do not announce. Will be available to counteract ITS performance deterioration in R21'.	7, 10
ITS Related Components	To receive early user exposure	3c, 5, 8

Table 5: Modified Release 20 Statistics (from Table 3)

Item	New Mods	Running Total	Changes	Running Total
1	2	2	382	382
2	0	2	600	982
6	17	19	141	1123
9	12	31	92	1215
7	3	34	32	1247
10	2	36	159	1406
8	8	44	82	1488

5	57	101	519	2007

3c	58	159	522	2529

In assessing achievable release intervals for these releases, we base our estimates only on the module change count and change rate. The constraints on the present example do not permit the full analysis which would consider models based on Figure 5g, and take into account additional data. At 10 modules per day change rate, implementation of the complete plan appears to require 253 days, say 9 months, whereas exclusion of 3c and 5 would reduce the predicted time required to some 7 months. This recommendation cannot be taken further without more information of both a technical and a marketing nature, and an examination of other interval models. But the need for a clean base for R21' suggests adoption of the maximum acceptable release interval. R21' will now include, at the very least, ITS (except 3c) and DSM. This involves at least 920 new modules, an excessive growth that cannot usefully be further split between two or more releases. Assuming a change fraction of, say, 70% (figure 5d), of a system that is expected to contain 5911 modules, a total of some 4200 changed modules in the release, many with multiple changes must be expected. Since there will now have been seven near average change-rate releases, it seems possible to plan for a change rate of 15-20 m/d, yielding a potential release interval of under 9 months. That is, it would appear that, by adopting the new strategy, all of the

original changes and additions could be achieved in about the same time, but much more reliably. More complete analysis, however, based on additional data, other models and taking into account the special nature of the releases might well lead to a recommendation to increase the combined release interval to, say, two years.

A further qualification must also be added. As proposed in the revised plan, R21' will still be a release with excessive incremental growth and is therefore likely to yield significant problems. The additional fact that the evidence indicates incomplete planning, reinforces concern and expectation of trouble ahead. It is therefore also recommended that R21' be announced as an experimental release for exposure to usage by selected users in a variety of environments. It would be followed after an interval of perhaps one year by an R22', a cleaned up system, suitable for further evolution.

4.4.4.9 Final Comments

The preceding section has presented a critique of a plan, and outlined an alternative which is believed technically more sound. The case considered is based on a real situation, though in the absence of complete information details have had to be invented. But details are not important since the objective has been to demonstrate a method. Software planning, can and should be based on process and system measures and models, obtained and maintained as a continuing process activity. Plans must be related to dynamic process and system characteristics and to statistics of change. By rooting the planning process in facts, figures and models, alternatives can be quantitatively compared, decisions can be related to reality and risks can be evaluated. Software planning must no longer be based solely on apparent business needs and market considerations; on management's local perspective and intuition.

5 The Life Cycle

5.1 The General Case

The preceding sections have explored the causes, the dynamics and the direct implications of continuing program evolution. In recent years the phenomenon has given rise to a concept of a program *life cycle* and to techniques for *life cycle management*. The need for such management has, in fact, been recognised in far wider circles, particularly by

national defence agencies and other organisations concerned with the management of complex artificial systems. In pursuing their responsibilities these must ensure continuing effectiveness of systems whose elements may involve many different and fast developing technologies; often they must guarantee utterly reliable operation under harsh, hostile and unforgiving conditions. The outcome is an ever increasing financial commitment. Only life time oriented management techniques applied from project initiation can permit the attainment of life time effectiveness and cost effectiveness.

The problems in the more general situation are essentially those we have already explored, except that the time interval between generations is perhaps an order of magnitude greater than in the case of pure software systems. In briefly examining the nature of the life cycle and its management in this section the terminology of programming and software engineering are used. The reader will be able to generalise and to interpret the remarks in his own area of interest.

5.2 Software Life Cycles

Section 3.5 showed that program evolution, and the cyclic effects that define a program life cycle can be identified on different time scales representing various levels of abstraction. The highest level concerns *successive generations* of system sequences. Each generation is represented by a sequence of system releases. This level corresponds most closely to that found in the more general systems situation, with each generation having a life span of, from, say five to twenty years. Because of the relatively slow rate to change it is difficult for any individual to observe the evolution phenomenon, measure its dynamics and model it as a life cycle process, since in the relevant portion of his professional career he will not observe more than two or three generations. It might therefore be argued that this level should not be treated as an instance of the life cycle phenomenon. The present author has, however, had at least one opportunity to examine program evolution at this level and to make meaningful and significant observations [LEH76d]. These indicated that much could be gained in cost effectiveness in the software industry if more attention were paid to the earlier creation of replacement generations; something that can be achieved effectively only if the appropriate predictive models are available.

The second level is concerned as we have been, with a *sequence of releases*. The latter term is also appropriate

sequence of releases. The latter term is also appropriate when a concept of continuous release is followed; that is when each change is made, validated and immediately installed in user instances of the system. In the most extreme instances the user will find himself with a changed system almost every day.

Figure 6 shows one view [BOE76] of the lowest level, if it is assumed that 'maintenance' in the seventh box refers to on-site *fixes* and repairs implemented as the system is used. The diagram indicates the *sequence of activities* or life cycle phases that take a system instance from first identification of a need or opportunity, through the application of the completed program in its operational environment, to its final withdrawal. Such withdrawal, not explicitly indicated by Boehm, may of course be preceded by its replacement with an improved version, a new release. If maintenance is taken to refer to permanent changes, affected through new releases by the system originator, then the structure becomes iterative with each maintenance phase comprised of all seven indicated phases. With this re-interpretation the single model reflects the composite life cycle structure of all the above levels.

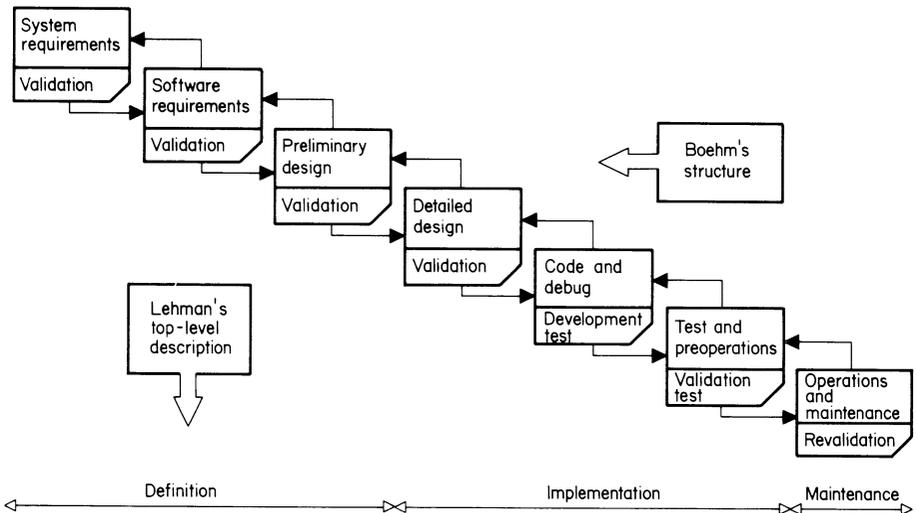


Figure 6: The Software Life Cycle According to Boehm

5.3 Assembly Line Processes

In strong contrast to the programming process considered above, an assembly line manufacturing process is possible when a system can be partitioned into sub-systems that are simply coupled and without invisible links. Moreover, the process must be divisible into separate phases without significant feedback control over phases and with relatively little opportunity for trade-off between phases.

If all these conditions exist the sub-processes of the total system development and construction process will not in general require knowledge of the total product or total process [BEL78]. Each part of the system and each phase of the process can be designed, implemented, evaluated and maintained more or less independently. Present day software technology is not like that. For example, at least some aspects of the specification and design processes are left over, usually implicitly, to the implementation (coding) phase. Fault detection through inspection [FAG76] is not yet universal practice and by default is often delayed till a system integration or system testing phase. When faulty behaviour is observed, the symptom is often removed by a local patch, leaving the real error to be located and repaired at some later time. Moreover, if at one stage inadequate attention is paid to structure, clarity, correctness, completeness, documentation or any other aspect of the process or of the attributes of its product, the consequences may not be felt till months or even years later when the code is to be modified or executed in an unexpected manner. And when some sub-system has to be changed, visible and invisible linkages and side-effects spread change requirements widely over the system.

In general, present day programming is constituted of tightly coupled activities that interact in many ways. One of the main concerns of life cycle process methodology research must be to develop techniques and tools and new system architectures (section 2.6). Programming support environments [DOL76], [HUT79], [BUX80] that permit partitioning of the program development and maintenance process into separated activities, coupled only by the program and its accompanying documentation are now recognised as essential for successful program development and maintenance.

5.4 The Significance of the Life Cycle Concept

For assembly line processes the life cycle concept is not, generally, of prime importance. For software and other highly complex systems it becomes critical if effectiveness, cost-effectiveness and long life are to be achieved. At each moment in time a manager's concern concentrates on the successful completion of his current assignment. His success will be assessed by immediately observable product attributes, quality, cost, timeliness and so on. It is his success in areas such as these that determine the furtherance of his career. Managerial strategy will inevitably be dominated by a desire to achieve maximum local pay off with visible short term benefit. It will not often take into account long term penalties, that cannot be precisely predicted and whose cost cannot be assessed. Top-level managerial pressure to apply life-cycle evaluation is therefore essential if a development and maintenance process is to be attained that continuously achieves the overall objectives of the organisation. Neglect will inevitably result in a life time expenditure on the system that exceeds many times the assessed development cost on the basis of which the system or project was initially authorised.

To overcome long time lags and the high cost of software, one may also seek to extend useful system lifetime. The decision to replace a system is taken when maintenance has become too expensive, reliability too low, change responsiveness too sluggish, performance unacceptable, functionality too limiting; in short, when it is more satisfactory to replace the system than to maintain it. But its expected life time to that point is determined primarily during conception, design and the initial implementation stages. Hence management planning and control during the formative period of system life, based on life time projections and assessment can be critical in achieving long life software and life time cost effectiveness. This represents an additional gain to the benefit gained from the continuing joint optimisation of expenditure, value received and cost per unit output over the lifetime of the system [GOL73].

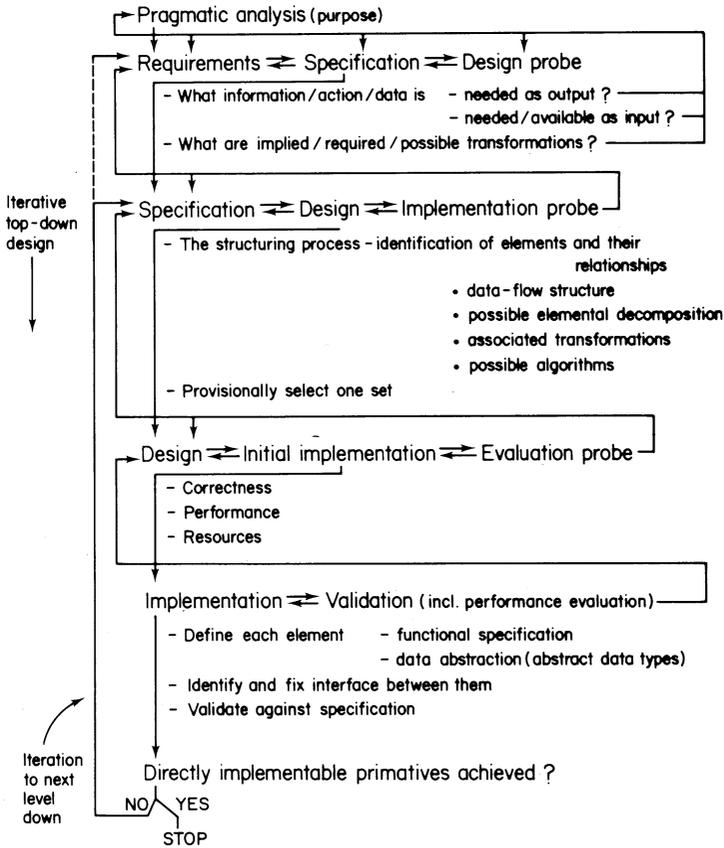


Figure 7: The Current-Idealised Model

5.6 Life Cycle Phases

5.6.1 The Major Activity Classes

At its grossest level a life cycle consists of three phases; definition, implementation and maintenance (19.4). As indicated in Figure 6, these three phases correspond approximately to the activities in the first three, the second three and the seventh box respectively of Boehm's model. In practice, however, many of these activities are overlapped, interwoven and repeated iteratively as suggested by the model of Figure 7. The details will vary with the environment, the methods employed and whether the project is concerned with the creation of a system to address an entirely new application, the preparation of a new release for an existing system, or the development of a replacement system for one that is reaching the end of its useful life. For simplicity the first of these cases is assumed in the brief discussion of the main activity areas.

5.6.2 System Definition

For E-Class systems in particular the development process begins with a pragmatic analysis leading into a systematic *Systems Analysis* to determine total system and program *Requirements* [BEL77c], [ALF78], [HEN79], [YEH80]. The analysis may examine manual techniques whereby the same purpose is currently achieved or, where appropriate, may be based on formal analysis. Whatever the approach, it has now been recognised that the analysis must be *disciplined* and *structured* [DAH72], [LIN77], the term structured analysis now being widely used [BEL78], [ROS77], [ROS77a].

By their very nature initial requirements, being an expression of the user's view of his needs, are likely to include incompatibilities or even contradictions. One would, for example, be most unlikely to be able to meet requirements that call for the creation within months of an utterly reliable, low-cost, system implementing a wide variety of related facilities that have not before been mechanised. Thus the analysis and the negotiation process by and between analysts and potential users, that produces requirements, must identify a balanced set that, in some sense, provides the optimum compromise between conflicting desires.

19.4 (Eds) *Today we would use the term evolution instead of maintenance.*

The requirements set will be expressed in the concepts and language of the application and its users. It must then be transformed into a technical specification. The *Specification* process [DEM78], [LIS79] must aim to produce a correct technical statement, *complete* in its coverage of the requirements and *consistent* in its definition of the implementation. It may include additional determinations or constraints that follow from a technical evaluation of the requirements in relation to what is feasible, available and appropriate in the judgment of the analyst and designer in agreement with the user.

It has long been the aim of computer scientists to provide formal languages for the expression of specifications so as to permit mechanical checking of completeness and consistency [BUX70], [VAN76], [TEI77], [YEH77], [COX80], but a widely accepted language does not yet exist. Given a machinable specification it is conceptually possible to reduce it mechanically to executable [ZUR67] and even efficient [DAR79] code but these technologies too are not yet ready for general exploitation.

Thus, for the time being the specification process will be followed by a *design* phase [COX80], [PET80]. The prime objective of this activity is to identify and structure data, data transformation and data flow [JAC75]. It must also achieve, in some defined sense, optimal partitioning of system function [PAR72], select computational algorithms and procedures, and identify system components and the relationships between them. It is now generally accepted that iterative top down [SWA] analysis and partitioning processes, as, for example, in figure 7, are required to achieve successive refinement [WIR71] of the system design to the point where the identified objects, procedures and transformations can be directly implemented.

5.6.3 Implementation

Following the completion of the design, system *implementation* may begin. In practice, however, design and implementation overlap. Thus as the hierarchical partitioning process proceeds, analysis of certain system elements may be considered sufficient to permit implementation, whilst others clearly require further analysis. In a software project, time always appears to be at a premium. A workforce comprising many different abilities is available and must be kept busy. Thus, regrettably, implementation of sub-systems, components, procedures or modules will be initiated despite

the fact that the overall or even the local design is not yet complete.

As the implementation proceeds code must be *validated* [MIL78], [G0080]. Present day procedures concentrate primarily on *testing* [G0075], though in recent years increasing use has been made of design *walkthrough* and code *inspection* [FAG76]. These later procedures are intended to disclose both design and implementation errors before their consequences become hidden in the program code. The ratio of costs of removing a fault discovered in usage as against the cost of removing the same fault if discovered during the design or first implementation phase is sometimes two or three orders of magnitude. Clearly it pays to find faults early in the process.

In any case, testing by means of program execution is generally achieved bottom up, first at the unit (module or procedural) level, then functionally, component by component. As tested components become available they are then assembled into a system in an *integration* process and *system test* is initiated. Finally, after some degree of independent certification of system function and performance the system is designated ready for *release*.

The above very brief summary has identified some of the activities in the current industrial system development process. Individual activities as described may overlap, be iterated, merged or not undertaken at all. As suggested in Figure 7 design of an element, for example, may be followed immediately by a test implementation and preliminary performance evaluation to ensure feasibility of a design before its implications spread to other parts of the system. Clearly there should be a set of overall controlled procedures to take a concept from the first pragmatic evaluation of the potential of an application for mechanisation to final program product executing in defined hardware or software and hardware environment(s).

5.6.4 Maintenance

Once the system has been released the maintenance process begins. Faults will be observed, reported and corrected. Where appropriate, repairs to the code, to the documentation or both will be authorised. If user progress is blocked because of a fault, a temporary by-pass of the faulty code may be authorised. In other circumstances a temporary or permanent fix to the code may be applied in some or all user

locations. The permanent repair or change to the program can then be held over for a new release of the system. In other cases a permanent change will be prepared for immediate installation by all those running the system. The particular strategy adopted in any instance will depend on the nature and severity of the fault, the size and difficulty of the change required, the number and nature of the program installations and user organisations, and so on. The actual strategy will have a profound impact on the rate of system complexity growth, on its life cycle costs and on its life expectancy.

The faults that are fixed in the maintenance process may be due to changes external to the system, incorrect or incomplete specification, design or implementation errors, hardware changes or to some combination of these. Since each user exposes the system in different ways, all installations do not experience all faults, nor do they automatically apply all manufacturer-supplied fixes or changes. On the other hand, installations having their own programming staff may very well develop and install local changes or system modifications to suit their specific needs. These patches, insertions or deletions may in turn cause new difficulties when further incremental changes are received from the manufacturer, or at a later date when a new release is received. The inevitable consequences of the maintenance process applied to systems installed for more than one user, is that the many installed instances of a system drift apart. Multiple versions of system elements develop to encompass the variations and combinations [BEL77b]. System *configuration* management becomes a major task. *Support environments* [DOL76], [HUT77], [BUX80] that automatically collect and maintain total activity records become an essential tool in programming process management.

5.7 Life Cycle Planning and Management

Section 4 considered one aspect of the planning process. The preceding discussion, while presenting a simplified view of the life cycle, will have made clear the difficulty associated with life-time planning. In recent years the problem has received much attention, for example, two meetings dedicated to the theme of Software Life Cycle Management [LEH77b], [BAS78]. A variety of techniques have been developed to improve estimation of cost, time and other resources required for software development and maintenance [[FEL77], [PUT77a], [BOE78a], [PAR80], [ARO80]. These techniques are based on extrapolation of past experience and

tend to be a self-fulfilling prophecy. In general, it has not yet proved possible to develop techniques a priori project resource estimation on the basis of objective measurement of such attributes as application complexity and of the work needed to create a satisfactory system. Techniques such as software science [HAL77], [FIT78] seek to do just this but to date lack substantiation [JOH67] and interpretation. Major research and advances are required if software engineering is to become as manageable as are other engineering disciplines, though fundamentally the peculiar nature of software [LEH77c] will always leave its engineering in a class of its own.

6 Conclusion

This paper has presented the concepts and implications of evolving programs, dynamics of evolution and the program life cycle. Through this it has supported the view, first expressed in Garmisch [NAU68], but still as valid, that there is an urgent need for a discipline of Software Engineering. This should facilitate the cost-effective planning, design, construction and maintenance of effective programs that provide, and continue to provide, valid solutions to stated (possibly changing) problems or satisfactory implementations of (possibly changing) computer applications.

Recognising the intrinsic nature of program change, the laws that appear to describe the dynamics of the evolution process and their technical and managerial implications were outlined, and their application to the planning process illustrated. In then reviewing the concepts, significance and phases of the program life cycle details of life cycle planning and management models were omitted. In particular, cost, resource and reliability models [BOE76a], [PAR77] [MUS80] have not been examined. Nor have examples from the many systems, other than system x, that have been studied, been provided. Approaches to process modelling based on continuous models [RIO77], [WOO70a] have also not been included, nor has the vital topic of software complexity [MCC76], [LEH77a], [BEL77a], [CHE78], [MUS80], [BEL77c].

Despite this limited coverage of relevant material it is hoped that the reader's interest and concern has been aroused. Understanding of the underlying concepts and recognition of the implications should make him pursue further details in the works listed in the references. Hopefully he will then be encouraged and enabled to apply the concepts to achieve more effective software projects, system

management and implementation. Through this his clients will also achieve more effective, cost-effective and continuing computer usage.

It would seem that many of the concepts and techniques presented in this paper could find wide applications outside the specific area of software systems, in other industries and to social and economic systems. Clearly they could not be pursued here.

7 Acknowledgements

First and foremost my thanks must be extended to L A Belady, my close collaborator for almost 10 years. Many others, particularly associates and colleagues at ALMSA, IBM, Imperial College and WG2.3 have contributed through their comments, questions, critique and original thoughts. All of them deserve and receive my grateful acknowledgements and thanks for their individual and collective contributions. May I be permitted to single out Professor W M Turski for the major contribution made on his recent visit to London. My sincere thanks also to him, Drs G Benyon-Tinker, P G Harrison and C Jones for their detailed and constructive criticism of an early draft of this paper, R Bailey for his artistic support and Miss W Huxtable for the patient typing and retyping of the manuscript. Finally, may I acknowledge the constant support of my wife without which neither the work itself nor this paper would have been possible.

CHAPTER 20

THE ENVIRONMENT OF PROGRAM DEVELOPMENT AND MAINTENANCE - PROGRAMS, PROGRAMMING AND PROGRAMMING SUPPORT*

1 Programs

A computer program may be defined as:-

... a statement of data attributes and associated algorithms that completely defines a mechanical procedure to implement a computer application or to obtain a solution to a problem by execution in a prescribed environment.

This definition has limitations. It does not, for example, address the question of whether a program that is incorrect or incomplete in relation to some problem statement is to be considered a program. This alone suggests that a more complete definition must be developed. There is, however, a more fundamental weakness. As formulated, the definition provides a basis for deciding whether a given symbol sequence is a program. In no way, however, does it help in the creation of a new program or show how to modify an existing instruction sequence such that it becomes a correct program in relation to some problem statement. Yet the prime problem faced by the computing community today is to discover methods and to develop tools that facilitate the *creation* and *maintenance* of functionally satisfactory and cost effective programs.

A more satisfactory definition may perhaps be achieved by first defining the *activity* of programming and then defining programs as the output of that process [LEH69]. To be able to do this pre-supposes that one knows how best to approach the total programming process in general, or the creation of a program for a specific application, in particular. But, precisely this has been recognised as the basic problem of the programming community for at least twelve years [NAU69].

Current programming methodology has evolved dynamically from the experiences of the past thirty years and represents a motley collection of relatively isolated methodologies, methods and techniques, associated through an experienced-based, but otherwise arbitrary, sequence of much discussed process phases [BOE76]. There does not exist at present even a heuristic understanding of what is *really required* to transform an application concept into a correct, usable and cost-effective program. A systematic total process (20.1) based on a scientific theory of programming appears remote.

The present paper represents the beginnings of an attempt to understand and describe the process *as a whole* on the basis of the intrinsic work that must be done in creating a program. As a first step we consider a program classification scheme named SPE, recently introduced [LEH80].

An *S-program* is a program for which the absolute criterion of process completion and program acceptability is satisfaction of a given specification. For a program to be of class S, a problem statement from which a complete and authoritative specification of a program for its solution must be possible and can be derived. In the past success in creating programs even such as these, has been determined exclusively through test executions of the program and its components, under conditions for which the computational result can be precisely predicted [G0080]. More recently testing has been preceded by formally organised program and code *inspections* [FAG76]. Industry in general and the majority of programmers still rely primarily on such procedures to determine the acceptability of a program. If a specification exists at all it is, at most, used to determine the set of tests that are to be applied. The process as briefly described is illustrated in Figure 1.

In the last decade it has been increasingly recognised that for a variety of reasons, testing is an unsatisfactory basis for demonstrating the veracity of a program. As Dijkstra puts it '*... testing can only show the presence of faults never their absence*' [DIJ72]. Thus the concept of program *verification* has been introduced. Techniques are slowly being developed for proving a program correct, that is demonstrating that it completely satisfies its specification.

20.1 (Eds) *Now referred to as a 'coherent process'*.

Techniques for achieving and demonstrating correctness progressively during the process of program development [DIJ78] using for example techniques such as program assertion. [HOA69] will prove even more significant in the future.

In essence any program correctness proof demonstrates equivalence, in some sense, between the program and its specification; no specification, no proof. Thus since by definition an S-program must have a formal specification, the property of 'correctness' (if present) is inherently demonstrable, though the proof may not be known. The S-program process may be represented as in Figure 2. The striking change in comparison to Figure 1 is that the *comparison* of data derived from the problem description with that from program execution is replaced by calculable *verification* of the program text by its derivation from the *authoritative* specification.

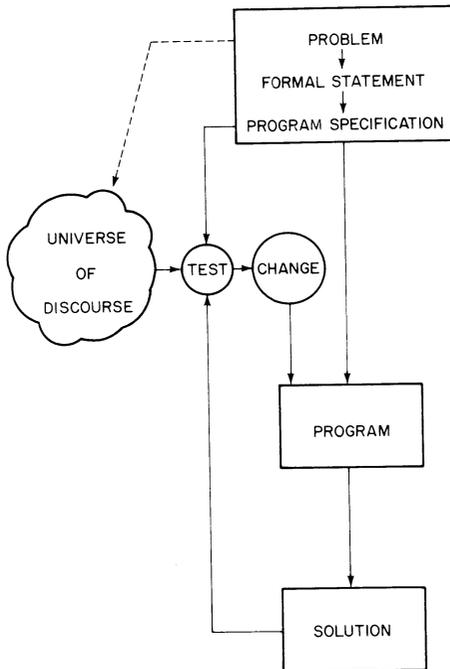


Figure 1: S-Program Traditional Approach

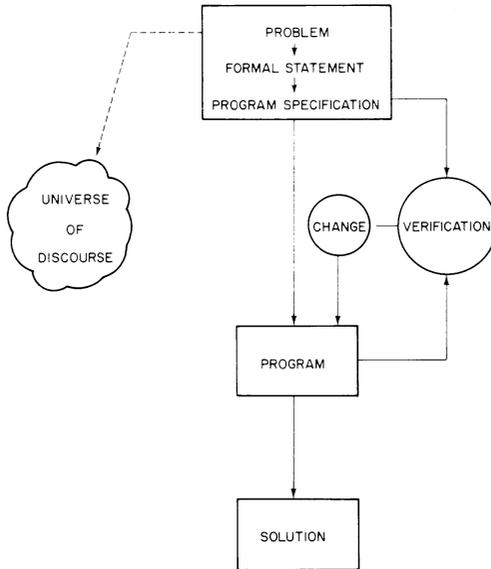


Figure 2: S-Program Verification

It may happen that the problem to be solved is not just of intellectual interest, but relates to some real world phenomenon. In that event the solution may be found to be unsatisfying, even *after* the program has been proven correct. When this happens the problem statement must be incorrect in some aspect or, assuming no derivation errors, inappropriate. It does not represent the situation of current concern. Thus a new problem statement and specification must be produced and from them a new program derived. It may be that these are all achieved by perturbation or extension of the original versions, but nevertheless they represent the implicit definition and derivation of a *new* program.

The process of evaluating a program in relation to the users' needs is termed *validation*, and we may now view the general process as in Figure 3. Verification and validation and their relationship, to one another and to the programming process are further discussed below.

P-programs are defined to differ from S-programs in that their satisfying a specification can ultimately not be accepted as the criterion of validity. This occurs when, for example, solutions to the problem that they define are to be

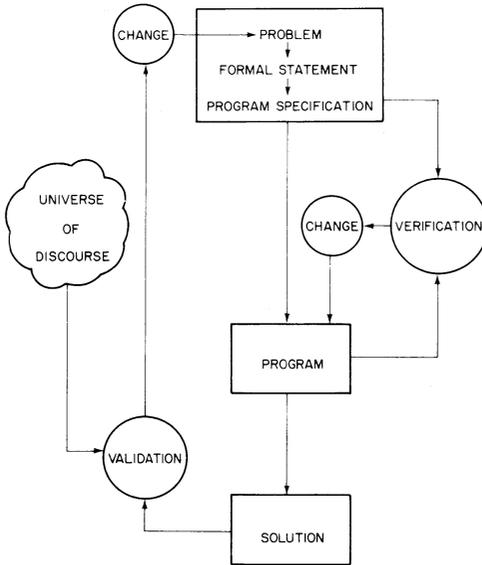


Figure 3: Full S-Program Process

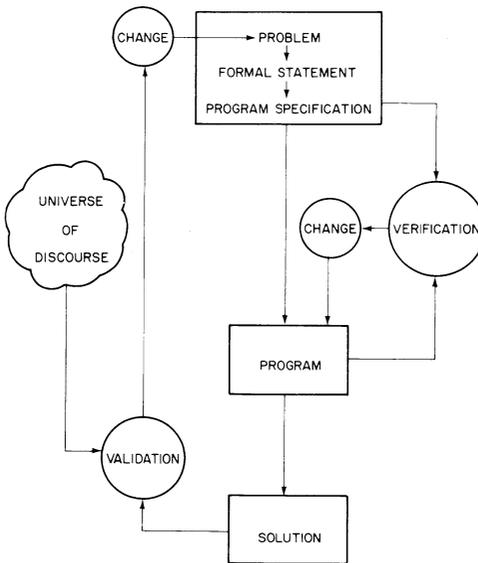


Figure 4: P-program Process

used in some application. It is the *relevance* of the computational output in the domain of concern that is decisive. Provable satisfaction of a specification, whilst encouraging for the programmer, may well be irrelevant if the specification is inadequate.

In this situation, illustrated by Figure 4, one cannot expect to achieve an acceptable program without some iteration. The problem analysis will almost certainly be incomplete; will for example make unjustified assumptions or approximations in its abstraction of the environment. One may also wish or need to change the program implementation to achieve more effective or cost-effective problem solution. Changes will suggest themselves as the program is executed, that is, as it is validated in test executions or in actual use; as a *better understanding is acquired of the real problem to be solved*. The consequent adaptations are applied iteratively and must take time. As the process proceeds the universe to which the problem statement refers is itself likely to change. That change too is likely to demand or provide opportunity for change to the problem statement and hence to the program. The program will tend to undergo extensive, if not continuing, change.

In this case one might also take the view that successive programs are *new* programs reflecting problems different to that originally stated. However, since the basic problem belongs to a class that relates to actual needs as ultimately experienced in practice, rather than being primarily of intellectual interest, it is more realistic to see the process as a gradual approach to the identifications and formulation of the problem and its solution. The resultant P-program is best viewed as evolving rather than as a sequence of new programs.

Finally we consider the defining characteristic of *E-programs*. These implement applications that in some sense control activities and or events in the environments within which they are embedded and executed. That is, an E-program is itself a part of the universe that it both models and controls. Implicitly at least, it contains a model of itself or rather of its interactions with its operational environment. Figure 5 illustrates the basic closed loop that is intrinsic to the development and use of E-programs.

In order to specify the properties of an E-program that is to be developed or modified, one must therefore predict its impact on its operational environment and on the people who

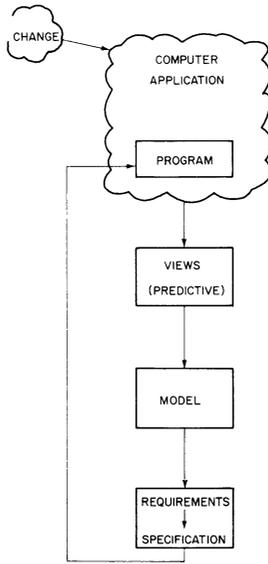


Figure 5a: E-program Basic Loop

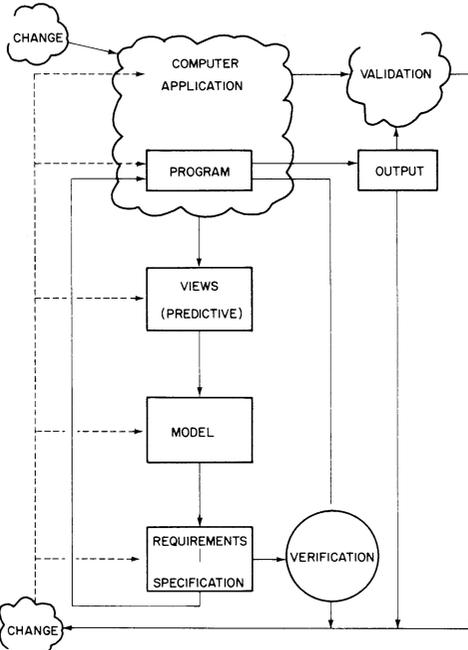


Figure 5b: Full E-Program Process

are to work with it. The designer must perceive the application activity and the environment within which it occurs as they *will be* once the system has been implemented and installed. The perception cannot be precise, the prediction cannot be perfect and system validation will inevitably result in a series of changes to the system. The system *must* evolve, changing the operational environment as it does so by virtue of its being embedded in and therefore a part of that environment. That change in turn, generates further evolutionary pressure. The process of change and evolution is never ending, the alternative being increasing ineffectiveness and ultimate obsolescence.

In passing we note that in fact all natural and artificial systems evolve [SIM69]. However, because of the tight coupling of computer applications and of software with humans, society and their activities, and because of the superficially apparent simplicity of the process of program change, the rate of software evolution is very high, absolutely and relative to the professional life-time of practitioners. Hence there arises the impression and reality of continuing change with all its implications on effectiveness, reliability and cost.

A programming manager's principle function is, in a very general sense, the management of evolutionary change. The characteristics of program evolution have been extensively studied in recent years, [LEH80], [BEL77] and they are not further discussed here. Instead we recognise that program evolution is the visible response of a closed-loop feedback system. The stability and rate of change of that evolution is a consequence of the phenomenological nature of that loop, of the characteristics of program creation and the processes whereby this is achieved. All of these need to be better and more fundamentally understood than is presently the case. We pursue this understanding in the remainder of the paper.

2 The State-of-the-Art in Programming Support; Foreseeable Developments

Continuing evolution must in practice, be achieved by a sequence of activities as discussed below. To direct and control the resultant process over the lifetime of a system requires abilities and facilities above and beyond those needed for the design and implementation of data structures, program structures and code sequences. It is recognition of this fact that has generated the wide current interest in programming support tools and environments - PSE's.

The state of the art in programming support systems sees them as having two prime functions. In the first place they provide documentation capability equivalent to the records, blue prints, card indexes, filing cabinets, typists and filing and retrieval clerks found in every conventional engineering project. The special power of the computer is then to be able to provide additional capabilities for relating entries to each other, for automatic searching, sorting and selection and for selective presentation. Secondly, it can include a variety of software tools for creating, checking, editing, converting (compiling, interpreting, assembling) and linking system requirements, specifications and program text and submitting the latter for execution. That is, the environment can contain, integrate and associate with its data collection and filing capability all those programming tools previously seen by the programmer as isolated aids to program definition, construction, validation, submission, documentation and maintenance.

There exist proprietary systems that provide additional facilities [DOL76], [HUT79]. Basically, however, the published work on programming environments reflects only a limited advance in programming methodology, systems engineering and project planning. In addition to the provision of data-base facilities that provide a permanent record of all activities in relation to evolving programs, environments are now viewed as integrated tool kits that automate and relate what were previously regarded as individual manual methodologies and tools; facilities such as program entry, editing, translation and loading, previously viewed as largely independent.

The new Stoneman proposal [BUX80] collects, consolidates and harmonises current concepts to present an environment definition that is meaningful and realistic in the context of the state of the art. It does not, however, call for major extensions of current techniques or for the invention and practical development of new concepts or of old concepts applied in new ways or on a new scale. That is as it should be. The Ada requirement for the next years is for methodologies and support systems that work and work effectively. They must achieve more effective, responsive and cost-effective development and maintenance of software for embedded systems even if, for the moment, they do not attain the maximum advance that technology could offer.

But there already exist much broader views of the potential for and of programming and software engineering development

methods, techniques and tools and their integration into a support system or environment [LEH80]. The views represent a conceptual generalisation which sees the environment as providing *all* the support and fulfilling all the functions that are provided by the total industrial complex in manufacturing industries. Design and drawing offices, tool rooms, machine shops, component stores, test laboratories, for example, provide essential yet unobtrusive support for the production line that produces the marketable artifact. In many industries these indirect support facilities absorb a major part of corporate expenditure. They may be set up as separate, semi-independent, entities because interaction between them occur at intervals measured in days and weeks. For software production, the level and rate of human interaction with the process is less material, more frequent and less visible. The only effective way for the provision of analogous support is, therefore, through the creation of an integrated, computerised system that automatically records all events and provides immediately accessible support facilities and tools as required.

Some aspects and implications of this new view are discussed below. They include at the one extreme, technologies that are currently being explored in isolation but that may be visualised as being integrated into, and applied within, the framework of a programming support environment. At the other extreme, there are the concepts and the methodological potential that arises from a recognition of the inadequacies of the current software engineering process and of the current view of the role of the computer as a programming and software engineering support tool. This leads to an integrated view of the total process as the transformation and continuing adaptation of an application potential with stated requirements into a system that is maintained satisfactorily as the operational and technological environment and the experience base change. It leads directly to to recognition of the computer as the basic support tool and development environment.

3 Computing Application Development

The process of implementing a computer application by creating or enhancing a software system consists of several overlapped and iterative phases that include activities such as requirements analysis, specification, design, implementation, component test, integration and system test [LEH80]. The sequential activities when iteratively executed ultimately produce the executable code and the accompanying

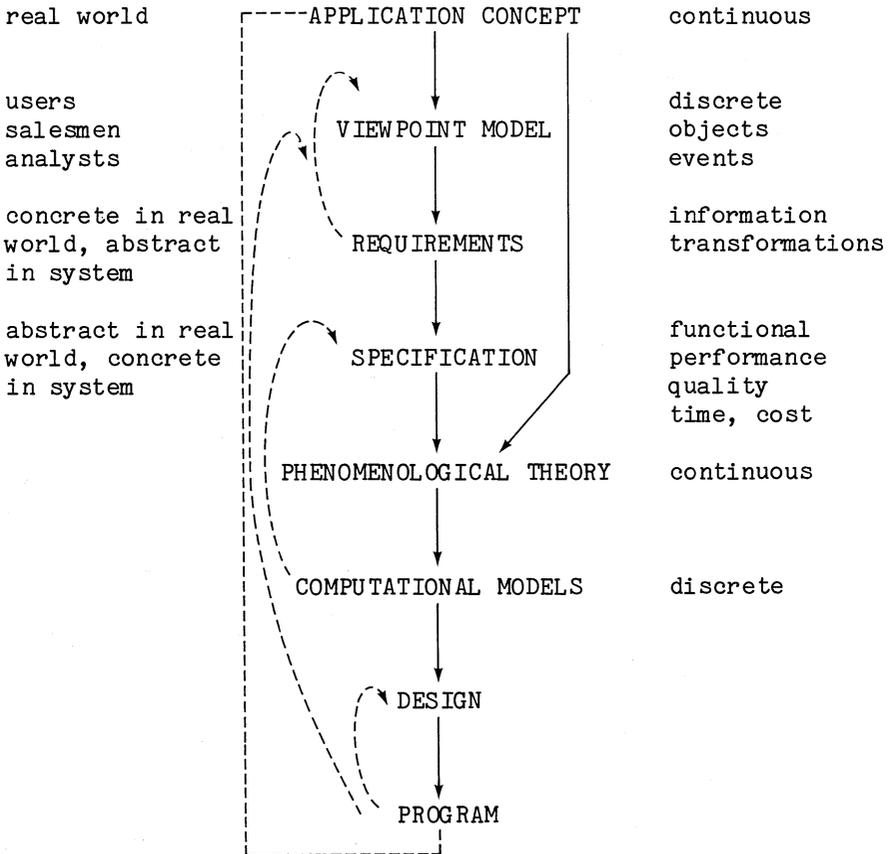
documentation that comprise the initial product. Subsequently, a maintenance phase directs and controls system evolution.

A common view of these phased processes sees each as a discrete entity having output produced by an appropriate methodology. That output then forms part of the input to the next phase. The application concept as detailed by the potential user constitutes the input to the first phase. In addition each phase should result in documented records that provide a factual historic record of the objectives, considerations, decisions and actions that underlie the design and construction of phase-products. In practice, of course, there will be considerable deviation from this ideal, in particular, overlap and iteration between phases.

There exists, however, an alternative viewpoint; first expressed by Zurcher and Randell in 1968 [ZUR68]. The total programming process is, or is equivalent to, a series of transformations of *representational* models. These transformations involve distinct conceptual levels as exemplified in Figure 6. Degenerate cases with less than that number of levels do occur but the precise number is of no consequence for present considerations.

The model at any one conceptual level will essentially contain all the detail relevant at that level. Often one will wish to examine models with much of the detail obscured, displaying only those aspects that are of interest at that moment. Subsequently, one will wish to display or add further detail. Thus, the representational models should themselves be constructed to reflect levels of detail. The support environment may then contain the equivalent of a zoom lens to permit an overview of any of the conceptual levels and then the addition of details as these are desired.

With current and foreseeable methods the transformation process must involve iteration. In the absence of a theory spanning the total process, reversions to higher level models must occur as the development and evaluation of lower level ones identifies changes in or additions to them and thence to higher level models. The resultant iterations may span any number of models from neighbouring pairs to the entire sequence. Figure 7 tries to illustrate this generality by suggesting that alternative versions of each model are conceivable at each level. The iterative design process consists of achieving an optimum trajectory for the total transformation process. With present methods this involves



An E-program is, *at least*, a model of an application in its evolving environment

Figure 6: Levels of the Development Process

successive perturbations to neighbouring models; horizontally at any one level or diagonally between levels, as illustrated in Figure 7.

In the long term it seems likely that, at least for certain classes of applications, that once the requirements model has been established, subsequent transformations can be linked and mechanised. At most they should require only occasional

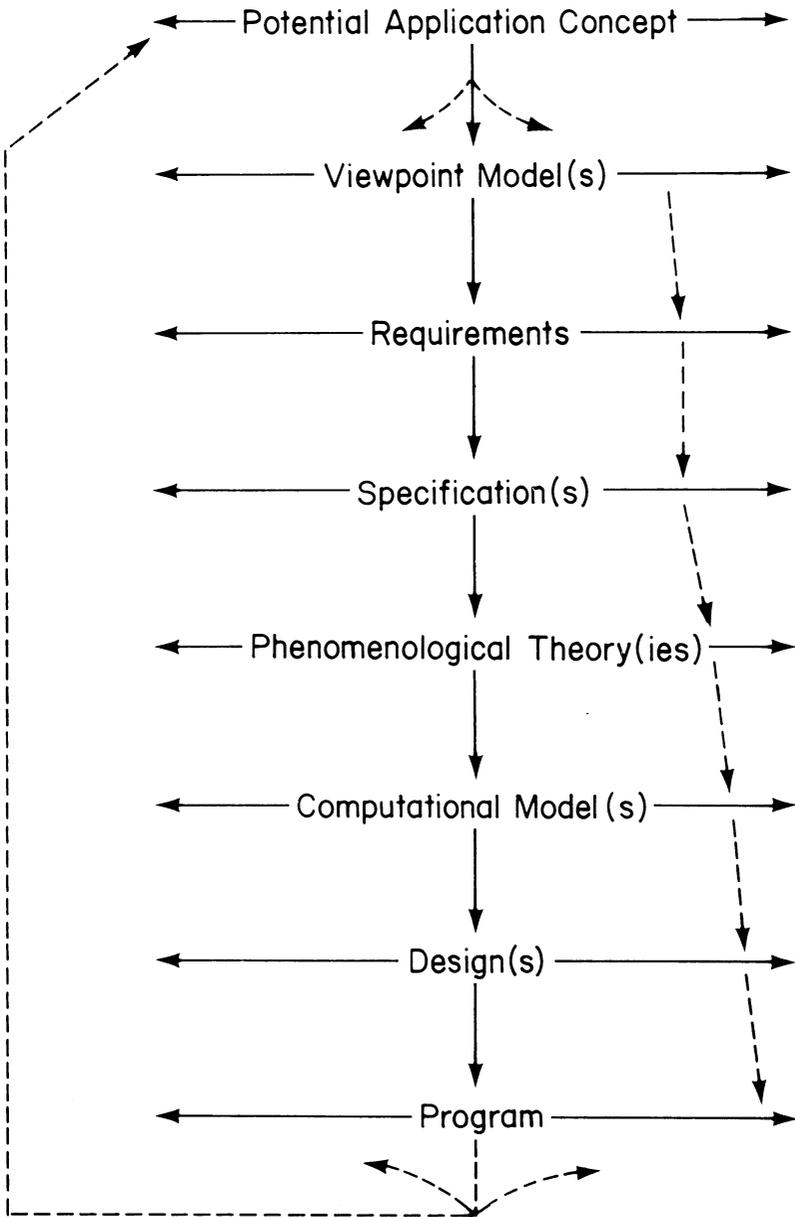


Figure 7: Development Trajectory

human intervention. Indeed limited transformation systems [DAR79] or, systems that produce executable code from *specifications* expressed in predicate logic [KOW79] already exist.

For the general case, however, human assessment, interpretation, guidance and decision will be required during most, if not all, of these transformations. This requires the person(s) involved to fully understand system and process objectives. Satisfaction in this context will have many aspects; structure, correctness, understandability, performance, reliability, projected costs, ease (or otherwise) of further continuation of process and so on. That is, progress through the process must be guided by the feasibility, and functional and economic acceptability, of what has been created so far, and its potential for future development.

Two classes of activity are involved in this assessment of each representational model; *verification* and *validation*. We note that these separate concerns replace the unified concepts of, for example, inspection of a design or testing of a program.

Verification will address the *consistency* and *completeness* of each model both in itself and in relation to its parent or source models; that is the *correctness* of the transformation process and the detail that has been added, and therefore of the model itself. Such verification will, in the future, be increasingly based on formal techniques requiring formal representation of the models and the transformation processes. Moreover, tools to support this activity will become an important component of PSEs.

Validation, on the other hand, is a *judgemental* activity based on *measurement* and human assessment of *expected effectiveness, usability, value* and *cost effectiveness* in the *operational* environment. Thus it is intrinsically predictive, having to be based on incomplete and imprecise *metric models* of the future operational environment as it will be after installation and during use of an, as yet non-existent system [LEH80]. Key questions to be answered must address the extent to which *metric* models may be implemented as *executable* models embedded in a PSE, other purposes for which such models may then be exploited and how they may be interfaced and integrated with representational models to achieve a tool-supported total process.

4 Executable Models

The previous section introduced representational and executable models as concrete embodiments of stages that an application concept has to pass through as it is converted into a usable, acceptable and cost-effective system. The former *are* stages of developmental evolution; indeed in their original paper [ZUR68] Zurcher and Randell wrote of 'the model *becoming* the system'. The latter are entities that permit assessment at each stage, of what has been achieved and a prognosis for the future.

The two classes of model need not, however, be implemented as separate entities. For example, it may be possible to construct a simulation model that includes a complete and faithful representation of the design as developed to some level. This system model will be embedded in a (parameterised) driver that also includes models of the anticipated operational computing system and external environments and interfaces. Thus execution of the representational model under desired environmental test conditions permits its, at least partial, validation.

Nor are models restricted to representation and validation of the system *product* and its *operational environments*. From the point of view of the project manager, *process* models for the estimation, planning and control of resources, time and cost are equally vital to success. Such models, whether analytic, statistical or simulation, must be based both on an understanding of the process and on historic system, project and organisation data [LEH80]. Clearly programming environments may include tools for the collection and storage of such data and for the construction and interrogation of derived models.

The role of organisational structure in the determination of system characteristics must also not be ignored. It is generally accepted that the structure of at least one major operating system was a direct reflection of the organisation that created it. That system's structure was undoubtedly a major contributing factor to the very high cost and the major difficulties encountered in its subsequent maintenance. In any event, in a large project, it may be profitable to maintain and relate models of the system and organisational structures. Their usage and support constitutes a potentially rewarding technology that must be explored in the context of PSE design.

Some examples of possible uses of executable models follow.

Exercising and Training

One of the problems in Requirements development and computing system specification is that both must be developed from a view of the operational environment that will exist *after* system *installation*. The installation and use of the system, however, changes that environment. For this reason, and because human reactions to system characteristics are unpredictable, indeterminacy and evolution are intrinsic properties of computing applications and software systems [LEH77]. Clearly an appropriate up-to-date system model that provides a feel for ultimate system behaviour, an exercisor, can reduce developmental instability and speed up convergence to a satisfactory system. Such models will also be usable for training purposes both during and after commissioning, much as simulators are used in pilot training.

Performance Evaluation

The potential role of simulation for predictive performance assessment during design has been perceived for some time [SCH66], [LEH68]. It has, however, not been widely adopted. One reason for this is that, despite Zurcher and Randell [ZUR68], the simulation model was seen as strictly separated from the representational model. In practice its development demanded a resource investment often larger than that needed for the latter. Thus, it becomes important to develop and investigate techniques and tools that will facilitate the development of performance assessment models and their integration with those for development and implementation.

Project and Process Planning

Investigations of the dynamics of software system evolution have shown that this is strongly influenced by historical factors [LEH80]. For effective management control, data relating to system, process and organisation must be collected, and made available for modelling and interrogation. The phenomena to be measured are reflected in programmer and managerial interactions with environments, even as currently conceived. It is however necessary to determine to what extent construction and interrogation of such models may be tool-supported within a PSE.

Organisational Grouping

A project involving more than about five people must be structured into separately managed groups. Each of these is then allocated a sub-objective, that is a sub-project within the main project. As the project progresses, however the group products must increasingly interface and be capable of communicating with one another during system operation. Each linkage between sub-systems necessitates human communication between, at least, the groups responsible for implementing them during system design, construction and modification. If the system is to be successfully and economically integrated and maintained, all such communication must be precisely and accessibly recorded. The complexity of these records relates to that of both system-internal and organisational communication structures. Mismatch will cause rapid complexity growth with all the penalties that such growth implies.

It is important to attempt to keep system and organisation structures aligned. The latter should develop dynamically and be modified or changed as design proceeds and the project grows, to minimise misalignment and the need for inter-group communication. It is suggested that models of the two structures can be used to develop system structure and control organisational structure to approach the desired degree of optimisation.

The data from which the above models can be derived, will be available in the PSE data base. In part, at least, they will have been captured during human interactions with that system. What models are possible, and/or required, how they are to be constructed, how their construction itself may be tool-supported and how they might be used, requires further study.

5 Summary

Programs may be classified into those that are inherently static, those that are likely to evolve and those that must inevitably evolve. The paper suggests that, as stated in the first law of software engineering [LEH80], programs that are used will evolve or decay into uselessness. Recognition of the inevitability of change is sufficient to demand that the programming process be supported by an integrated computerised support system providing a data base facility and an extendable tool kit.

Programming support environments currently in operation [DOL76], [HUT79], under development [PEA79], or as perceived [RID80] do not provide tools that explicitly address the process as described. In particular, they do not include or propose integrated facilities for model transformation, verification and validation. The Stonemen document [BUX80] also omits major reference to these issues since DoD's principle concern must have been to ensure that a state-of-the-art system is available when Ada enters into full service in the mid-eighties. Just as recently the European Space Agency issued a call for tenders [GOE80] to study its specific programming process and to determine the degree to which it might be automated. That call makes specific reference to '... verify completeness and consistency of requirements ... that the design satisfies them ... tools integrated into a single system ...'. But basically current proposals have been restricted to providing a system and design data base and the now classical tools for preparing, editing, and loading program text and documentation.

In the long term, current concepts of PSE properties are too limiting. To fully support the total program development and maintenance process over a system's life cycle, to increase the responsiveness of the programming process and to reduce the high cost of software, the capabilities of advanced evolutionary programming support environments should be significantly extended. They should support the methods used to design a sequence of representational system models. Equally they should address methods and provide techniques and tools that permit and facilitate verification and validation, and that support project management, process planning and control. There is good reason to suppose that executable metric models will have a significant role to play in the programming process.

6 Acknowledgements

The author gratefully acknowledges the continuing and critical support of his immediate colleagues Drs G Benyon-Tinker and P G Harrison, and of his close associates L A Belady and Professor W M Turski. Also the support of the European Research Office under grant DAJA-37-80-C0011 and the encouragement of Mr G M Sokol, its Chief of Communications Engineering and Information Sciences.

CHAPTER 21

PROGRAMMING PRODUCTIVITY - A LIFE CYCLE CONCEPT*

1 Introduction

1.1 Productivity and Productivity Measures

Productivity is primarily an economic concept. It addresses the problem of optimisation of *return on investment* in connection with activity on or with some artifact. Investment and return are themselves concepts with many connotations and interpretations. Thus the focus of any concerned individual must be on that aspect of productivity that relates to his area of responsibility or interest; production rate, financial performance, resource utilisation, time, people and so on. These alternative foci can inherently not be unified. In general, productivity cannot be maximised simultaneously in every dimension of the multi-dimensional space in which it could be defined, though gains achieved in one will often be beneficial in others.

The problem is compounded when a multi-step process is considered; even more so when iteration is present. In fact, meaningful definition of productivity over an activity requires that the *product* and its associated *process* are so *defined* and *structured* that the effectiveness of the activity does not depend significantly on a predecessor activity. Equally the *quality* of the output of any activity must not significantly affect the effectiveness of a successor activity. Where these conditions hold, an assembly line process can be said to exist [BEL79b] and overall productivity may well be a linear combination of that obtained on individual steps. If, on the other hand, some characteristic on one step strongly influences a subsequent step, productivity concepts must relate to and be defined over the entire activity within and over which interactions occur. Over that activity at least, the process may be said to be *non-linear*.

Moreover absolute, quantitative productivity measures are, to a first order, meaningful only when the output from an

activity does not depend on external factors, unless definition of the measure specifically takes input attributes into account. Furthermore, the quality of a product and the productivity of the process that produces it are, in fact, inter-dependent and cannot, in general, be optimised independently. If productivity measures are to be meaningful, either output attributes are so controlled that there exists no unacceptable variability in product quality or compound measures of both can be defined to take inter-dependencies into account.

1.2 Productivity in the Software Process

The theme of this conference is productivity in relation to the conception, development, construction, application, maintenance and evolution of computing systems; systems of which software forms a significant part. This paper will not, however, address productivity and quality issues in the engineering of such systems. Nor does it discuss the specific properties of software systems that make consideration of these issues an urgent but elusive area of concern. Instead it considers the nature of the *Programming Process*, the process whereby software systems are developed and maintained. As implied above, understanding its nature and role is crucial for productive consideration of productivity in the computer industry in general and in computer applications development through increased programming productivity in particular.

Statistics indicate that of the total lifetime expenditure on a software system, some 70% (+20%) is spent after first installation in the operational environment. In practice there is considerable opportunity for trade-off between the various phases of the total life-cycle. The individual manager seeking to meet time targets or to reduce expenditure attributable to him may cause a significant decrease of the quality or productivity associated with *subsequent* work on the system or increased cost of that work, for example by permitting the introduction of unnecessary complexity or through careless fault eradication. The present day software process over the system's life time is, in fact, highly non-linear. Decisive comprehensible specification may, for example, *simplify* the design process that follows. Incomplete or inconsistent requirements selection or specification will make the subsequent design and also coding, testing and integration more *difficult*. Design weaknesses or omissions may be detected and eliminated early in the design process or may be left for detection during

testing, integration or after the system becomes operational. Attention paid to system structure during design, will determine the ease or difficulty of subsequent modification of the system and the rate at which that difficulty will increase. In all these instances the policy pursued will have a strong but distributed influence on maintenance characteristics, on life-cycle cost and on productivity, however defined.

Process trade-offs are, however, not often evaluated in terms of their life-cycle impact on product quality or process productivity. Instead they are based on local perspectives. Planners and managers allocate time allotments and divide other resources to achieve local or personal optimisation. Productivity measures within the existing process must therefore be treated with some suspicion. Nevertheless one may ask whether the process, as currently practiced, may be made more effective. Discussion of this issue is postponed to Section 4 where it can be addressed in terms of the viewpoints developed in the earlier sections. That discussion will conclude that productivity studies can assess time-local gains only if they relate to *natural* process functions that are, in some sense, orthogonal. It demands full understanding of sub-processes, their interactions and the way they interface and integrate to form a *total process*.

This contribution therefore concentrates on analysis of the total process required for evolution of a software system, seeking to determine fundamental activities and structures. The resultant understanding may be expected to provide a framework for further study of the systems process and elements of a future process theory. It will facilitate development of more effective and efficient procedures. To focus ideas concepts are expressed specifically in terms of the programming process. Many of the conclusions will, however, also apply to the development of larger systems within which software is embedded and which, in some sense, it controls.

1.3 The Approach

Current practice in software design and maintenance has evolved gradually over the past thirty years. Methods and tools have been developed gradually and individually for specific aspects of the process in specific environments. Only afterwards have they been adapted, combined and used, iteratively and sequentially, within some global implementation environment to achieve a target system within that

environment. This ad hoc, bottom-up, synthesis of a total process cannot be expected to achieve a separation of concerns or simple, defined, interfaces between process steps. It tends to destroy local information, not to hide it. It does not yield a homogeneous process. It is no wonder that progress towards an integrated engineering science and discipline of software systems engineering has been so painfully slow. There is no reason to suppose that a bottom-up approach in this area is likely to be any more effective than it is in creating or maintaining correct programs.

A top-down analysis is presented as an alternative approach. It seeks to separate concerns by identifying and isolating the concepts, representations and transformations that are intrinsically part of the process of converting a computer application concept into an operational system. Given such separation, information relevant only to an individual step of the process may be hidden within the methods tools and information storage facilities of that step. Information relevant to neighbouring steps must be made available via *totally defined interfaces* to establish the communication that is necessary to drive and guide the process and to transmit the information that will eventually define the operational system.

This approach to *process design* can be expected to yield a reliable and effective process that may be doubly integrated; over the development cycle from system, or system-change conception, to its operational installation and over system life-time evolution as the application environment and the user perception of system potential evolve. When formulation and formalisation of an integrated and complete process is achieved, vertically integrated programming support environments become feasible. Moreover when adequate separation and definition of process sub-objectives is achieved, one may attain an essentially sequential process. In such a process, the *ideal process*, there should rarely be a need to repeat or amend the output of an earlier activity or step. Productivity concepts relating to individual steps may be expected to be meaningful, in that the achievement in any one is largely independent of and buffered from the others. No greater contribution to the search for increased productivity in the development and maintenance of software can be visualised than that which results from complete understanding of the necessary and sufficient activities required to transform a computer application concept into an effective operational system; and to maintain it in that state as the environment changes.

2 Programming Process Models

2.1 Basic Model

The most abstract view of the program development process may be described as "The Transformation of a Computer Application Concept into an Operational System (Program)". This is illustrated in figure 1.

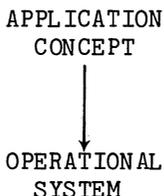


Figure 1 Most Abstract Model of Programming Process

2.2 Alternative Process Decompositions

Many alternative decompositions of this model are possible. The so called waterfall model [BOE76] and its derivatives are representative of the most common current view. They describe the process that has evolved over the years, expressing the broad pragmatics of the learning experience of mankind's first steps in software engineering. They therefore provide a phenomenological base from which a science and engineering discipline can develop. Because they represent a bottom-up synthesis of evolving practice, they are however unlikely to prove adequate as a direct base for the disciplined development of an ideal process or of a close approximation to it.

An alternative approach could partition the process on the basis of the various distinct levels of formal, programming, languages that have been developed over the years. Requirements definition, specification, design, high-level programming and assembly languages each reflect a process objective. They mirror and facilitate human intellectual involvement in the transformation process and, as such, elementary and possibly fundamental ingredients of the process of program creation. But these languages have also evolved as a *consequence* of mankind's increasing understanding of the nature and process of problem solving with the aid of computers. Using existing languages as the base for a process analysis also constitutes a bottom-up approach that puts the cart before the horse.

In a paper describing the development of THE system [DIJ68c], Dijkstra already recognised the need for a disciplined, step by step, approach to software system development. That system was in fact developed from a recognition of six levels, differentiated primarily by the time scale needed to measure and discriminate between events and actions at each level. Thus, for example, at the lowest level the system clock must discriminate between events, interrupts for example, separated by a micro-second or less. On the other hand at the highest level, the level concerned with operator actions, events are separated by, at least, tens of seconds.

Dijkstra's layering of THE system and the process by which its final structure evolved certainly involved a partitioning of the total design and implementation process. What he really contributed, however, was to demonstrate, as did Zurcher and Randell at about the same time [ZUR67], that one may conceive of many levels of primitives in which to express a design. In fact a process of *stepwise refinement* [WIR71] leads one via successively more basic primitives to that level at which the primitives can be implemented as actual executable mechanisms. If the gap between successive levels is not too wide, if the structure of the system at each level and the primitives from which it is to be constructed are fairly self-evident, if in general the system is not too complex, one may perhaps achieve the descent between neighbouring levels directly with a simple transformation as suggested by figure 1. In the general case however, this transformation is itself compound, a succession of the process elements that we seek.

For the reasons given in each case, none of the three approaches outlined above can serve as a basis for process analysis. In by-passing them their application at a later stage in the analysis is not ruled out. The proposed decomposition must, however start from a search for *fundamental* constituents of the process; process steps that derive from basic intellectual and human contribution. In doing this one must accept that the resultant process model will almost certainly be an idealisation remote from current attainability. Some of the activities or steps that are identified may be superficially similar to elements of current practice. The latter, however, are diffuse, lack precise definition, discipline and interfaces. When activities are identified by means of a disciplined analysis, appropriate definition can circumvent such weaknesses.

2.3 A Primitive Decomposition

We proceed therefore with a decomposition of the model of figure 1 into more primitive actions.

The programming process is generally initiated by an apparently simple concept statement, 'A Payroll Program', 'A Set of Statistical Programs for the Social Sciences', 'An Air Traffic Control System', 'A Programming Support Environment'. Many people may be involved in further clarification of the intent and objectives at this point, end users, executives, analysts and programmers for example. Each will have a different understanding of the stated concept, a different vision of the ultimate system. Thus the first step of the ideal process must be to reach a consensus, an agreed viewpoint as to the proposed application and its objectives in its domain of operation. This understanding should be represented in a *problem statement*, an abstraction that provides a model of the intended and expected operational environment. The model must include representations of the entities, structures, relationships, events and processes that are or may be relevant to the proposed application. It will also include representations of all relevant theories that relate to the domain of interest and all procedures that have been laid down to regulate activity within the domain. These theories and procedures must be sufficient to provide a definitive framework within which the desired mechanisation can be achieved.

Given agreement on the application one may proceed to identify and define a solution described by a *solution model*. The initial representation of that solution constitutes an abstraction of the system ultimately to be constructed, installed and used, [BEN81]. Development of the appropriate model constitutes the next basic and intrinsically necessary step of the process.

Once the solution has been defined, the intended system may be designed. The design process is required to create a structure of primitive mechanisms, a *design model*. Primitive in this context means that the designer is able to specify each mechanism or object in terms of its *inputs*, *outputs* and *transfer function*, leaving it to others or to a later time to determine *how* the mechanism is to be constructed. This is, in fact, precisely the method adopted by Dijkstra in his design of each level of the THE system. That is, Dijkstra's methods cover just one step of the more complex process required when a larger system than THE system is to be

constructed; requiring more people working over a longer period to produce a long life system usually intended to operate in more than one installation or location.

Note that in terms of current terminology each object in the design model may, if suitably defined, may be viewed as an abstract data type. Such descriptions may be interpreted as an 'Application Concept' as defined above. Thus their individual design and implementation will, in turn, demand performance of the activities we are outlining. In this secondary recursive application of the process, however, the concept and the viewpoint model are represented by the output of the previous steps. If these are designed to provide a formal and therefore precise output, the need for the defining steps disappears in the lower level iterations. Moreover the system specification implicitly carried over from stage to stage, may be seen as the invariant of the iterative or recursive process (21.1). This provides a potential for verifiable integration of the total process.

Some of the productivity implications of this viewpoint will be discussed in section 4. For now we merely comment that one of the challenges of *process design* must be to achieve the, in some sense, most effective number of process iterations or of levels of process recursion.

In any event, the design model in all its detail must now guide and control implementation of the desired system through the construction of the *program model* to complete the process. Thus the process model may be further developed as illustrated by figure 2.

2.4 The Process as a Sequence of Model Transformations

The above analysis, suggests that the programming process may be viewed as a series of transformations of *representational* models. A program is a model of a model of a model of an application concept [LEH80]. Each such model constitutes a double abstraction. Looking 'upwards', it abstracts the application domain, the Real World. Looking 'downwards' it is an abstraction of the ultimate system.

Several implications follow. In particular model construction, transformation, verification and validation methods and tools are seen to constitute elements of an effective and

21.1 (*Orig*) This observation was suggested to me by J D Lehman.

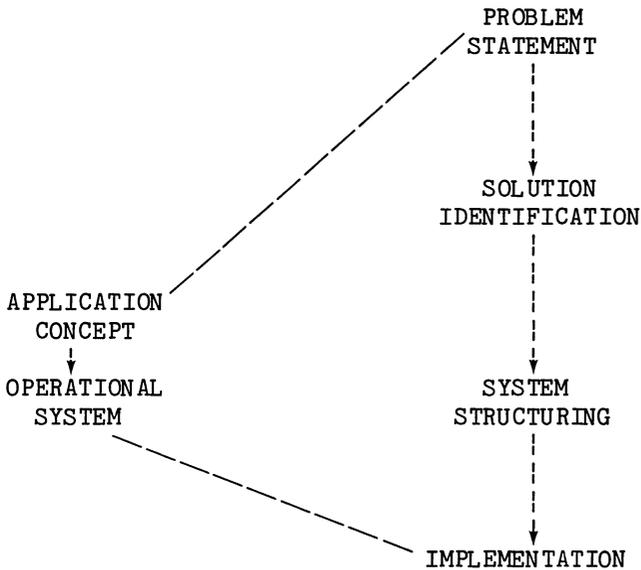


Figure 2 A Second Level of the Process Model

cost-effective programming process. They are therefore potential ingredients of a vertically integrated programming support environment; a means for achieving increased productivity. Furthermore since each model is a double abstraction, the process, and therefore the methods and tools, must be reversible so that a change to any model in the sequence, for whatever reason, can be reflected through appropriate adjustments to other models of the sequence. Maintenance of *all* the models is of course essential if the system is to be effectively and cost-effectively maintained in a changing environment over a long period. Extending the time period over which a system can be so maintained is clearly a major potential source of increased productivity.

The view of the process as a sequence of model transformations also suggests that the adoption of suitable structures and notations for the formal representation of these models can yield, at least, a semi-automatic process in which human intellect guides the process of transformation by providing the new information and decisions required at each level, with other structural and functional transformations left to verified mechanisms. The work of Darlington [DAR79] and of the TUM group [BRO80] is beginning to demonstrate what can be achieved.

2.5 A Third Level

2.5.1 Introduction

In continuing the development process decomposition, we consider a single transition of the, possibly multi-level, process, assuming a perfect process that is purely sequential from concept statement to final implementation of each level. The state of the art in problem solving and in programming certainly does not permit this. Moreover it has recently been shown, that P- and E-type programs [LEH80], by their very nature, when finally operational can never be entirely satisfactory. Such programs will always undergo evolution, since their development must be based on a prediction of the operational environment as it *will* be after installation. Iteration is therefore *inevitable* over the entire process and, at least for moment, is also highly likely over the internal steps. Nevertheless, even at this time, identification of the ideal process must yield insight and understanding that may then be applied to modify and improve the current process and to support the further development of programming methodology and technology. In the long run one would hope to be able to implement a process approaching the ideal, and to support it with the appropriate tools.

2.5.2 The Viewpoint Model

Referring back to the process model of figure 2, problem identification and statement really consists of two separate parts. A viewpoint common to users, developers and the appropriate managers must be agreed and reflected in a *viewpoint model*. The extent to which this *representational model* can be formalised will determine the degree to which its subsequent transformation into a more refined model can be mechanised and formally verified. Additional facilities for validation of this model using, for example, an *executable metric model* [LEH81], [ZAV81], (which ideally would be the viewpoint model itself) would be a powerful additional tool to ensure optimum decisions at this stage.

2.5.3 The Requirements Model

Following the development of an authoritative and documented viewpoint of the application in its operational environment and on the basis of this first model, a complete set of application objectives, that is system requirements, must be developed and agreed. Even if the viewpoint model was informally described, the *requirements model* should certainly

be so structured and formalised that, at the very least, its self consistency may be verified and the development process more easily continued. This second model may be regarded as being concrete in real world application terms, abstract in terms of the object system. Its purpose is to express *what* the system and its mechanisms are intended to do; not *how* they are to be realised or constructed.

2.5.4 The Specification Model

Requirements address and state real world needs. That model therefore completes the 'problem statement' but does not need to reflect those system properties required for problem solution. A *specification model* expresses the identified needs in terms of system components, or primitives that are known to be, or can reasonably be expected to be, ultimately realizable in physical mechanisms. The specification model is, in fact, conjugate to the requirements model, in that it is concrete in systems terms, abstract in terms of the application domain, the Real World.

2.5.5 The Phenomenological Model

Design of the specification model represents the transition from problem statement to solution identification. Together the requirements and the specification record the objectives that have been adopted and the system characteristics that are proposed to achieve them. These must now be recombined with the theories and laws that are part of the viewpoint model and have thereby been accepted as adequate to describe and govern phenomena in the real world, and therefore the behaviour of the solution model. Thus the next step is the transformation and re-combination that leads to the appropriate *phenomenological model*, that includes in the developing system description, the constraints arising from the attributes of the real world system that it abstracts and the operational objectives that have been adopted.

2.5.6 The Computational Model

The phenomenological model provides the necessary and sufficient constraints to define a mechanism that supports the desired application. In the general case it will not, however, define mechanisms and procedures implementable by computers and other devices. For example, if the application involves moving bodies, the model may include differential equations. If the application involves humans, the model may include rule books, collections of laws or procedure

descriptions. All these must now be cast into a form which can be mechanised in the intended technology. Differential equations for example may be replaced by difference equations, rule books by decision tables or finite state machines [HAR81].

Furthermore, the structured or unstructured description provided by the phenomenological model, will include only such time and sequencing constraints as are implied by the ordering of events in the real world. Additional constraints must arise in the computational process, for example because of information and resource sharing, because of the sequential nature of devices and because of limitations on the numbers of concurrently operating devices. The phenomenological model must therefore be further transformed into a *computational model* to achieve a design that is mechanisable using digital computer technology.

2.5.7 System Design and the Structure Model

Once the computational model has been developed, it must be transformed and re-structured to provide a partitioning into available or implementable primitives. This is the process generally called *system design*. However, we prefer to term it the *system structuring process* since, as stated above, 'design', the process of applying intellectual power to add the fresh ingredients to a model, is required in the development of each abstract model. However, only at the step now reached is it precisely the structure properties that represent that new ingredient. The preceding development has repeatedly stressed the need for structuring the models identified. One of the functions of structure is to make the model it constitutes *understandable*, not only to the designer but to all who subsequently need to work with it. At the step of the total process now reached, system structure must be sufficiently developed to make the system implementable, learnable, usable and maintainable; characteristics that demand that the mode of operation of the system and its constituent parts be comprehended to the level of detail required to complete relevant tasks effectively.

The *structure model*, developed to the appropriate level of detail identifies the primitive mechanisms from which the system is to be constructed, and the intercommunication between them. Each such primitive may be regarded as a unit for which an input set, an output set and a transfer function or unit semantic is defined. Current concepts such as abstract data types, Modula 2 modules [WIR78] and Ada pack-

ages [ADA80] are special instances of the primitive concepts presented. However, on the basis of the development process described, the primitive units which we envisage would be completely specified, semantically as well as syntactically.

2.5.8 The Program Model

Following the development of the fully specified structure model, one reaches the final transformation of the system process, the *implementation*; in the case of a software system, the *program model*. The total transformation process as now visualised is illustrated by figure 3, which also indicates how this process model related to the more elemental models of figure 2 and 1.

PROCESS MODELS

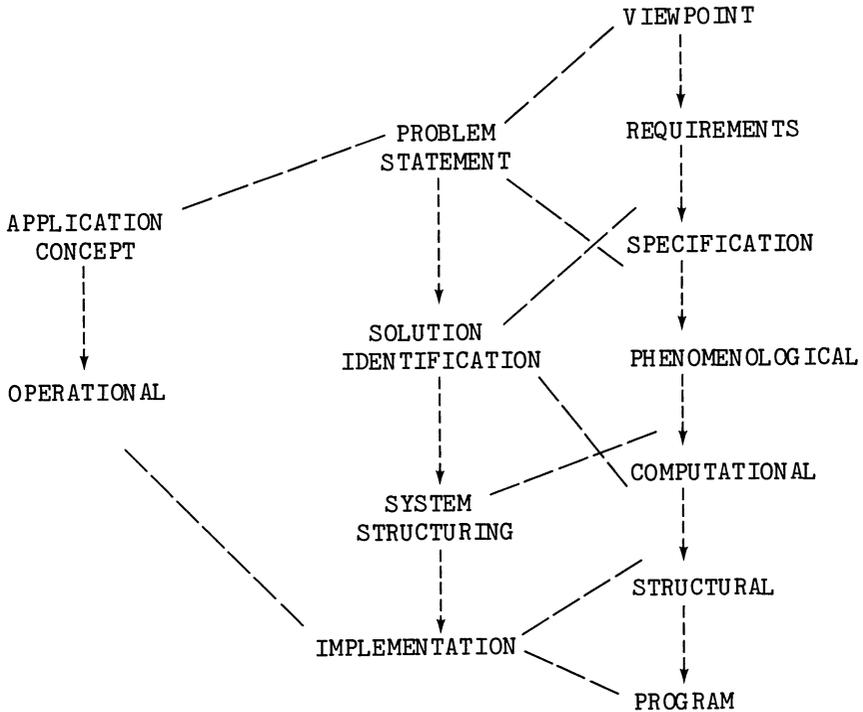


Figure 3 The Basic Steps of the Programming Process

3 Step Structure

3.1 The Step Paradigm

The previous section has outlined an idealised seven step program development structure. Each of its steps encompasses a fundamental human intellectual contribution that is orthogonal to those contained in the other steps though the extent of potential mechanisation increases as one proceeds down the process. It is now suggested that there exists a common structure for the steps, as illustrated by figure 4.

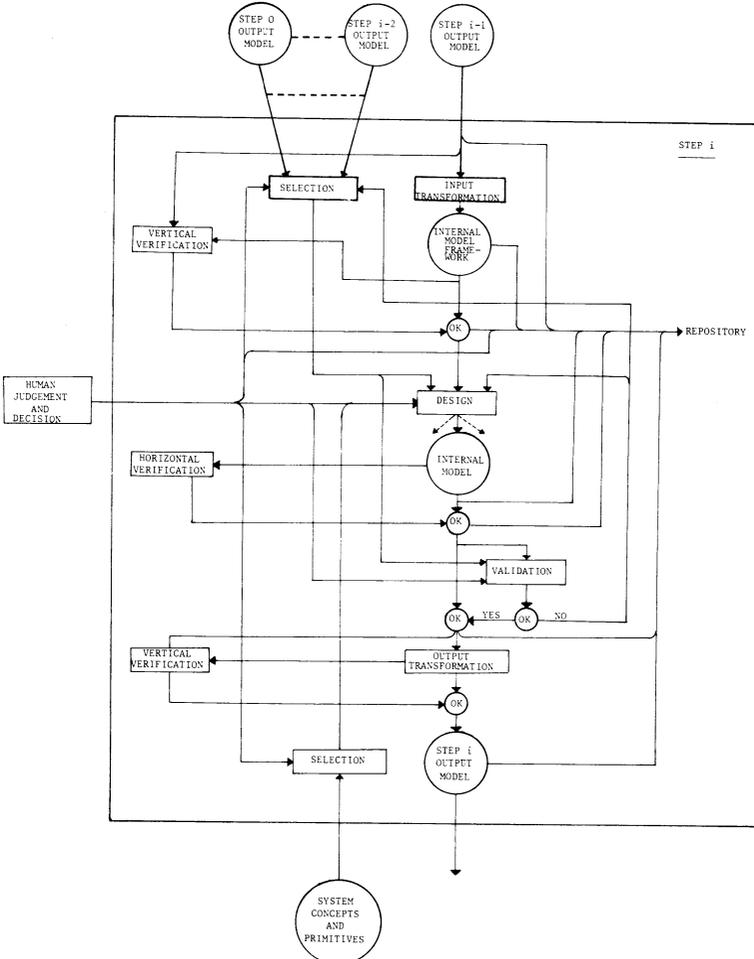


Figure 4 The Ideal Step Structure

It is the totality of the transformation described by the above figure, the design, the mechanical part, and the accompanying verification and validation, that comprises the model transformational-step paradigm. Space does not permit a detailed discussion of this model or its relationship to the individual steps and representational models. Brief comments on structure, notation, verification and validation are however desirable. It is convenient to discuss these with reference to the requirements and specification steps but the remarks are relevant to all aspects of the total process,

3.2 Structure and Notation

The critical properties of any model are its structure and the notation in which its content is described. A *specification model*, for example, should serve as a mechanism for users, designers and implementors, to provide answers to questions about the intent, the completed system or the current state of its design. For any question asked about the system, the model must either lead the enquirer to the answer or it must demonstrate that no answer exists; that is that the feature in question has not been defined. In the latter case, the mechanism must also indicate *where* any definition now formulated is to be recorded so that the next enquirer asking the same question, or rather asking *any* question requiring the newly formulated attribute as, at least, part of its answer, will indeed be guided to that answer.

This view of specification suggests that the most fundamental property of a specification is its *structure*. *Notation* that ensures an entry, *when located*, is unambiguous, complete and formally manipulable is important but secondary to structure. As already observed, this comment applies more generally.

3.3 Vertical Verification

Each step is initiated by verifiable mechanistic translation. This transforms the structure and notation of the input model from a form suitable for conveying understanding about the previous step of the process into a *framework* in which, for the current step, the design decisions and model refinements may be made and precisely, retrievably and understandably expressed. If the source and object models of this transformation are both formally described and if a verified mechanical transformation is used, then the precise correspondence between the representations may be guaranteed. Alternatively

the results of the transformation can be verified. In either case this type of verification is termed *vertical verification* of the transformation process.

3.4 Design, Horizontal Verification and Validation

Once the requirement framework has been created through transformation from the viewpoint model, the requirements may be *designed* and their structured, formal or informal, description embedded in the framework. This produces the requirements model. Use of the word 'design' here is quite deliberate. The construction of each of the models discussed in the current analysis involves design; selection between alternative model elements - in this instance requirements - and their structuring into a complete and cohesive system of minimal complexity that reflects a final system that could be operationally satisfactory and cost-effective. It is this design activity that demands human creative intellect to identify needs and opportunities; benefits and costs; to recognise relationships, dependencies and commonalities; in general, to exercise judgement, choice and discrimination.

The design portion of the transformation process adds and 'colour' and detail and calls for a second form of verification, *horizontal verification*. This must demonstrate that the set of decisions that have been taken are consistent and, in some sense, complete; covering at the very least all the requirements implied by the source model.

Any model constructed by the application of choice and decision, is one of many that could have been selected at that level during the associated step. It is therefore appropriate to judge the effectiveness of the decisions taken in a *validation* process using, for example, the executable metric models referred to above. One would hope to determine whether, in terms of one's knowledge and expectation of the remainder of the total process and of the implementation technology, the successive design activities at each step can reasonably be expected to ultimately lead to a satisfactory operational system. Moreover for operational characteristics that have been left unspecified, validation should provide an estimate of what they are likely to be, or at least that they are in an acceptable range. The further one proceeds in the total process, the more precise these forecasts should and can be.

3.5 The Common Step Structure

This common step structure, like the process structure of the previous section, cannot be achieved with present methodology and technology. It is however, considered attainable over the next decade (21.2). If achieved and duly supported with a vertically integrated programming support environment, that is a complete set of homogeneous methodologies and tools, it should result in a process significantly more effective and cost-effective than exists at present. In addition it will be possible to improve the, essentially linear, process by stepwise refinement, both over sets of steps and within steps to achieve incremental productivity improvements.

4 Productivity Implications and the Current Process

The absence of significant productivity increases in programming over the last twenty years is a direct consequence of the history and resultant make-up of the total process that is followed today. On the other hand it appears that a total replacement process, even one only remotely approaching the ideal, is probably a decade away. Thus one must still seek to increase productivity in today's environment, accepting any gains as a palliative until a more fundamental breakthrough can be made.

For a true study of productivity issues the total life-cycle process must be considered. For both the current and conceptually ideal process this may be represented by a four tiered structure. A major cycle encompasses the entire level and step process to provide the framework and mechanism for system adaptation to a changing environment; that is system evolution. The potential for increased productivity during evolutionary maintenance arises from the availability of a structured linear process that can consistently yield and maintain a well structured, comprehensible system with formally documented function and properties. The comprehensive historical archive (information repository) whose existence is indicated in figure 4 must maintain the information required to make effective changes to the system and to make them cost-effectively. Complete and, at all times up-to-date, information is essential to achieve a long-life system that can be responsively and economically maintained

21.2 (Eds) *An IPSE conceptually based on the concepts and on insights developed since then [LEH84] is currently under development at Imperial Software Technology Limited (IST).*

as the application environment and available technology evolve. Where the means for its provision exist, the consequent increase in the time over which the investment in the software may be depreciated must represent a significant increase in effective productivity. Moreover, the additional maintenance investment during the extended life period will be minimised.

The step structure illustrated by figure 4 constitutes the other extreme of the process structure visualised. It has been widely recognised that adoption of formal representation, where applicable, is a clear productivity aid. It must facilitate human understanding, avoid imprecision and ambiguity, forestall errors due to misunderstanding. It also opens up the potential for automatic transformation of the model structure and content from step to step; to recast the model into a form suitable for analysis, design and the addition of the new detail relevant to each step.

Furthermore, formal representation facilitates verification. More general, the detection and correction of errors, weaknesses or deviations from optimum, become progressively and rapidly more difficult and costly as one proceeds through the process. Thus the maximum degree of verification and validation at each step, leading to early discovery of faults, must make a major contribution to productivity growth both in their own right and because their application must simplify the subsequent development of a system as a consequence of the reduced error content. Many issues, methodological, structural and notational must be resolved if the maximum benefit is to be reaped from the concepts outlined; particularly if full advantage is to be taken of the fact that a structurally similar process (21.3) is executed within each step. This is true for the ideal process and even more so for the current process. Neither investigation can be pursued here.

Finally, productivity growth potential at the middle tiers of the total process must be examined. At the higher of the two, the process of stepwise refinement develops a sequence of levels of description in terms of primitives, progressively more abstract with reference to the real application in its environment; progressively closer to implementable and executable system primitives. The main

21.3 (Eds) *As demonstrated by the LST canonical step*
[LEH84].

productivity issues in this process relate to selection of the optimum number of levels and the optimum set of primitives in terms of which the levels are described or constructed. The step procedure described in 2.5 in relation to progress through a single level, is highly relevant to a closer study of this optimisation problem.

The second of the two middle tiers is the step process between neighbouring levels as described in section 2.5. The process structure provides a framework within which the potential of various techniques for improving productivity may be examined for the inter-level activities and intra-level sub-activities.

For example consider the use, as first proposed by McIlroy [MCI68], of standard software packages. Evaluation of this concept requires one to identify which step of which level, within the total process it relates to and what technologies must be mastered if it is to be at all effective. In doing this one may, of course, approach the evaluation from the viewpoint of the current process or one may base it on 'ideal' models such as those developed above. Since the steps identified are considered, in some sense, fundamental and orthogonal to one another, they provide a suitable framework for a productivity analysis; even with reference to a process not precisely based on those steps, since the consequences of a change in the context of a step, may be evaluated independently of other steps.

The existence of various approaches to productivity growth has been widely recognised. The recent IEEE Software Engineering Productivity Workshop, [SEP81], for example identified a variety of such techniques. These and others like them cannot be examined more closely in the present paper. But we re-affirm our conviction that the opportunities they offer must be examined *in relation to the process within which they are applied*. That analysis must be applied over all steps of what we have termed the four tier process, or its equivalent in current methodology. This may be done by setting up a five dimensional array, or a lower dimensional sub-array. The potential scope of each technique in relation to the fundamental activities of the process may then be determined and their potential contribution to productivity growth in the relevant activity, estimated. An outline two dimensional array that served to guide the discussion at the above referenced workshop is given in figure 5.

PHASE	1	2	3	4	5	6	7
ASPECT							
SIMPLE SOLUTIONS	<u>F</u>	<u>F</u>		F	<u>F</u>		
RE-USE	<u>F</u>		<u>F</u>		<u>F</u>		
EFFORT REDUCTION		<u>T</u> ← →			→ <u>T</u> →	→ <u>T</u>	
FAILURE AVOIDANCE	Va	<u>Va</u>	<u>Ve</u>	Va	Ve	Va	Ve
USER PARTICIPATION	<u>F</u>	<u>F</u> Va		F Va			
ITERATION REDUCTION		F	<u>F</u>			Va	Ve
COMPLEXITY REDUCTION	<u>F</u>	F		F →	→ <u>T</u>	F →	→ <u>T</u>
LOCALISATION OF CHANGE	F	F			F	<u>F</u>	

POTENTIALLY MOST REWARDING PHASES UNDERLINED

F=formulation
 T=transformation
 Ve=verification
 Va=validation

1=Viewpoint
 2=Requirements
 3=Specification
 4=Phenomonological
 5=Computational
 6=Structural
 7=Program

Figure 5 Process Based Productivity Approach Evaluation

5 Conclusion

The present paper has not made a direct contribution to the exploration of productivity improvements in the programming process. It was not intended to do so. Instead attention has been drawn to the underlying reason for the lack of significant growth in programming productivity in recent years. Only by linearising the total process to the point where local actions can be implemented without impact on other parts of a system or on a later stage of the design, implementation and maintenance process, can one expect to make real progress. The correct approach to achieve this is to consider the total process and to analyse it top-down in order to obtain a purer process structure within which appropriate methods can be devised, formalised and supported. The time is now ripe for such exploration and development. Moreover this is the real implication of the now widely accepted, but not fully understood, concept of the vertically integrated programming support environment. The next decade should see such systems emerge and take their place operationally, to achieve ever more productive computer usage through the availability of software that is correct and can be maintained correct; software that continually provides effective and cost-effective function in an ever evolving environment.

6 Acknowledgements

I gratefully acknowledge the many creative discussions I have had with Professor W M Turski, with my close associates Drs G Benyon-Tinker and P G Harrison, with Mr G M Sokol of ERO and over many years with L A Belady. All have helped in no small measure to develop the understanding and the concepts that this paper has tried to present. Thanks are also due to ERO for the support so generously given under contract number DAJA-37-80-00011. Finally, my grateful thanks to Miss S Warren for her uncomplaining willingness to cope with the stepwise refinement process needed to produce this paper.

CHAPTER 22

THE ROLE OF SYSTEMS AND SOFTWARE TECHNOLOGY IN THE FIFTH GENERATION*

1 Introduction

This paper will not discuss the virtues or potential of the Japanese Fifth Generation Plan (FGCS) [JIP81] or of the 'primitive' concepts and sub-systems that are to be used in the implementation of the systems proposed. The potential of knowledge-based expert systems, functional languages, abstract data-types, logic programming, data-flow architecture, distributed systems and VLSI is clear. The increasingly widespread use of these terms may give them the appearance of buzzwords. In fact they represent important and potentially valuable concepts and technologies that will have significant impact on future generations of computer systems; and therefore on a society increasingly reliant on, even dominated by, such systems.

Nor will the paper consider the realisability of the plan in terms of the maturity of the technologies proposed, the technological challenges arising from changes of scale of several orders of magnitude or the time scale envisaged for system implementation. Instead the paper will draw attention to some issues and problems which must be considered in parallel with pursuit of the objectives set out in the FGCS plan, if the 'Dawn of the Second Computer Age' is not to lead to 'Sunset for Civilisation'. The content may seem to sound a discordant note, even alarmist. That is not the intention. There is every reason to believe that the concepts presented by the Japanese plans will only be realisable when solutions have been obtained to the problems to be outlined. Moreover, if solutions are not found and plans such as this are pursued remorselessly, it will lead, at the very least, to a repetition and intensification of the 'Software Crisis'. At worst, it could develop into a serious threat to civilisation. Thus a search for such solutions becomes a matter of the highest priority.

Reprinted with kind permission from 'The Fifth Generation: Dawn of the Second Computer Age', proceedings of an international conference, July 1982, SPL Insight, Abingdon, UK.

2 VLSI

It is of interest to first note that the analysis to follow is as relevant to VLSI 'hardware' as it is to software. As element numbers per chip increase, the latter will develop the complexity, invisibility, evolution and uncertainty characteristics of software on top of the inherent problems of the technology itself. As the manufacturing process becomes automated, chip functionality will be defined by that process and its formal inputs. Often it may even be a design option whether a chip functional-specification is used to control a manufacturing process or as source code for a program subsequently to be stored in and executed by a simpler chip. Thus in the remainder of this paper, 'program' is to be interpreted as including both soft and VLSI implementations. In fact, the use of VLSI makes many of the problems to be discussed more critical. In connection with the issue of correctness, for example, an error, once etched, can, in general, not be corrected without repeating the entire manufacturing process.

3 The Problem Areas

3.1 Identifiers

The problems which may be receiving insufficient emphasis in the FGCS community can be indicated by words such as *Requirements*, *Evolution*, *Complexity*, *Understanding*, *Correctness*, *Responsiveness*, *Cost*. These point to problems that have plagued the software community for nearly two decades. Yet, to the author's best knowledge, published material on the FGCS plan makes no significant reference to the implied problem areas, to their likely impact during development and after installation of FGCS systems or to their implications on R & D in the concepts, methodology and software tools areas. The remainder of this paper briefly examines their significance and implications.

3.2 Requirements

The analysis of a proposed new computer application or of a change to an existing one is one of the most critical activities in the computing system life cycle. It is required so as to determine and specify the attributes of the system to be implemented and the criteria by which it is to be judged. Errors or omissions at this stage are likely to prove most disturbing to the end user and are certainly the

most expensive to rectify. Methods and tools to support the requirements definition process and to make it more reliable, are under widespread development. But at the present time it is still more of an art than an engineering discipline.

There is no reason to believe that the problem of determining and fixing requirements will be simpler for the new generation of systems, even if a more precise and concise notation is available for their representation. The problem may well prove more difficult and critical because of increasingly ambitious and complex applications, larger systems and the ever-growing interweaving of automated systems with the very fabric of society. The consequences of ignoring it will inevitably result in repetition and compounding of the earlier experience with von Neumann software. The plan appears to neglect the problem and its implications.

3.3 Evolution

The evolutionary nature of computer applications and, therefore computing systems, is now generally accepted; recognised as intrinsic and not merely a weakness of the development process [LEH80,82]. The implications of this on the process, on system architecture and on the relative strengths and weaknesses of different implementation primitives has not been widely considered. But past experience in the software world indicates the likely consequences of a research and development plan that ignores these issues. It is one thing to design and implement an initial system. It is quite another matter to eliminate weaknesses, to provide improved or new facilities, to adapt an operational system to new circumstances and external change.

In the present state of the art, even minor changes to the clauses of a Prolog program or to the structure or content of a data base demand, for example, consistency checks that may cause serious problems of verification. It may even require a complete re-creation of the system. The published work to date on examining and controlling the consequences of continuing change in an operational system based on these technologies is insufficient. The problems must be successfully addressed before the proposed technologies may safely play a major role.

3.4 Complexity

A likely consequence of evolutionary pressure on a system not designed with evolution in mind, is a rapid increase in system complexity, or a need for increasing effort to control it. This will be accompanied by corresponding decreases in reliability and in the ability to respond in timely fashion to external (user) requirements for change. Either way there will be a rapid and continuing increase in maintenance costs.

There has been no indication that a strategy for complexity control has been seriously considered in the FGCS plan, or even that there is an awareness of the problems and the pitfalls. For a project of its size and imaginative extent, this is a matter for serious concern.

3.5 Understanding

There are two separate issues to be addressed under this heading. The first relates to human understanding of the objectives and function of a computer application and the system that implements it. If it is accepted that mankind must remain in control of its own destiny, we may only construct systems that, in the fullest sense of the word, are *understood*. If, subsequently, they are to continue to be adapted, reliably, responsivly and cost-effectively, to more and more ambitious application concepts in changing operational environments, total human understanding of the systems is even more vital.

As computerisation increasingly penetrates every facet of human activity, total understanding and control of the system becomes essential if the visions of science fiction are not to become reality. Thus systems development must be paralleled by the development of technologies and structures that ensure continued human comprehension of what the system is, what it does and how it does it. The FGCS system is conceived in terms that require the provision of support methods and tools not currently available, but essential if the system is to be and to remain under control. Once again the plan shows no awareness of the problem; makes no provision for its investigation and solution.

The second area of concern under the heading of 'Understanding' concerns basic assumptions that underlie the technologies of artificial intelligence and expert, knowledge based, systems. Human living and progress relies on the application of knowledge *and* understanding, common sense. We

all apply these to a greater or lesser extent. The expert is one whose viewpoint, interpretation or prediction is perceived as correct, say, 51% of the time. They also make mistakes. But the events that slowly unfold after judgement has been made or decision given, usually permit timely feedback-triggered corrective action based on developing understanding of consequence, or on new information. It is foolish to act on the assumption that the expert is always, or even almost certainly correct. May one expect expert systems technology to develop at a rate to cope adequately with uncertainty?

Current technology in knowledge based systems is just that. The system is limited in its discerning powers by the information that humans have stored in, or caused to be captured by, the system before the moment of decision. It seems most unlikely that the state of the art in AI will advance at such a rate that *knowledge* based systems can display *understanding* on the timescale envisaged by the FGCS plan. And when they can be designed to do so initially, can that capability be reliably maintained under continuing change? Can they be expected to be capable of creative, understanding based, concept development? If they cannot, what is the danger?

The doctor seeking support from an expert diagnostic system, the military commander seeking interpreted information in a defence application, can and will apply discrimination. Viewpoints and judgements will be developed but then modified by human experience and intuitive judgement. Creative insight will be applied to refine and evolve the application and its reliability. The output of the systems visualised in the FGCS plan, based at present almost exclusively on selective information retrieval, will in most instances be incompletely understood but inadequately filtered because of the faith in computers of the uninformed, the obscurity of the derivation process and the volume and speed of that output (22.1). Putting such systems into the public domain prematurely is fraught with danger.

22.1 (Eds) *The intention of IKBS implementors to accompany the inferences that are presented as output, by the chain of reasoning that was used to reach them, is irrelevant in any real and major application, where the chain is likely to be long, complex and cannot normally be 'understood completely' in a real-time context particularly if such inferences are generated at a high rate. Moreover the human will have his attention focussed on what is presented and will be unlikely to notice what is missing in the chain of reasoning.*

3.6 Correctness

The published plan makes passing references to 'simple verification' of programs (eg p 62, 78, 83) and to 'partial verification' of knowledge based systems at 'the meta-knowledge level' (p63). Such limited references to issues of correctness under the generic title 'Systemisation Techniques', suggests that insufficient consideration has been given by the planners to ensuring that systems to be implemented are correctly defined or implemented or, indeed, what correctness in the FGCS context means. There is no expression of either the importance of these issues or how far away and costly their solution is. The fundamental requirement that a computing system must *be demonstrably correct in the first place and stay correct throughout the life-time of the system* will require much more emphasis than it has so far received. The development of methodologies and tools to satisfy it becomes a priority.

Specifically, references to concepts of verification and validation in relation to the technologies to be exploited, are conspicuous by their absence from the publicised FGCS R&D plan. Nor can it be relying on work already done in the technologies to be exploited. The problems of specification, design and verification in the context of the ten million transistor chip envisaged in the plan are, to say the least, non-trivial. Published work in Expert Systems does not appear to have given much attention to such matters. The position in logic programming is a little better [CLA81] but much remains to be done.

It appears that the logical basis of all these systems is assumed sufficient to ensure the absence of error. Correspondence between a set of clauses in a Prolog program and the real world model that they represent, may be self-evident. But what is 'obvious' in a ten or even hundred clause context, presents a major problem when the precise total semantics of a thousand clause system must be determined, *and demonstrated*. The correctness of a thousand item 'knowledge base' may perhaps be relied upon. Increase that size to that implied by the systems envisaged and no human mind can comprehend what it contains, even at inception. Even with chips of sizes currently manufactured, questions arise about their 'correctness' or *precise* functionality.

To meet operational requirements the application dependant functional content of software implemented with non von Neuman components or techniques will not be less than that of

traditional software; nor will the pressure for evolution be any less. Individual sub-systems implied under the FGCS plan will have to exceed the size of present day implementations in the same technologies, possibly by orders of magnitude. When one considers the consequences of size, of evolution and of the related growth in complexity the magnitude of the problem of maintaining veracity and control becomes self evident. Appropriate process technology does not yet exist.

It may be argued that given 'correct' transformation procedures, the issue of correctness is restricted to specifications. But that is merely shifting the problem one level. Specifications for systems in actually implemented examples of the FGCS primitive technologies, are small in relation to what will be required in FGCS systems. With available techniques, the work involved in demonstrating that the latter satisfy the need for which they are intended, will inevitably prove obscure and difficult, even if expressed in first order logic, say. Even quite short published proofs of program correctness in that logic have contained errors. The fact is that for a specification or system description that exceeds the intellectual grasp of a single individual, it is the total *structure* rather than the *form* or language of its content that is the more critical in the context of understandability and correctness. The structuring of Prolog and Expert systems and of their specifications requires urgent investigation.

3.7 Responsiveness

The spreading application of computers makes it ever more vital to keep the system in sympathy with changing and evolving needs. Adaptation of the system to ensure appropriate behaviour should be implementable on a time scale determined by user needs. Limitations imposed by technological capability can cause major problems. Such limitations exist with present systems. Responsiveness of the software *process*, will become an increasingly critical parameter in the next generation. Unless *process technology* for the FGCS primitives can be developed and applied, serious consequences must follow.

3.8 Cost

The problems of the high cost of software have been recognised for almost a decade [GOL73]. Expenditure on software development and maintenance in the USA in 1977 exceeded 3% of the GNP. That fraction is increasing every

year. A recent DoD report [RED81] quotes an Electronic Industries Association forecast that DoD expenditure on embedded software alone will grow from \$B2.82 in 1980 to \$B5.62 in 1982 to \$B32.10 in 1990, 'if we do nothing'. Organisations making significant use of computing already dedicate 10-15% of annual expenditure to that activity, an increasing fraction of it being for software.

The greatest part of all such expenditure is incurred in coping with problems of incorrectness, adaptation and evolution. As already discussed, there is every reason to question the availability of adequate technologies to cope with these same problems as they will appear in FGCS. 'Something' *must* be done in all areas of Software Technology and Engineering, to permit development of the proposed systems and their continued evolution. Equally, the economic implications of ever increasing expenditures that *must* be met because of societal dependence on computer systems demands urgent attention to the problems outlined above and to others not discussed here because of space limitations.

4 Conclusions

The above comments do not imply that the long-term system concepts of the FGCS plan are impractical or undesirable, merely that Research and Development activity in broad areas of Software Process Technology must be intensified and extended. In practice, it means that the further development of the technology and its tools, and their extension as required by the new approaches to system architecture and implementation, become an urgent priority. In view of the under-developed state of process technology one must, however, also question the FGCS timetable on the grounds of both practicability and desirability.

Software Engineering includes the management of complexity and change. Its objectives include meeting the needs of users effectively and cost-effectively. It must also continue to support reliable and responsive satisfaction of those needs as they change and evolve and as advancing insight and technology opens up new application potential and new opportunities. With ever growing use of computers, the welfare, even survival, of society will increasingly depend on correct and appropriate computer outputs. For this to be achieved and maintained demands the availability of an effective engineering discipline with its insights, models, methods, procedures, tools and rules.

Past experience has shown that a less than adequate technology will prove time consuming, costly and painful. The present focus provided by the plan does not appear to have considered these factors. Total re-examination of the process whereby an application concept is transformed into an operational system and then maintained in tune with its operational environment is urgently needed. The process as it exists today has evolved in bottom-up fashion over two decades in response to local needs and advancing technology. As such it must *now* be reviewed and designed as an integrated continuous process, taking advantage of primitive concepts that already exist or that can be developed. Methods and tools to support *all* aspects of the total process and to make it reliable and responsive to societal needs, must be explored, developed where necessary, and applied as appropriate in a *software technology process* that is able to justify the faith that society will put into its products. If we fail to do this mankind will be gradually, insidiously, undermined by an accumulation of incorrect or incomplete information that could even lead to the destruction of Society as we know it. The dangers stemming from an all pervasive but incomplete Information Technology are surely no less than those posed by genetic engineering or biological warfare. That suggests a risk we dare not take, a responsibility that we must shoulder.

5 Acknowledgement

I am most grateful to Dr G Benyon Tinker, Mr R J Cunningham, Dr J Darlington, Miss V A Downes, Dr P G Harrison, Dr R A Kowalski, Mr G M Sokol and Dr I Torsun for their constructive criticisms of this paper. The opinions expressed are, of course, my own.

POSTSCRIPT

For purely pragmatic reasons selection of material for inclusion in this book had to be completed in early 1982. Much has happened since then. The reader who wishes to achieve an overview of more recent developments is referred to the proceedings of two recent meetings. The first Software Process Workshop was held in Egham, Surrey in February 1984, sponsored by ACM, BCS, ERO, IEE and IEEE and its proceedings published by the IEEE under Catalogue number 84ch2044-6. The second, with equally wide support, was held at Coto de Caza, California in March 1985. Its proceedings are to be published later this year. Their contents, with references provided by the various authors and discussants, presents an up-to-date picture.

It is worthy of note that a third Process workshop is planned for November 1986, that the title for ICSE 8 (to be) held in London in August 1985 is 'Improving the ... Quality of Software ... by improving the Software Engineering Process', and that of ICSE 9 (Monterey, 31 March - 2 April, 1987) is 'Formalising and Automating the Software Process'. The central role of process directed thinking and the related phenomenon of program evolution are clearly becoming more widely accepted.

LAB
MML

24 May 1985

REFERENCES

- [ADA80] 'Reference Manual for the Ada Programming Language', Proposed Standard, US DoD, Jul 1980, Section 7, pp 7.1-7.9. (Ch 21)
- [AHO75] A V Aho, J E Hopcroft and J D Ullmann, 'The Design and Analysis of Computer Algorithms', Addison Wesley, 1975, 470p. (Ch 15)
- [ALE74] M J Alexander, 'Information System Analysis' SRA, 1974. Ten definitions are quoted on P 4, but, being slanted to information systems, do not represent an exhaustive listing. (Ch 11)
- [ALF78] M W Alford, 'Software Requirements Engineering Methodology (SREM) at the Age of Two', Proc COMPSAC 78, IEEE Cat No 78CH 1338-3 Nov 1978, pp 1-14. (Ch 19)
- [ALL77] J J Allen (Ed), 'CAD Systems', North Holland, 1977, pp 361-364. (Ch 15)
- [AMA69] American Management Association Report, 'Computer Based Management, - 1969'. (Ch 3)
- [ARO74] J D Aron, 'The Program Development Process, Part 1: the Individual Programmer', Addison Wesley Reading Mass, 1974, 264p. (Ch 12, 14)
- [ARO80] J D Aron, 'The Program Development Process, Part II The Programming Team', Addison-Wesley 1981. Reading, MA, 690p. (Ch 19)
- [BAC60] J W Bacus et al, 'Report on the Algorithmic Language ALGOL 60', Comm ACM, 1960, pp 299-314. (Ch 14)
- [BAK72] F T Baker, 'Chief Programmer Team Management of Programming' IBM Systems Journal, Vol 11, No 1, 1972, pp 56-73 (Ch 12)
- [BAK75] F T Baker, 'Structured Programing in a Production Programming Environment', Proc Int Conf Reliable Software, Los Angeles, CA, Apr 1975, pp 172-83, IEEE Cat No 75CH 0904-7CSR. (Ch 12)

- [BAR64] E F Bardain, 'Research Studies of Programmers and Programming', Unpublished studies, New York, 1964. Quoted by D B Mayer and A W Stalnader. 'Selection and Evaluation of Computer Personnel', Proc 1968 ACM Nat Conf, pp 657-670. (Ch 3)
- [BAS68] H B Baskin and S P Morse, 'A Multilevel Modelling Structure for Interactive Graphic Design', IBM Systems Journal, Vol 7, No 3/4, 1968, pp 218-228. (Ch 8)
- [BAS77] V R Basili and R W Reiter Jr, 'Selecting Automated Measures for Software Development', U Maryland, 1977. (Ch 15)
- [BAS77a] V R Basili et al, 'The Software Engineering Laboratory', U Maryland, TR-535, SEL-1, 1977. (Ch 15)
- [BAS78] V R Basili, E Ely and D Young (Eds), 'Second Software Life-Cycle Management Workshop, 21-22 Aug 1978', Atlanta, GA, IEEE Pub No 78CH1390-4C, Dec 1978, 220p. (Ch 19)
- [BAU67] W J Baumol, 'Macro-Economics of Unbalanced Growth: The Anatomy of Urban Cities', Am Econ Rev, Jun 1967, pp 415-426. (Ch 5, 6, 7, 19)
- [BAU77] F L Bauer, H Partsch, P Pebber and H Wessner, 'Notes on the Project-CIP: An Outline of a Transformation System', TUM-INFO-7729, Tech U Munich, 1977, 67p. (Ch 19)
- [BEI75] H Beilner and L A Belady, 'Speculations on Program Complexity', Unpublished Paper, IBM Research 1975. (Ch 15)
- [BEL67] C S Bell et al, 'Macroeconomics of unbalanced growth: comment', Am Econ Rev, Vol 58, No 4, Sept 1968, pp 877-897. (Ch 5)
- [BEL71] L A Belady and M M Lehman, 'Programming System Dynamics or the Metadynamics of Systems in Maintenance and Growth', IBM Res Rep T J Watson Res Centre, Yorktown Heights, NY 10598, RC 3546, p30, Sept 1971. (Ch 1, 2, 6, 7, 8, 9, 11, 12, 14, 17, 18, 19)
*Ch.5

- [BEL72] L A Belady and M M Lehman, 'An Introduction to Growth Dynamics', From 'Statistical Computer Performance Evaluation', W Freiburger (Ed), Academic Press, 1972, pp 503-11. (Ch 1, 7, 8, 9, 16)
*Ch.6
- [BEL72b] L A Belady and M M Lehman, 'A Systems Viewpoint of Programming Projects', Imperial College, Department of Computing and Control Research Report 72/31, 1972. Also Published in 'Advances in Cybernetics and Systems, I', Gordon and Breach, London 1975, pp 15-28. (Ch 8, 9)
- [BEL74] L A Belady and M M Lehman, 'Programming Systems Growth Dynamics', Published in Computer Reliability, Infotech State of Art Lecture No 20, 1974, pp 391-412. (Ch 9, 12)
- [BEL75] L A Belady and M M Lehman, 'The Evolution Dynamics of Large Programs', IBM Res Rep RC5615, T J Watson Res Centre, Yorktown Heights, NY 13598, Sept 9 1975, 45p. (Ch 1)
- [BEL76] L A Belady and M M Lehman, 'A Model of Large Program Development', IBM Sys J, Vol 15, No 3, 1976, pp 225-252. (Ch 1, 9, 10, 11, 12, 14, 15, 16, 18, 19)
*Ch.8
- [BEL77] L A Belady and M M Lehman, 'Large Programs and Large Scale Programming', Infotech State of the Art Report 37, Software Reliability, 1977, pp 15-27. (Ch 12)
- [BEL77b] L A Belady, 'Software Complexity', In Working Papers of the Software Life Cycle Management Working Party, 1977, pp 371-84. (Ch 12, 19)
- [BEL77c] L A Belady and P M Merlin, 'Evolving Parts and Relations - A Model of System Families', IBM Res Rep RC 6677, T J Watson Res Centre, Yorktown Heights, NY 10598, August 1977, 14p. (Ch 12, 14, 19)
*Ch.10
- [BEL77d] T E Bell, D C Bixler and M E Dyer, 'An Extendable Approach to Computer-aided Software Requirements Engineering' IEEE Trans on Software Eng, Vol SE-3, No 1, Jan 1977, pp 49-59. (Ch 19)

- [BEL78] L A Belady and M M Lehman, 'Characteristics of Large Systems', Part 1, Ch 3, pp 106-142 in *Ch.14 'Research Directions in Software Technology', Sponsored by Tri-Services Committee of the DOD, in Proc Conf Research Directions Software Technology, Oct 1978, Brown U, Providence, RI, and by MIT Press, 1979. (Ch 1, 2, 12, 17, 19, 20, 21)
- [BEL78b] L A Belady, 'Staffing Problems in Large Scale Programming', Proc Infotech State of the Art Conf, *Ch.13 'Why Software Projects Fail', Apr 1978, pp 4/1-4/12. (Ch 1, 14)
- [BEL79] L A Belady and C J Evangelisti, 'System Partitioning and Its Measure', IBM Res Rep RC 7560, Mar 1979. (Ch 15)
- [BEL79b] L A Belady, C J Evangelisti and J Cavanagh, 'GREENPRINT - A Graphical Representation for Structured Programs', IBM Res Rep, RC-7763, Jul 1979. (Ch 18)
- [BEL79c] L A Belady, 'On Software Complexity', IEEE NY Poly *Ch.15 Workshop on Quantitative Software Models for Reliability, Complexity and Cost, Kianasha Lake, NY, Oct 1979, IEEE Pub No Th0067-9, pp 90-94. (Ch 1, 2)
- [BEL80] L A Belady, 'Modifiability of Large Software Systems', Proc 14th IBM Comp Soc Symp Tokyo, Jan 8-13, Oct 1981.
- [BEL80b] L A Belady, Ed, 'Proceedings of the IEEE Special Issue on Software Engineering', Vol 68, No 9, Sept 1980. (Ch 19)
- [BEN81] G Benyon-Tinker, P G Harrison and M M Lehman, 'Complexity of Large Scale Software', Presented at the 1981 Minnowbrook Conf. (Ch 21)
- [BEN83] G Benyon-Tinker, P G Harrison and M M Lehman, 'Program Complexity', ICST Res Rep DoC 83/23, Oct 1983. (Ch 1, 2, 8)
- [BLA76] W W Black, 'The Role of Software in Successful Computer Applications', Proc 2nd ICSE, IEEE Cat No 76CH1125-4C, San Francisco, CA, Oct 1976, pp 201-205. (Ch 14)

- [BOE76] B W Boehm, 'Software Engineering', IEEE Trans on Computers, Vol C-25, No 12, Dec 1976, pp 1226-1241. (Ch 12, 14, 18, 19, 20, 21)
- [BOE76b] B W Boehm, J R Brown and M Lipow, 'Quantitative Evaluation of Software Quality', Proc 2nd ICSE, San Francisco, CA, Oct 1976, IEEE Cat No 76CH1125-4C, pp 592-605. (Ch 19)
- [BOE78] B W Boehm, 'Software Engineering - As It Is', Proc 4th ICSE, Munich, September 1979, IEEE Cat No 79CH1479-SC, Sept 1979, pp 11-21. (Ch 19)
- [BOE78b] B W Boehm and R W Wolverton, 'Software Cost Modelling - Some Lessons Learned', in 'Second Software Life-Cycle Management Workshop, 21-22 August 1978', Atlanta, GA, IEEE Pub No 78CH1390-4C, Dec 1978, pp 129-132. (Ch 19)
- [BOX70] G E P Box and G M Jenkins, 'Time Series Analysis', Holden-Day 1970, 17p. (Ch 7, 8)
- [BRA68] Dick H Brandon, 'Managing the Economics of Computer Programming, The Problem in Perspective', Proc 1968 ACM Nat Conf, Pub P-68, pp 332-334. (Ch 3)
- [BRO75] F P Brooks, 'The Mythical Man Month', Addison-Wesley, Reading, Mass, 1975, p206. (Ch 12, 17, 19)
- [BRO76] J C Browne. 'A Critical Overview of Computer Performance Evaluation', Proc 2nd ICSE, San Francisco, Oct 1976, pp 138-145. (Ch 14)
- [BUX70] J N Buxton and B Randell (Eds), 'Software Engineering Techniques', Report on a Conference sponsored by the NATO Science Committee, Rome, Oct 1970, 164p. Pub Brussels, 1970. 164p. (Ch 19)
- [BUX80] J N Buxton, 'Requirements for ADA Programming Support Environment - STONEMAN', US DoD, Washington DC, Feb 1980, 44p. (Ch 2, 19, 20)
- [CHE78] E T Chen, 'Program Complexity and Programmer Productivity', IEEE Trans Software Engineering, SE-4, pp 187-194, 1978. (Ch 16, 19)

- [CHO80] C K S Chong Hok Yuen, 'Phenomenology of Programs Maintenance and Evolution', PhD Thesis, ICST, DoC, London, Nov 1980, 302 p. (Ch 1, 19)
- [CIC75] A Cicu, M Maiocchi, R Polillo, A Sordini, 'Organising Tests During Software Evolution', Proc ICRS 75. (Ch 9)
- [CLI69] T E Climis, 'Old Problems, New Directions and the Art of the Practical', Proc IBM Symp on New Directions in Computer Technology, Vol 1, Poughkeepsie Laboratory Technical Rep TROO 1895, Jul 14 1969, pp 23-31. (Ch 3)
- [COB78] G W Cobb, 'A Measurement of Structure for Unstructured Programming Languages', Software Eng Notes, Vol 3, No 5, Nov 1978. (Ch 15)
- [COD67] E F Codd, R T Burger and L L Lunde, 'An Approach to Software Specification and Design', SDD Technical Rep TROO 1588, Poughkeepsie, May 31 1967. (Ch 3)
- [COM76] 'Conference Report, 2nd International Conference on Software Engineering', Computer, Dec 1976, P 71. (Ch 14)
- [COR69] F S Corbato, 'PL/1 as a Tool for Systems Programming', Datamation, May 1969, pp 68-76. (Ch 3)
- [COX66] D R Cox and P A W Lewis, 'The Statistical Analysis of Series of Events', Methuen, London 1966, p 38. (Ch 7, 8)
- [COX80] T A Cox (Ed), 'Proceedings of the Symposium on Formal Design Methodology', Cambridge, 1977, Pub STL, Harlow, Essex, 1980, 350p. (Ch 19)
- [CRS75] Proc Int Conf on Reliable Software, Los Angeles, CA, IEEE Cat No 75CHO 940-7CSR, April 1975. (Ch 9)
- [CUR78] B Curtis, S B Sheppard, M A Borst, P Milliman and T Love, 'Some Distinctions Between the Psychological and Computational Complexity of Software', Second Software Life Cycle Management Workshop, Atlanta, GA, USA, 21-22 Aug 1978 (New York: IEEE 1978), p 166-71. (Ch 15)

- [DAH72] O J Dahl, E W Dijkstra and C A R Hoare, 'Structured Programming', Academic Press, New York, 1972. (Ch 12, 19)
- [DAR64] Charles Darwin, 'On the Origin of Species'. Cambridge, Mass: Harvard Univ Press, 1964, facimille of the 1st edition. (Ch 4)
- [DAR79] J Darlington, 'Program Transformation: An Introduction and Survey', Comp Bull, Ser 2, No 22, Dec 1979, pp 22-24. (Ch 2, 19, 20, 21)
- [DEJ73] S P DeJong, 'The System Building System (SBS)', IBM Res Rep RC 4486, 15 Aug 1973. (Ch 10)
- [DEM78] T Demarco, 'Structured Analysis and System Specification', Yourdon Press, New York, NY, 1978, 352p. (Ch 19)
- [DEM79] R A Demillo, R J Lipton and A J Perlis, 'Social Processes and Proofs of Theorems and Programs', Comm ACM, Vol 22, No 5, May 1979, pp 271-280, and No 11, Nov 1979, pp 621-630. (Ch 19)
- [DER75] F deRemer and M Kron, 'Programming-in-the-Large versus Programming-in-the-Small', SIGPLAN Notices, pp 114-121, Jun 1975 (Ch 18)
- [DIJ68] E W Dijkstra, 'The Structure of "THE" Multiprogramming System', Comm ACM, Vol 11, No 5, May 1968, pp 341-346. (Ch 21)
- [DIJ68b] E W Dijkstra, 'A Constructive Approach to the Problem of Program Correctness', BIT, 8, 1968, pp 174-186. (Ch 7, 8, 19, 20)
- [DIJ68c] E W Dijkstra, 'GOTO Statement Considered Harmful', Letter to the Editor, Comm ACM, Vol 11, No 11, Nov 1968 pp 147-8. (Ch 6, 14)
- [DIJ70] E W Dijkstra, 'Structured Programming', EWD247, (Privately circulated), Techk Hochs, Eindhoven, 1970, (Ch 6)
- [DIJ72] E W Dijkstra, 'Notes on Structured Programming', In Dahl, Dijkstra and Hoare, 'Structured Programming', Academic Press 1972, pp 1-82. (Ch 7, 8, 14, 19)

- [DIJ72b] E W Dijkstra, 'The Humble Programmer', ACM Turing Award Lecture, Comm ACM, Vol 15, No 10, Oct 1972, pp 859-866. (Ch 8, 11, 14, 20)
- [DOL76] T A Dolotta and J R Mashey, 'An Introduction to the Programmer's Workbench', Proc 2nd ICSE, San Francisco, CA, Oct 1976, IEEE Cat No 76CH-1125-4C, pp 164-168. (Ch 2, 19, 20)
- [DRI68] G C Driscoll, 'A System for Processing Decision Tables in APL/360', IBM Res Rep RC 2065, Apr 1968. (Ch 3)
- [EMD71] M H Van Emden, 'An Analysis of Complexity', Mathematische Centrum, Amsterdam, 1971. (Ch 15)
- [FAG76] M I Fagan, 'Design and Code Inspections to Reduce Errors in Program Development', IBM Sys J, Vol 15, No 3, 1976, pp 182-211. (Ch 12, 14, 19, 20)
- [FAR65] L Farr and H J Zagorski, 'Quantitative Analysis of Programming Cost Factors: A Progress Report', Proc ICC Symp, Rome, North Holland, 1965, pp 384. (Ch 15)
- [FEL77] C P Felix and C E Walston, 'A Method of Programming Measurement and Estimation', IBM Sys J, Vol 16, No 1, 1977, pp 54-73. (Ch 19)
- [FIS58] Ronald A Fisher, 'The Genetical Theory of Natural Selection', New York: Dover, 1958, 2nd revised edition. (Ch 4)
- [FIT78] A Fitzsimmons and T Love, 'A Review and Evaluation of Software Science', Computing Surveys, Vol 10, No 1, March 1978, pp 3-18. (Ch 19)
- [FRE76] P Freeman and A I Wasserman, 'Software Design Techniques: A Tutorial', IEEE Press, 1976, 277 pages. (Ch 10)
- [GOL69] M M Gold, 'Time Sharing and Batch Processing, An Experimental Comparison of their Value in a Problem Solving Situation', Comm ACM, Vol 12, No 5, May 1969, pp 249-259. This paper includes a list of references to earlier studies. (Ch 3)

- [GOL73] J Goldberg (Ed), 'Proceedings of the Symposium on The High Cost of Software', Naval Postgraduate School, Monterey, Sept 1973, Pub SRI, Menlo Park, CA, 138p. (Ch 14, 18, 19)
- [G0075] J B Goodenough and S L Gerhart, 'Toward a Theory of Test Data Selection', IEEE Trans Software Eng, Vol SE-1, No 2, Jun 1975, pp 156-173. (Ch 19)
- [G0080] J B Goodenough, 'Software Quality Assurance Testing and Validation', "Proc IEEE", (Special Issue on Software Eng), Vol 68, No 9, Sept 1980. (Ch 19, 20)
- [GUE80] M Guerin, 'Research Study in Software Engineering: Tools for Requirements and Design Phases', Appendix 1 to AO/1-1237/80/NL/PP(SC), ESA Nordwijk, July 1980. (Ch 20)
- [HAL77] M Halsted, 'Elements of Software Science', Elsevier, 1977, 160pp. (Ch 15, 19)
- [HAN72] F M Haney, 'Module Connection Analysis - A Tool for Scheduling Software Debugging Activities', Proc Fall Joint Comp Conf, Anaheim, CA, 1972, pp 173-179. (Ch 15)
- [HAN76] S L Hantler and J C King, 'An Introduction to Proving the Correctness of Programs', Computing Surveys, Vol 8 No 3, Sept 1976, pp 331-353. (Ch 10)
- [HAR81] P G Harrison, 'The Finite State Machine as a Software Engineering Tool', IBM Res Rep, RC 8861, 1981, 19p, 'Title Efficient Table-Driven implementations of the finite State Machine' in J Sys and Software, Vol 2, No 3, Sept 1981 pp 201-212. (Ch 21)
- [HEN79] K Heninger, 'Specifying Requirements for Complex Systems: New Techniques and their Application', Proc Specifications of Reliable Software Conf, 1979, Cambridge, Mass, IEEE Cat No 79CH1401-9C, Mar 1979, pp 1-14. (Ch 19)
- [HER69] P S Herwitz, 'Programmer Evaluation and Considerations of Productivity', IBM Corporation, Armonk, Presented to Diebold Res Group, Chandler, Arizona, Jan 28 1969. (Ch 3)

- [HEY69] N Heyer, IBM Confidential Report on Manpower Trends, 1969. (Ch 3)
- [HOA69] C A R Hoare, 'An Axiomatic Basis for Computer Programming', Comm ACM 12, Vol 10, No 12, Oct 1969, pp 147, 576-583. (Ch 14, 19, 20)
- [HOA79] C A R Hoare, 'Review of a Paper by DeMillo, Lipton and Perlis: "Social Processes and Proofs of Theorems and Programs"', ACM Comp Rev, Vol 22, No 8, Rev No 34897, Aug 1979, p 324. (Ch 19)
- [H0075] D H Hooton, 'A Case Study in Evolution Dynamics', MSc thesis, ICST, CCD, London, Sept 1975, 61p. (Ch 7, 9, 11)
- [HUM68] W S Humphrey Jr, 'Address to R&E Conference', Oct 23 1968. (Ch 2)
- [HUT79] A F Hutchings, R W McGuffin, A E Elliston, B R Tauter, and P N Westmacott, 'CADES - Software Engineering in Practice' Proc 4th ICSE, Munich, Sept 1979, IEEE Cat No 79CH-1479-5C, pp 136-152. (Ch 1, 2, 19, 20)
- [IBM77] '1130 Continuous System Modelling Program', Order No H20-0209-1, IBM Data Processing Division, White Plains, NY, 10504. (Ch 8)
- [IBM73] OS/VS System Modification Program (SMP), IBM Corp GC28-0673. (Ch 10)
- [IBM76] 'Selectable Unit Packaging and Distribution', Programming Announcement, IBM, DP Division, White Plains, NY, May 1976. (Ch 14)
- [ICSE1] Proc First International Conference on Software Engineering, Washington, DC, 8/9 Sept 1975, IEEE Cat No 75 Ch 0992-8c. (Ch 9)
- [ICSE2] Proc Second International Conference on Software Engineering, San Francisco, Oct 1976, IEEE Cat No 76 Ch 1125-4c. (Ch 14)
- [JAC75] M A Jackson, 'Principles of Program Design', Academic Press, London, 1975, 297p. (Ch 19)

- [JOH67] D B Johnston and A M Lister, 'Software Science and Student Programs', Software : Practice and Experience, Vol 10, No 2, Feb 1980, pp 159-160. (Ch 19)
- [JON80] C Jones, 'Software Development - A Rigorous Approach', Prentice-Hall Inc, NJ, London 1980, 400p. (Ch 19)
- [JON80b] C B Jones, 'The Role of Formal Specifications in Software Development', Proc Infotech State of the Art Conf on Life Cycle Management, 1980, p 20. (Ch 19)
- [KEN69] Kenzo-Inque, 'On the Development of Large Scale Operating Systems in Japan', IRIA Colloquium on High Powered Computing Systems, Paris 1969. (Ch 3)
- [KER76] B W Kernigham and P J Plauger, 'Software Tools', Addison-Wesley, Reading, Mass, 1976, 366p. (Ch 10)
- [KOE71] A Koestler, 'The Act of Creation', Pan Books, Chapter 7, final paragraph, beginning the bottom of page 176. 1971. (Ch 11)
- [KOP79] H Kopetz, F Ohnert and W Merker, 'An Outline of Project Mars - Maintainable Real-time Systems', Bericht 79-09, Technische U Berlin, July 1979, 19p. (Ch 19)
- [KOW79] R A Kowalski, 'Logic For Problem Solving', North Holland Elsevier, 1979, 278p. (Ch 1, 20)
- [LAC68] K L LaCroix, 'Report on a Survey of Currently Available Evaluation Tools', Proc SIMSYMP 1968, IBM Res Div, Nov 1968, pp 3-1 - 3-22. (Ch 3)
- [LEH66] M M Lehman, 'A Survey of Problems and Priliminary Results Covering Parallel Processing and Parallel Processors', Proc IEEE, Spec Iss on Computers, Vol 54, No 12, Dec 1966, pp 1889 - 1901. (Ch 1)
- [LEH68] M M Lehman, 'Mediocrity in Middle Management', Unpublished MSS 1968. (Ch 4, 5, 7, 17)
- [LEH68b] M M Lehman and J L Rosenfeld, 'Performance of a Simulated Multiprogramming System', Proc FJCC, 1968, p 1431-1442. (Ch 1, 20)

- [LEH69] M M Lehman, 'The Programming Process', IBM Res Rep
*Ch.2 RC 2722, Dec 1969, p 46. (Ch 1, 2, 4, 6, 7, 8, 12,
19)
- [LEH69a] Private communication, P S Herwitz, IBM, CHQ, to M
M Lehman, dated Jul 3 1969. (Ch 3)
- [LEH74] M M Lehman, 'Programs, Cities and Students -
*Ch.7 Limits to Growth?', Inaugural Lecture Series, Vol
9, ICST, London, May 14, 1974, pp 211-229. Also
in 'Programming Methodology', D Gries (ed),
Springer-Verlag, 1978, pp 42-69 (Ch 1, 8, 9, 11,
12, 14, 17, 19)
- [LEH74b] M M Lehman, 'Programming Systems Growth Dynamics',
Infotech State of the Art Lectures, No 20, 1974,
pp 391-412. (Ch 19)
- [LEH76] M M Lehman, 'Human Thought and Action as an
*Ch.11 Ingredient of System Behaviour', In The
Encyclopedia of Ignorance, R Duncan and M Weston-
Smith (eds), Pergamon Press, London, 1976, pp 347-
354 (Ch 1, 9, 12, 14, 17, 19, 20, 21)
- [LEH76b] M M Lehman and F N Parr, 'Program Evolution and
*Ch.8 its Impact on Software Engineering', Proc 2nd
ICSE, San Francisco, 1976, IEEE Cat No 76 Ch 1125-
4c, pp 350-357. (Ch 1, 9, 12, 14, 16, 19)
- [LEH76c] M M Lehman, 'Notes on the Evolution Dynamics of
Large Programs', MML-138, Feb 1976. Privately
Circulated. (Ch 11)
- [LEH76d] M M Lehman, 'OS-VS2-MVS-Long Range Prognosis',
Private Comm, MML-104, Apr 15, 1975, 13p. (Ch 19)
- [LEH77] M M Lehman and F N Parr, 'Program Evolution
Dynamics and its Role in Software Engineering and
Project Management', Software Systems Engineering,
Proc Eurocomp Conf, London, Sept 1977, pp 393-412.
(Ch 12, 14)
- [LEH77a] M M Lehman, 'Complexity and Complexity Change of
a Large Applications Program', ERO Res Proposal,
Mar 1977. (Ch 12)

- [LEH77b] M M Lehman, 'Software Engineering Research Projects', ICST-CCD Res Rep 77/7, March 1977. (Ch 12)
- [LEH77c] M M Lehman, 'The Funnel - A Functional Channel', ICST, CCD Res Rep 77/29, Jul 1977, 14p. Also IBM Technical Disclosure Bulletin, 1976. (Ch 12, 14, 19)
- [LEH77d] M M Lehman, 'Evolution Dynamics: A Phenomenology of Software Maintenance', In SLC77, 1977, pp 324-323. (Ch 12)
- [LEH77e] M M Lehman and J Patterson, 'Preliminary CCSS System Analysis Using Techniques of Evolution Dynamics'. In Working Papers of the (first) Software Life Cycle Management Workshop, Airlie VA, 1977. Published by ISRAD/AIRMICS, Computer Systems Command, US Army, Fort Belvoir, VA, Dec 1977, pp 324-332. (Ch 12, 19)
- [LEH77f] M M Lehman, 'On Productivity, Structured Programming and all That', ICST CCD Res Rep 77/51, Dec 1977. (Ch 12)
- [LEH77g] M M Lehman, 'Complexity and Complexity Change of a Large Applications Program', ERO Res Proposal, Mar 1977, 32p. (Ch 19)
- [LEH77h] M M Lehman and L H Putnam (Eds), in 'Software Phenomenology Working Papers of the (first) Software Life Cycle Management Workshop, Airlie, VA, August 1977', Published by ISRAD/AIRMICS, Computer Systems Command, US Army, Fort Belvoir, VA, Dec 1977, 682 p. (Ch 19)
- [LEH78] M M Lehman, 'Laws of Program Evolution - Rules and Tools of Programming Management', Proc Infotech State of the Art Conf, 'Why Software Projects Fail', Apr 1978, pp 11/1-11/25. (Ch 1, 14, 17, 19)
- [LEH78b] M M Lehman, 'Laws and Conservation in Large Program Evolution', Proceedings of the 2nd Software Life Cycle Management Workshop, August 1978, Atlanta, IEEE Cat No 78CH1390-4C, pp 140-145. (Ch 16, 17)

- [MAI84] T S E Maibaum and W M Turski, 'On What Exactly is Going On When Software is Developed Step by Step', Proc 7th ICSE, Orlando, FA. March 1984. Publ IEEE Comp Soc, Silver Spring, MA, IEEE Cat No 84ch2011-5, pp 525 - 533.
- [MAY63] Ernst Mayr, 'Animal Species and Evolution', Cambridge, Mass: Harvard University Press, 1963. (Ch 4)
- [MCC76] T J McCabe, 'A Complexity Measure', IEEE Trans Software Eng Vol SE-2, No 4, Dec 1976, pp 308-320. (Ch 15, 16, 19)
- [MCI68] M D McIlroy, 'Mass Produced Software Components', Software Engineering Rep on Conf supported by NATO Science Committee, Garmisch, Oct 1968. Pub NATO Scientific Affairs Committee, Brussels, Jan 1969, pp 138-142 (Ch 21)
- [MCI72] M D McIlroy and C Boon, 'The Outlook for Software Components', Infotech State of the Art Rep No 11, 'Software Engineering', 1972, pp 243-252. (Ch 14)
- [MEA72] D H Meadows et al, 'The Limits to Growth', Signet, 1972. (Ch 7)
- [MEL67] B F Melkun and W R Brittenham, 'BSL: A Basic Systems Language', SDD Technical Rep TROO. 1600, Proc Programming Symposium - 1967, Swamscott, Mass, 1967. (Ch 14)
- [MIL65] R E Miller, 'Switching Theory', John Wiley, 1965. (Ch 10)
- [MIL76] H D Mills, 'Software Development', IEEE Trans Software Eng, SE-2, No 4, Dec 1976, pp 245-273. (Ch 12)
- [MIL73] M Mills, 'The Complexity of Programs', in 'Program Test Methods', W C Hetzel (Ed), Prentice Hall, 1973, pp 225. (Ch 15)
- [MIL78] E Miller and W E Howden (Eds), 'Tutorial: Software Testing and Validation Techniques', IEEE Comp Soc, IEEE Cat No EHO 138-8, 1978, 423 p. (Ch 19)

- [MUS80] J Musa, 'The Measurement and Management of Software Reliability', Proc IEEE (Spec Iss on Software Engineering, L A Belady (Ed)), Vol 68, No 9, Sept 1980, (Ch 19)
- [MYE75] G J Myers, 'Reliable Software Through Composite Design', Petrocelli, NY, 1975. (Ch 14, 15)
- [MYE77] G J Myers, 'An Extension to the Cyclomatic Measure of Program Complexity', SIGPLAN Notices, Oct 1977. (Ch 15)
- [MYE78] G J Myers, 'Composite/Structured Design', Van Nostrand Reinhold, New York, NY, 1978, 134p. (Ch 19)
- [NAT69] 'Software Engineering', Rep on Conf sponsored by NATO Science Committee, Garmisch, 1968. Scientific Affairs Division, NATO, Brussels 39, Jan 1969. (Ch 3)
- [NAU69] P Naur and B Randell, 'Software Engineering', Rep on Conf sponsored by NATO Science Committee, Garmisch, 1968, Proc Publ by Scientific Affairs Division, NATO, Brussels 39, Jan 1969, 231p. (Ch 1, 3, 7, 19, 20)
- [NEW53] J Von Newman and O Morgernstern, 'Theory of Games and Economic Behaviour', Princeton U Press, 1953. (Ch 10)
- [NEW76] 'IBM seeks "selectable units" to solve huge software tasks', New Scientist, Vol 71, No 1019, Sept 23, 1976. (Ch 10)
- [NOR77] P V Norden, 'Project Life Cycle Modelling: Background and Application of the Life Cycle Curves', In Working Papers of the Life Cycle Management Workshop, 1977, pp 217-306. (Ch 12, 17)
- [OXF33] 'The Oxford English Dictionary', Vol. III, Clarendon Press, Oxford, 1933, p 354, definitions 1, 5, 7. (Ch 2)
- [PAD64] A Padega, 'The Structure of System 360 - Channel Design Considerations', IBM Sys J, Vol 3, No 2, 1964, pp 165-180. (Ch 14)

- [PAR67] D L Parnass and J A Dorringer, 'SODAS and a Methodology for System Design', AFIPS Conf Proc Vol 31, 1967 FJCC, Thompson Books, Washington, DC, pp 449-474. (Ch 3)
- [PAR69] D L Parnass, 'More on Simulation Languages and Design Methodology for Computer Systems', AFIPS Conf Proc, Vol 34, 1969 SJCC, AFIPS Press, Montvale, NJ, pp 739-743. (Ch 3)
- [PAR72] D L Parnass, 'On the Criteria to be used in Decomposing Systems into Modules', Comm ACM, Vol 15, No 12, Dec 1972, pp 1053-8. (Ch 11, 12, 14, 18, 19)
- [PAR76] D L Parnass, 'On the Design and Development of Program Families', IEEE Trans on Software Eng, Vol 2, No 1, Mar 1976. (Ch 10)
- [PAR77] F N Parr and M M Lehman, ICST, CCD Res Rep 77/15, London, 102p. (Ch 19)
- [PAR80] F N Parr, 'An Alternative to the Rayleigh Curve Model for Software Development Effort', IEEE Trans on Software Eng, Vol 6, No 3, May 1980, pp 291-296. (Ch 19)
- [PEA79] D Pearson, 'Software Engineering - A New Approach', Telesis, Oct 1979, pp 23-27. (Ch 20)
- [PET80] L Peters, 'Software Design Engineering', Proc IEEE, (Spec Iss on Software Eng, L A Belady (Ed)), Vol 68, No 9, Sept 1980, pp 1085 - 1093. (Ch 19)
- [PUT76] L H Putnam, 'A Macro Estimating Methodology for Software Development', 13th IEEE Computer Society Int Conf, 1976, Washington, DC, 7 - 10 Sep 1976, pp 138-43, IEEE Cat No 76CH1115-5C. Proc Compon 76 fall, 1976. (Ch 14)
- [PUT77] L H Putnam, 'The Influence of the Time-Difficulty Factor in Large Scale Software Development', In Working Papers of the Life Cycle Management Workshop, 1977, pp 307-312. (Ch 12, 17, 19)
- [PUT77a] L H Putnam and R W Woverton, 'Quantitative Management - Software Cost Estimating', Comp Soc 77, IEEE Comp Software and Applications Conf

- (Tutorial), IEEE Cat No EHO 129-37, Nov 1977, 326p. (Ch 19)
- [RAB77] M O Rabin, 'Complexity of Communications', 1976 Turing Award Lecture, Comm ACM, Vol 20, No 9, Sept 1977. pp 625-633. (Ch 12)
- [RAN71] B Randell, 'Operating Systems - The Problem of Performance and Reliability', Invited paper, Proc IFIP Cong 71, Information Processing 71, Ljubljania, Yugoslavia (2 vols), pp 1.100-9. (Ch 7)
- [RID80] W E Riddle and R E Fairly, 'Software Development Tools', Proc Pingree Park Workshop, May 1979, Springer Verlag, New York, 1980, 280p. (Ch 1, 20)
- [RIO76] J S Riordan, 'An Evolution Dynamics Model', ICST CCD Res Rep 76/13, 22p. (Ch 8, 12, 14)
- [RIO77] J S Riordan, 'An Evolution Dynamics Model of Software Systems Development', in 'Software Phenomenology - Working Papers of the (First) SLCM Workshop', Airlie, Virginia, Aug 1977. Pub ISRAD/AIRMICS, Comp Sys Comm US Army, Fort Belvoir, VI, Dec 1977, pp 339 - 360. (Ch 16, 19)
- [ROS77] D T Ross and K E Schoman, 'Structured Analysis for Requirements Definition', IEEE Trans on Software Eng, Vol SE-3, No 1, Jan 1977, pp 6-15. (Ch 19)
- [ROS77a] D T Ross, 'Structured Analysis (SA): A Language for Communicating Ideas', IEEE Trans on Software Eng, Vol 3, No 1, Jan 1977, pp 16-33. (Ch 19)
- [SCH66] A Scherr, 'Analysis of Main Storage Fragmentation', Proc Symp Storage Hierarchy Systems, TR 00 1556, Dec 1966, pp 159-174. (Ch 20)
- [SDD69] Programming Development Support Strategy, June 23 1969. (Ch 3)
- [SDD69b] 'Consolidated Performance Activity Report', No 3, SDD, OS/360 Performance Analysis Department, Poughkeepsie, July 1969. (Ch 3)
- [SEP81] Proc Software Eng Productivity Workshop, San Diego, March 1981. (Ch 21)
- [SILT] 'SILT' presentation at SHARE XLM. (Ch 14)

- [SIM68] Proc SIMSYMP 1968, IBM Res Div, Nov 1968. (Ch 3)
- [SIM69] H A Simons, 'The Science of the Artificial', MIT Press, 1969. 123p. (Ch 2, 6, 10, 11, 12, 14, 15, 17, 19, 20)
- [SHA80] N Shaw, 'The Impact of Abstraction Concerns on Modern Programming Languages', Proc IEEE (Spec Iss on Software Eng, L A Belady (Ed)), Vol 68, No 9, Sept 1980. (Ch 19)
- [SLC77] Software Phenomenology: Working Papers of the Software Life Cycle Management Workshop, Airlie Va, Aug 1977, Pub ISRAD/ AIRMICS Comp Sys Comm, US Army, Fort Belvoir, VA, Nov 1977.(Ch 12)
- [STE74] W P Stevens, G J Myers and L L Constantine, 'Structured Design', IBM Sys J, Vol 11, No 2, 1974, pp 115-139. (Ch 14, 19)
- [SUT75] J W Sutherland, 'System: Analysis, Administration, Architecture', Van Nostrand-Rheinhold, NY, 1975. This book is a good reference source of the literature in the field of systems science. (Ch 12, 14)
- [SWA] G H Swaum, 'Top-Down Structured Design Techniques', Petrocelli Books Inc, New York, NY, 140p. (Ch 19)
- [TEI77] D Teichroew and E A Hershey II, 'PSL/PSA: A Computer-aided Technique for Structured Documentation and Analysis of Information Processing Systems', IEEE Trans Software Eng, Vol SE-3, No 1, Jan 1977, pp 41-48. (Ch 19)
- [TUR75] W M Turski, 'Software Engineering - Some Principles and Problems', Mathematical Structures - Computational Mathematics - Mathematical Modelling, Paper dedicated to Prof L Iliev's 60th Anniversary, Sofia, 1975. pp 485-91. (Ch 11)
- [TUR78] W M Turski, 'Computer Programming Methodology', Heyden, London, 1978, 208p. (Ch 12, 19)
- [TUR79] W M Turski, 'Report on an SRC-sponsored Visit to Imperial College', ICST CCD Res Rep, Oct 1979, 2p. (Ch 19)

- [TUR81] W M Turski, 'Specification as a Theory in the Computer World and in the Real World', Infotech State of the Art Report 'System Design', Se 9, No 6, Pergamon Infotech Ltd, Maidenhead, 1981, pp 363-377. (Ch 1)
- [VAN76] P Van Leer, 'Top-Down Development Using an Orogram Design Language', IBM Sys J, Vol 15, No 2, 1967, pp 155-170. (Ch 19)
- [WAG75] H Wagner, 'Principles of Operations Research', Prentice Hall, Englewood Cliffs, NJ, 1975, 2nd Edition. (Ch 16)
- [WEB59] 'Websters New Collegiate Dictionary', 1959 Edition, p 286, also Unabridged New International Edition 1979, p 789. G C Merriam Co, Springfield, Mass. (Ch 2)
- [WEI70] G M Weinberg. 'Natural Selection as Applied to Computers and Programs', General Systems, vol 15, pp 145 - 150, 1970. Also in Tutorial on Software Maintenance, published by IEEE Comp Soc Press, Order No 453, IEEE cat no EH0201-4, 1982, pp 191-198.
*Ch.4
- [WEI74] L Weissman. 'Psychological Complexity of Computer Programs', SIGPLAN Notices, Vol 9, No 6, June 1974. (Ch 16)
- [WEI77] L Weissman. 'A Methodology for Studying the Psychological Complexity of Computer Programs', U Toronto, R77-230, pp 240. (Ch 16)
- [WIR71] N Wirth. 'Program Development by Stepwise Refinement', Comm ACM, Vol 14, No 4, Apr 1971, pp 221-7. (Ch 11, 12, 19, 21)
- [WIR78] N Wirth. 'Modula-2', ETH Institute for Informatics, Dec 1978, 36p. (Ch 21)
- [WIR79] N Wirth. 'The Module: A System Structuring Facility in High-Level Programming Languages', Proc Symp Prog Languages and Methods, Sydney, Australia, J Tobias AAEC (Ed), Lucas Heights, NSW, 1979. (Ch 18)
- [WOO79] M R Woodward, M A Hennell and D Hedley, 'A Measure of Control Flow Complexity in Program Text', IEEE-

Trans Software Eng, Vol SE-5, No 1, Jan 1979, pp 45-51. (Ch 15)

- [W0079b] C M Woodside. 'A Mathematical Model for the Evolution of Software', ICST CCD Res Rep 79/55, Apr 1979. Also in J Sys and Software, Vol 1, No 4, Oct 1980, pp 337-345 (Ch 1, 17, 19)
- [WUL77] W A Wulf, 'Languages and Structured Programs', in 'Current Trends in Programming Methodology', Edited by R T Yeh, Prentice-Hall Inc, Englewood Cliffs, NJ, 1977, pp 33-60. (Ch 19)
- [YEH77] R T Yeh (Ed), 'Current Trends in Programming Methodology', Vol 1 Software Specification and Design, Prentice Hall Inc, Englewood Cliffs, NJ, 1977, 275 p. (Ch 19)
- [YEH80] R T Yeh and P Zave, 'Specifying Software Requirements', Proc IEEE, (Spec Iss on Software Eng, L A Belady (Ed)), Vol 68, No 9, Sept 1980, pp 1077 - 1085. (Ch 19)
- [YIN78] B H Yin and J W Winchester, 'The Establishment and Use of Measures to Evaluate the Quality of Software Design', Software Eng Notes, Vol 3, No 5, Nov 1978, pp 45 - 52 (Ch 15)
- [YOU68] E Yourdon, 'Structured Walk-throughs', (2nd Edition), Yourdon Inc, New York, 1968, 137p. (Ch 19)
- [ZAV81] P Zave and R T Yeh, 'Executable Requirements for Embedded Systems', Proc 5th ICSE, 9-12 March 1981, IEEE Cat No 81 CH 1627-9, pp 295-304. (Ch 21)
- [ZUR67] F W Zurcher and B Randell, 'Iterative Multi-Level Modeling - A Methodology for Computer System Design', IBM Res Div Rep RC - 1938, Nov 1967. Also Proc IFIP Congr, 1968 Edinburgh, Aug 1968, pp D138-142. (Ch 1, 3, 19, 20, 21)

Index

A

- Abstract ideal process, 22
- Abstraction, 361
- Academic world, computer use, 394
- Accounting, oil royalty, 89, 92
- Accuracy, 393
- Achievable plan, 264
- Action, human, 237
- Activity
 - constant but stratified, 107
 - elements, 110
 - threshold, 108
 - total, 108
 - dynamics, 99
 - human, 302, 315, 333, 408
 - inter-level, 487
 - intra-level, 487
 - major classes, 444
 - phases, 275
 - repair, 318
 - targets, 384
- Actual system, 228
 - permitted, 228
 - set, 228
- Ada, 394, 423, 468, 480
 - future requirements, 459
- Adaptation, 20, 27
 - closed-loop cyclic, 169
 - continuous, 169
 - implementation by code, 395
 - intellectual effort involved, 421
 - relationships, mutual, 91
 - systems, 241
 - tool, 421
- Addition of detail, 32
- Advanced systems-programming language,
 - 74
- Age, 188, 206, 207, 304
 - cumulative work achieved, 210
 - expected ν observed handles, 216
 - fraction of modules, 214
 - system, 172
 - parameter, 172
 - system x, 427
- Aging, 189, 410
 - process, 172
 - software project, 415
- Air Traffic Control, 289, 402
 - system, 475
- Aircraft, 279, 283, 356
- ALGOL, 68, 72, 185, 204, 325, 394
- Algorithms
 - analysis, 332
 - development, 139, 165
 - large programs, 375
- Altdorfer, Albercht, 134
- Alte Pinakothek Museum, 133
- Alterability
 - concept, 420
 - dynamic correctness, 420
- Alternative process decompositions, 473
- Analog, 193
- Analysis, critical activity, 492
- Ancestor systems, 203
- AND, 223
- Ant, 138, 162
- Anti-regressive activities, 149–152, 154–157,
 - 162, 192, 193, 325, 329, 424
 - city life, 153
 - Club of Rome report, 156
 - conflict and balance, 154
 - education, 158, 159
 - investment, 267, 319
 - middle management, 155
 - neglect, 156
 - non-uniqueness of assignments, 154
 - organizations, 155
- APL/360, 70, 76

- Apollo space program, 116
 - Application
 - concept, 461, 476
 - evolution dynamics studies, 217
 - mathematical model, 352
 - Applied dynamics, 418
 - Approach
 - informal, 332
 - measures, 471
 - Appropriate process technology, 497
 - Arbitrary units, 224
 - Architects, 355, 359
 - Architecture
 - new system, 441
 - system, 426
 - systematics, 65
 - Archival system, 127
 - Artifacts, 355, 356
 - Artificial
 - activity, 243
 - complex, 439
 - evolution, 458
 - parameter values, 347
 - sciences, 238
 - selection, 87, 92
 - system, 14, 243, 300, 380, 408, 409
 - Artist, 277
 - Assembling, 459
 - Assembly language, 71, 74, 203
 - Assembly line, 278
 - process, 442, 469
 - Atmosphere, 156
 - Attention, system module, 172
 - Attributes
 - identifiable, 166
 - input, 470
 - output, 470
 - Automatic
 - program compilation, 394
 - searching, 459
 - Automobile, 279
 - Availability, new hardware, 174
 - Average absorption level, 390
 - Average growth trends
 - particular attributes, 168
 - planned growth and, 167
 - system attributes, 167
- B**
- B5500, 72
 - B8500, 72
 - Bacteria, 86
 - Bank accounts, 289
 - Banking, 356
 - Banking application system, 296
 - Banking system growth, 307
 - Base control, 56
 - Base-line, 285
 - Basic steps, programming process, 481
 - Basis, family, 222
 - Bats, 91
 - Battle scene, 135, 139
 - Battlefield, 133
 - Baumol's principle, 150
 - effect, 151
 - Behavior
 - mathematical model validation, 347
 - Beilner, Heinz, 200, 331
 - Belady, L. A., 220
 - Belagarung Von Alexia, 136
 - Bell-shaped curve, 126, 127
 - Bell Telephone Laboratory, 1
 - IBM Tss/360 use, 69, 70
 - Berkeley model, 148
 - Bevier, R., 49
 - Biblical viewpoint, 162
 - Binary
 - activity-tree, 104, 105
 - card loading routine, 93
 - Biological
 - organism, 377
 - sciences, 376
 - systems, 252, 301
 - Biologists, 237
 - Birth rate, 157
 - Bivariate relationships, 173
 - Black box units, 326
 - Blueprints, 422, 459
 - technology, 423
 - Boehm's model, 440, 444
 - Bookkeeping, 318
 - Boolean, 368
 - function, 230
 - variable, 223, 225
 - Bottom-up synthesis, 473
 - Breeding, selective, 87
 - BSL, 71, 74
 - Budget, 192
 - control, 384
 - simulation example output, 195
 - value, 346
 - Bug, 89, 92, 93, 95, 99, 100, 372
 - Bureaucracy, 277
 - Burroughs, 72
 - Business

- considerations, 217
- release slippage consequence, 435
- Buzzwords, 491
- C**
- Caesar, Julius, 315
- Calculus, 24
- Capability, 396
- Categories, approach, 332
- Cathode ray tube, 194
- C, cross-correlation, 111, 112
 - total activity, 117
- Cellular systems, collective behavior, 376
- Chain of reasoning, 495
- Challenges, programming, 282–284
- Change, 186
 - propagation, 337
 - refamiliarization, 385
- Changeability, multifunction programs, 202, 250
- Characteristics of large systems
 - communication, 319
 - complex interactions, 312
 - continuing enhancement, 300
 - continuing maintenance, 302
 - documentation, 319
 - empirical study, 306
 - future, 325
 - life-cycle cost pattern, 312
 - local and global optimization, 323
 - measurement in software engineering, 291
 - nature of largeness, 289
 - part-number explosion, 318
 - product ν process knowledge, 317
 - program collections, 314
 - program systems, 313
 - software: knowledge, skill and communication, 315
 - specialization, human activities, 315
 - structure reflecting manufacturing process, 321
 - system behavior, optimization, 322
 - system performance, execution, 322
 - traditional indicators, 292
 - variety, 300
- Chess program, 399
- Chief programmer teams concept, 3, 266
- Clean-up, 426
 - points, 145
 - release, 430
- Clear-Caster project, 67
- Clerks, 459
- Closed loop control, 55
- CLU, 368
- Club of Rome, 156
- Coagulation, 235
- COBOL, 62, 72, 394
- Code, 99, 126, 129, 176, 202, 203, 236, 253, 262, 263, 275, 282, 296, 320, 321, 377
 - changes, 177, 378, 385, 386
 - deletion, 301
 - efficient sequences, 394
 - existing, 410
 - generation, 139, 165
 - healthy, 325
 - low level, 360
 - machine, 359
 - repairs, 446
 - rigidity program, 413
 - source, 492
 - structure, 422
 - supportable, 321
 - timely, 218
 - unfamiliar, 386
- Coding, 317
 - reduced quality, 175
- Coherent process, 452
- Collection, programs, 314
- Commercial computer use, 66, 394
- Complexity, 204, 253, 492, 494
 - activity required, 178
 - changing, 172
 - concept, 199
 - control, 284
 - definition, 172, 173, 331
 - external ν internal ν intrinsic, 254
 - fixed fault, 218
 - growth, 120
 - internal, 179
 - increasing, 115, 259, *see also* Release interval
 - intuitive models, 361
 - measures, 218
 - minimum starting point, 420
 - parameters, 173
 - programming system dynamics, 110
 - reduction, 488
 - test text length correctness, 336
 - time, 114
- Communication
 - cost, 147
 - interhuman, 315
 - inter-software unit, 328

- Communication (*continued*)
 - intra-system, 313
 - key, 319
 - linkages, 314
 - v personal relations, 78
 - programming roles, 281
 - team spirit, 286
- Comparative measures, complexity, 333
- Compilers, 92, 93, 95, 276, 359, 368
 - type checking, 372
- Compiling, 459
- Complete paradigm, 29
- Complex
 - large system interactions, 312
 - problems, 331
- Components, 87, 123
 - independence, 277
 - numerical increase, 90, 95
 - partitioning, 239
 - peripheral, 97
 - release growth, 124
- Computational
 - complexity, 253
 - model, 479
- Computer
 - adaptive population, 87
 - environment, 91
 - science, 219, 253
 - scientists, 237, 445
- Computer-user community, 393
- Computing application development, 460
- Concept, code conversion process, 419
- Concepts
 - computer application, 473
 - operational system transformation, 473
 - program methodology, 133
 - reasonableness, 173
 - simple statement, 475
- Concordance, 251
- Concorde, 239
- Conditions, natural selection, 85
- Configurable family model, 234
- Configuration, 221
 - management, 5, 65
 - permitted, 224, 230
- Connection, program, 249
- Consensus, 475
- Conservation, large program life cycle
 - evolution
 - execution laws, nature, 376
 - feedback consequences, increasing understanding, 379
 - gross nature, 379
 - regularity cause, 377
 - scientific laws spectrum place, 376
- fifth law interpretation, 385
 - ability averaging, human interactions, 389
 - change and refamiliarization, 385
 - familiarity and statistically invariant release content, 390
- laws, 380
 - continuous change, 380
 - familiarity, 384
 - increasing complexity, 384
 - large program evolution, fundamental, 383
 - organizational, 168, 170
 - stability, 381
- Consolidation effect, 176
- Constraints, software managers, 422
- Constructive correctness, 31
- Constructs
 - more efficient, 231
 - non-universal, 231
- Continuing
 - enhancement, 300
 - maintenance, 302
- Continuous evolution, 240, 245
- Contraceptive advances, 157
- Control
 - base, 56
 - blocks, 365, 369
 - modules, 365, 366
 - closed loop, 55
 - environmental, 54
 - evolution, 221
 - growth, 54
 - open loop, 55, 58
- Corrective activity, 103, *see also* Release
- Correctness, 201, 301, 396, 398, 441, 453, 464, 492
 - absolute, 406
 - clarification of concept, 405
 - judgments, 400
 - mathematical terms, 405
 - proving, 68, 165
 - questions, 496
- Correlation, 110, 111
 - serial, 142, 168
- Corrosion, 395
- Cost, 110, 148, 492
 - labor, 360
 - model problems, 242
 - over-runs, 265
 - programming systems, 48

- release, 123
 - structure, 50
 - activities, 50
 - components, 50
 - Cost-effective
 - maintenance, 387
 - parameters, 250, 265, 300
 - replacement, 412
 - usage, 449
 - CP 67, 65
 - Creativity, 275
 - Criminal activity, 153
 - Critical
 - growth mass, 177
 - size, 196, 197
 - Cross-correlation, 110, 111
 - Crystallography, 237
 - Cumulative
 - handle ν system age, 180, 181
 - models handled, 268
 - work, 210
 - work count, 211
 - Current-Idealized model, 443
 - Current programming process, 15
 - Curves, three classes, 126
 - Customer, 285
 - delivery, 303
 - involvement, 284
 - support teams, 433
 - virtual, 285
 - Cyclic, 178
 - components, 181
 - effects, 383
 - invariance, 267
 - pattern, 267
 - terms, 182
 - trends, 174
 - Cyclically self-regulating, 170
 - Cyclicity, 185
 - growth process, 175
- D**
- DAL, 147, *see* Documentation, accessibility, and learnability
 - Darwin, Charles, 85, 86, 94
 - Data, 140, 173
 - abstraction, 362, 367, 369
 - averaged, 141, 167
 - environment, 92
 - later, 141
 - models handled, 209
 - program evolution, 203
 - scattered, 142
 - smoothed, 141
 - variations, 167
 - Dead installation, 241
 - Death
 - process, 382
 - rate, 157
 - Debug, 276, 362
 - Decay, 143, 150, 154, 155, 169, 256, 381, 382, 383
 - cumulative, 193
 - trends, 131
 - Decision-table functional definition, 76
 - Declaration, 420
 - Decomposition, 234
 - primitive actions, 475
 - process, 407
 - software systems, 276
 - Defects, 185
 - detection, 421
 - ν fault, 185
 - Defense agencies, national, 439
 - Defense systems, 289
 - Degenerate cases, 461
 - Degree of concentration, 215
 - Degrees of freedom, 183, 190
 - Delay, significance of, 400
 - Delivery
 - new release, 282
 - problems, 434
 - De-pollution activities, 154
 - Design, 28, 29
 - bugs, 302
 - horizontal verification and validation, 484
 - machine sided, 368
 - model, 475
 - modifiability, 367
 - process
 - simplify, 470
 - programming-distribution-usage system, 175
 - Designer, 359, 386
 - Destruction, local information, 472
 - Deterioration, 383
 - system, 395
 - Deterministic approach, 336
 - Development
 - beyond target, 55
 - distribution work, 211
 - model
 - evolution laws, 169
 - formal, 185
 - fault penetration, 187
 - interpretation, 191
 - limited growth, 194

- Development (*continued*)
 - management decision, 192
 - management simulation, 193
 - program faults, 185
 - programming, 165
 - process, 167
 - statistical, 170
 - available data, 171
 - observables, 172
 - present, 173
 - study basis, 170
 - relationship, 174
 - work rate, 182
 - size, 182
 - system approach, 165
 - Diagnostic programming, 96
 - Diagram, Nassi-Sneiderman, 363
 - Differential equations, 479
 - families, 149
 - release content and relationship, 388
 - Digital, 193
 - Dining philosophers, 397
 - Discipline, 393
 - Disciplined analysis, activities, 474
 - Discomfort, unfamiliarity, 415
 - Disorder
 - introduction, growth, 342, 353
 - reduction, structural work, 345
 - Distributed
 - effort, axiom, 101
 - system, 407
 - Distribution, development work, 211
 - Division, operating system, 175
 - Documentation, 129, 145, 147, 165, 176, 191, 202, 315, 319, 320, 441, 468
 - accessible, 319
 - accessibility and learnability, 191
 - capability, 459
 - deficiencies, 186
 - logging, 175
 - system check, 114
 - testing, 139
 - DOD report, 468, 498
 - Domain, interest, 475
 - Dortmund, University of, 331
 - DOS, 294
 - Double abstractions, 20, 476
 - DP
 - Development Beyond Target, 55
 - industry growth, 293
 - Dragons, 133
 - DSM, 435
 - Duality, system, 234
 - DVT, 370
 - Dynamic
 - reliability, 202
 - storage management, 435, 437
 - Dynamics, program evolution, 14, 15
- E**
- Eastern Europe, 293
 - E-Class, systems analysis, 444
 - Ecological interrelationships, 157
 - Economic
 - concept, 469
 - factors, 273
 - sciences, 376
 - systems, 418
 - Economists, 237
 - Economy, national, 393
 - Education, 80, 81
 - Educational systems, 159, 162
 - knowledge ν understanding, 158
 - Effective productivity increase, 486
 - Effects
 - feedback, 260, 262
 - inertial, 260, 261
 - momentum, 260, 261
 - Effort reduction, 488
 - Eight Queens, 397
 - inventory control, 277
 - Electorate, 154
 - Electronic Industries Association, 498
 - Electronic Switching Systems Division, 1
 - Embryonic methodologies, 63, 65
 - Empirical approaches, 337
 - Encapsulation, 290
 - Encyclopedia of Ignorance, 6, 237
 - End-user community, 294
 - Energy sources, 156
 - Engineering, 14, 285
 - blueprints, 363
 - technology, 292
 - Engineers, 52, 237, 289
 - Entropy, 118, 129, 143, 145, 147, 169, 304, 333, 336, 342
 - Environment
 - computer, 91
 - data, 92
 - Environmental support, 460
 - Epidemiology, 238
 - E-program, 397, 402, 404, 405, 413, 456
 - A-type formation, 405
 - basic loop, 457
 - feedback loop, 402
 - full process, 457

- model of a model, 462
 - pressure for continuous change, 403
 - properties, 456
 - P union, 405
 - Equipment, peripheral, 92
 - Error, 169, 185, 186, 226, 284
 - ν fault, 185
 - ν defect, 185
 - rate models, 434
 - repair, 195
 - residual, 283
 - types, 99
 - Errors, law of numbers, 93
 - test programs, 96
 - ES, 370
 - Estimation step, 340
 - E-type
 - application development, 27
 - program, 10, 16, 19, 23, 27, 478
 - European space agency, 468
 - Evil, 315
 - Evolution, 60, 408, 492
 - artificial system, successive generation, 3
 - complete paradigm, 29
 - controlling, 221
 - current programming process, 15
 - decimal, sub-releases, 19
 - dynamics, 3, 37, 144, 202, 339
 - application of studies, 217
 - feedback controlled system, 16
 - generation, 17
 - historical summary, 9
 - ideal process, 15, 19
 - concept role, 27
 - intrinsic, 13
 - iteration, 23
 - levels, 16, 36
 - life cycles
 - applied dynamics, 418
 - implications, 421
 - case study, system X, 426
 - problem, 428
 - process dynamics, 430
 - system characteristics, 426
 - laws, 408
 - dynamics, 410, 413
 - measure, 411
 - nature, 417
 - models, 397
 - correctness, 405
 - E-program, 402
 - P-program, 399
 - S-programs, 397
 - problem, 393
 - programming, 393
 - paradigm, 32
 - practical process, 26
 - preliminary design paradigm, 32
 - process, 13, 19, 34, 36
 - program, 356
 - software engineering impact, 201
 - data, 203
 - development work distribution, 211
 - dynamic studies application, 217
 - effective work rate, 208
 - system size, 205
 - system T, 216
 - software process support, 37
 - SPE classification, 9–11
 - step paradigm, 28
 - successive releases, 18
 - systems, 14, 15
 - time-dependent behavior, 171
 - Executable model, 466, 478
 - Execution dynamics, 323, 356
 - Existence rule, 413
 - Exogenous inputs and pressures, 376, 378
 - Expenditure, 39, 40
 - maintenance ν development, 395
 - projected programming, 293
 - Experience, 285
 - programming system dynamics, 105
 - decay, 107
 - Expert systems, 158, 496
 - Explicit program object, 251
 - Exponential growth, 126, 168
 - Externalization, 278, 287
 - External
 - complexity, 254
 - structure, 370
- F**
- Face validity, 347
 - Factors countering increasing growth rate, 175
 - Factual historic record, 461
 - Failure avoidance, 488
 - Fallacy of basic appearances, 301
 - Family
 - configurable model, 234
 - first, 232
 - highly evolutionary, 230
 - real life, 231
 - restricted, 231
 - Fault, 99, 112, 119, 123, 129, 302, 303, 446, 447

- Fault (*continued*)
 absence, 197, 452
 activity sequence to repair, 101
 classes, 188, 190
 corrected during maintenance, 395
 correction, 171
 decay, 190
 ν defect, 185
 definition, 100
 detection, 113
 early discovery, 486
 elimination-to-residue ratio, 190
 ν error, 185
 expected, 434
 field-discovered, 325
 free state, 190, 250, 321
 qualitative interpretation, 191
 generation, 187, 188, 189
 network, 189
 pattern, 188
 penetration, model of, 187, 189
 primitive model, 187
 program, 185
 rate, 145, 148, 151, 385
 removal of symptoms, 179
 removed, 187, 189
 repair, 436
 reports, 376
 residual, 187, 188
 sources, 100
 system repair, 169
 Federal aid, 154
 Feedback, 309
 two-loop, nested, 194
 organizational level, 384
 positive loops, 262
 structure, intrinsic non-linear, 265
 system, 211
 consequences, 377
 Feselen, 136
 FGCS, 494-498
 Fifth generation technology, *see*
 Technology
 Fifth law, 260, 262, 270, 272, 341, 420, 425
 First law, 88, 178, 250, 342
 Fission effect, F, 175, 176
 Fitness population, 86
 Five dimensional array, 487
 Five laws of program evolution, 381
 Fixes, on site, 440
 Forecasting, 168
 Forecasts, 484
 Foreign languages, 287
 Formal language, 71
 Formal specification language, 76, 198
 Format, logically universal, 225
 FORTRAN, 62, 72, 94, 394
 Fourth law, 260, 262, 266, 341, 420, 425
 Fraction
 active system, 213
 modular function of age, 214
 Free energy, 119
 Full step paradigm, 33
 Function f, 223-225, 229
 Function, new, 108, 109
 Functional components, 251
 Fundamental law of program evolution,
 381, 412, 413
- G**
- G, 127, 129, 147
 structure, 127
 system size measure, 117, 118
 Game theory, 245
 Garmisch report, 2
 General system theory, 300
 Generic modules, 226, 227
 Genetics, 238
 Geometric decay, 103
 Goedel's theorem, 243
 GOTO, 325
 GREENPRINT, 363
 Growth, 151
 alternating with cleanup, 354
 control, 53, 54
 data, 341
 dynamics, 99, 118, 123, 126, 129, 131
 incremental, 430
 law of statistically smooth cycles, 145
 limits, 133, 139, 140, 143, 144, 150, 151,
 153, 157, 158, 163
 normalized component per release, 124
 normalized net instruction, 125
 rate, 1, 70, 183, 341, 348, 430
 trends, 166, 168
- H**
- Handle
 expected ν observed, function of age,
 216
 rates, 172, 178, 179, 180, 181, 184, 211,
 213, 215
 release interval, 180, 181
 Hardware
 repair, 358
 support, 436
 systems, 395

technology, 203
 Heisenberg's principle, 244
 High-level languages, 72, 73, 177, 209, 213, 420
 Hooton, D. R., 219
 Horizontal verification, 31, 484

I

IBB

TSS/360 system, 1
 Yorktown Heights Research Laboratories, 149
 IBM, 1-3, 39-41, 49, 51-56, 66, 71, 72, 74, 79-83, 184, 185, 200, 219, 294, 363
 Clear-Caster project, 67, 70
 cost control, 53
 director of research, 2
 maintenance organization, 360
 manpower growth, 42
 research division, 64, 81
 system development division, 39, 40, 55, 56, 57
 IBR Research Division, 221
 Ideal process, 15, 16, 19, 21, 25, 37, 472
 cycle, 20
 step structure, 482
 IFIP working group, 3
 IKBS, 4, 495
 Imperial College, 7, 162, 200
 Imperial Software Technology, Ltd., 485
 IMP executive, 1
 Index, 232, 233, 345
 Industrial
 assembly line, 317
 revolution, 82
 Information, 355, 499
 theory, 238, 332, 336
 Input, 28, 175, 376, 398, 407
 Interconnection topology, 129
 Interface definitions, 236, 251, 326, 472
 Internal complexity, 254
 Inter-parameter relationships, 149
 Interplay effect, 174
 Inter-release interval, 172
 Interrupts, 474
 Intra-system communication, 313
 Invariance, 177, 183, 266, 268
 Invariant
 statistically, 384
 work rate, 381, 412
 IPSE, 117, 485
 IPSES, 116

IR factors, 431
 IST, 485
 Iteration, 23, 171, 173
 ideal process, 25
 reduction, 488
 ITS
 delay, 431
 facility, 428
 low interconnection ratio, 433
 release, 434, 435, 436

J

Japan, 51, 74
 Japanese fifth generation plan, 491

K

Knowledge, 166, 277, 491
 v understanding, 158

L

Language(s), 69, 71, 72, 74, 75, 77, 165, 296, 394, 423, 445, 473
 Largeness, 248, 249, 261, 413
 concept, 397
 definition, 397
 programs, 253, 258, 262, 290, 326, 356, 375, 413, 418
 software, 279, 284
 systems, 6, 273, 280, 289, 313, 321, 471
 Law of conservation of familiarity, 381, 412-413
 Law of continuity of change, 143, 250, 252, 412, 413, 380
 Law of evolutionary potential, 87, 97
 Law of increasing complexity, 253, 381, 412-413
 Law of increasing entropy, 143
 Law of incremental growth limits, 270, 272, 384, *see also* Law of conservation of familiarity
 Law of invariant work rate, 266, 269, 384, *see also* Law of organizational stability
 Law of organizational stability, 384, 412-413
 Laws of program evolution, 169-170, 247-257, 260, 263, 266, 269, 270, 272, 408, 412
 Law of statistically regular growth, 257, 260, 263
 Law of statistically smooth growth, 143,

- Law of statistically smooth growth (*continued*)
 170, 383, *see also* Fundamental law
 of large program evolution
- Law of supply and demand, 376
- Lead times, programming systems, 49
- Life cycle, 154, 174, 368, 393, 438
 concept, 469
 costs, 252, 301, 304, 312
 distribution, 396
 history, 264
 management, 252, 438
 methodology research process, 441
 phases, 444
 significance concept, 442
 software, 372
- Life-span system, 17
- Localization of change, 488
- Logically universal format, 225
- London model, 151
- London University, Imperial College of
 Science and Technology, 4
- Long life system, production, 476
- LSI, 48, 72, 75, 77, 273
- LST, 486
- M**
- Machine aided design, 368
- Machine errors, 89
- Macro-model, 100, 126, 131, 147
 deductions, 116
 dynamic, 148
- Magnetic tape error routines, 89
- Maintenance, 12, 275, 446
 life cycle oriented, 257
 phase, 461
 process, 383
 v redesign, 358
 staff requirements, 279
 teams, 179
- Management, 77, 79, 108, 155, 156, 193,
 289, 350, 351, 375, 421, 424, 458
- Manufacturing, 356
 process, 321, 492
- Marketing
 considerations, 433
 requirements, 359
- Mathematical model
 equations, 345
 mathematics, 292
 parameter trajectories, 346, 348
 proof standards, 405
 software evolution, 339, 340, 343, 344,
 346, 347, 348, 350, 352, 354
 variables, 345, 346
- Matrix
 calculations, 89
 connectivity, 366
 two-dimensional, 365
- MEL, 370
- Meta-dynamics of systems in maintenance
 and growth, 99, *see also* Program-
 ming system dynamics
- Meta-knowledge level, 496
- Meta-stable plateau
 evolutionary process, 421
- Metasystem, 141, 143, 147, 168, 170, 418
 dynamics, 410
 invariance, 177
 organization, 411
- Methodology, 59, 62, 63, 65, 82, 148, 150,
 180
- Micro-code, 319
- Micro-model, 100, 111, 117, 118, 129
- Micro-processor, 322, 328, 329, 393, 407
 distributed systems, 418
 technologies, 273
- Micro-theory, 118
- Minimax strategy, 246
- Mission concept, 64, 65
- Modula 2, 480
- Modules, 172, 211
 changed, 432
 fraction, 259, 260
 function of age, 214
 growth, 176
 overlapping, 326
 per day, 168
 program measure, 343
 specification uncertainty, 408
 versions, 227, 228
- Multi-dimensional space, 469
- Multi-level model, 64, 65
- Multiloop feedback, 177
- Multiprocessing, trend, 96
- Multivariate regression, 168
- Mutations, 27
 frequency, 410
- Mutual adaptation relationships, 91
- MVS, 223
- N**
- NASA project engineering, 65
- Nassi-Sneiderman diagram, 363
- Natural language, 71
- Natural selection, 85, 87, 90, 91, 95
- Nested representations, 363
- Node, 188, 190, 336
- Non-linear growth, 168

- Non-universal constructs, 231
- Notation, unambiguous, 483
- N-point average, 128, 130
- NS diagram, 363

- O**
- Oil royalty accounting, 89, 92
- Open loop control, 55
- OMEGA, 294
- Optimal structure management, 353
- Optimization, 322, 323
- Optimum performance, 323
- Organizations, 415
 - human, 277
- Original implementation, 101, 102
- Orthogonal, 482
 - natural process functions, 471
- OS/360, 3, 5, 44, 166, 167, 170, 172, 175, 177, 182, 222, 294, 357
 - complexity, 47
 - growth, 81
 - average rates, 46
 - table, 45
- OS/370, 222, 294, 357
 - selectable units, 223
 - VTAM component, 362
- Oscillations
 - damped, 351
 - growth rate, 341
- Output
 - accuracy, 407
 - growth limits, 152
- Output model, 28, 242
- Overlapping modules, 226

- P**
- Perceived complexity, 386, 412
- Peripheral equipment, 92
- Permitted
 - actual systems, 228, 230, 231
 - description, 229
 - configurations, 222, 223, 224, 230
- Phase-products, 461
- Phase review, 62
- Phenomena, 166
 - discussion, 411
 - interpretation, 260
 - law basis, 418
 - model, 479, 480
 - programming, 418
- Phenomenology, 291
 - increased, 389
 - science transition, 391
- Phosphorous images, green, 363
- PL/1, 72, 74, 76, 363, 368, 394
- PL/1-like macros, 74
- Positive feedback loops, 262
- Potentially interacting modules, 48
- P-programs, 397, 399, 400, 401, 405, 413, 454
 - A type formation, 405
 - E union, 405
 - process, 455
- PREVAIL-ON, 232, 233
- Problems, 498
 - areas, 492
 - complex, 331
 - parts number, 5
 - requirements, 492
 - statement, 475
- Procedure protocol, 420
- Product dynamics process, 247
- Productivity, 69, 127, 150, 469, 471, 485
 - increased, extending maintenance period, 477
 - programming, 68, 80, 281, 282, 312, 393, 409, 459
 - activity rate, 266
 - development support, 57
 - evolution dynamics, 170
- Program, 91, 92, 165, 225, 314, 346, 406, 409, 444, 454, 460, 465
 - complexity, 245
 - development model, *see* Development model
 - evolution, *see* Evolution program
 - execution rate, 409
 - growth dynamics, 5, 340, *see also* Program evolution dynamics
 - maintenance, 165, 241, 395
 - methodologies, 395, 419
 - models, 397, 399, 402, 476
 - structures, 406
 - systems, 313
 - temporary fixes, 222, 223, 225, 226, 341
 - verification, 452, 454
- Programmer, 41, 52, 80, 317
 - average, 389-390
 - productivity, 51, 52, 344
 - projected population, 293
- Programming, 39, 48, 49, 51, 52, 165, 167, 470
 - growth, 39, 54, 55, 56
 - cost, 39, 48, 49, 51, 52
 - systems, 41

Programming (*continued*)

- language, 71, 74
 - advantages, 72
 - aspects, 71
 - BSL, 74
 - formal specification, 76
 - level, 71
 - linguistic wealth, 75
 - present practice, 71
- management, 77
 - dynamic structure, 79
- methodology, 59, 60, 62, 63, 65
- model, 473
- tools, 66
 - development, 67-69
- productivity
 - alternative process decomposition, 473
 - computational model, 479
 - current process, 471, 485
 - implications, 485
 - measures, 469
 - model, 473, 478-481
 - primitive decomposition, 475
 - third level, 478
 - transformations sequence, 476
- Program-system-models as common starting point, 76
- Progressive activity, 149, 151, 152, 154, 155, 163
 - v* anti-regressive work, 339, 342
 - city life, 153
 - complementarity, anti-regressive, 150
 - conflict and balance, 154
 - education, 158
- Project
 - constraints, 383
 - IMP 1, phase three, 2
 - planning, 459
 - statistics, 167
 - successful, 263
 - team, 343
- Prolog program, 493, 496
- Propriety systems, 459
- PSE, 458
 - data base, 467
 - design, 465
 - executable models, 464
 - properties, 468
 - tool supported models, 466
- Pseudo-hierarchical structure, 408
- PTF, 222, 223, 226, 231, 232, 235, 341, *see also* Program, temporary fixes
- P-type program, 10, 478

Q

- Quadratic form, 178
- Quantification, complexity, 335
- Quantum
 - jump, 314
 - physics, 244
- QUEUE, 370
- Quicksand, 396

R

- R19, 434
- R20, 429, 435
 - change rate needs, 434
 - plan analysis, 431, 436
- R21, 433
 - experimental release, 438
- R22, 438
 - general release, 435
- Radio links, 402
- Random transient events, 410
- R and D, 82, 492
- Reachability measure, 335
- Real-life, 194, 196
 - family, 231
 - use patterns, 282
- Real world, 60, 244, 291, 377, 478
 - approximations, 399
 - context, 400
 - phenomena, 454, 479
 - problem solution, 399
 - problem-systems complexity, 60
- Recreated *v* dynamically changed program, 412
- Recycling, 157
- Redesign, 280, 304
 - v* maintenance activity, 358
 - staff requirements, 279
- Refamiliarization difficulty, 389
- Refinement, defined, 32
- Regression, 166
- Relational description of permitted actual systems, 229
- Relationships
 - inter-parameter, 149
 - nature, 174
- Relativity, program complexity, 254
- Relays, 89
- Release, 18, 44, 101, 116, 123, 141, 147, 167, 168, 171, 174, 175, 176, 177, 180, 188, 211, 217, 283, 295, 341, 353, 385, 386, 387, 417, 421, 434, 446

- budget allocation, 112
 - consecutive, 166, 294
 - continuous, 440
 - cyclic growth, 176
 - degree overlap, 180
 - extreme instance, 440
 - internal, 180, 182, 211
 - interval, 105, 111, 115, 183, 201, 272
 - modules handled, 213, 215
 - number, 146, 181, 184
 - planning two, 352
 - OS/360-370, 357
 - period, 433
 - overlap, 211
 - rate, 180
 - saturation point, 357
 - sequence, 439, 440
 - numbers, 168, 171, 172, 175, 179, 260
 - successive, 126, 353, 381
 - system, 420, 428
 - target date, 174
 - timing, 352
 - two, 353, 385
 - Release-cycle period, 258
 - Relevance
 - output, 456
 - program, 406
 - Reliability
 - dynamic management structure, 79
 - multi-function program, 202
 - Repair, 129, 148, 279, 318
 - Repository, 32
 - Representational model, 461, 476, 478
 - programming process, 476
 - Representation, nested, 363
 - Requirements, 492
 - contradictory, 324
 - development, 466
 - model, 478
 - Research and Development, 82, 219, 492
 - Research in operating systems, 360
 - Restoration system, 395
 - Restricted sub-sets, 77
 - Review, phase, 62
 - Reward, complexity reduction, 282
 - Right profile module, 370
 - RJE
 - addition, 433
 - facility, 435
 - Riordon's complexity function, 343
 - RSN, 171, 175, *see also* Release sequence numbers
- S
- Saturation point, 357
 - Scheduler, 314
 - Science Research Council, 220
 - Scope rules, 420
 - Scoping, program, 362, 363
 - S-curve, 180
 - Secondary repairs, 112
 - Selective breeding, 87
 - Self-documentation, 319
 - Self-stabilizing feedback system, 175, 267
 - Sequence of activities, 440
 - Sequence transformations model, 476, 477
 - Serial growth trend, particular attribute, 168
 - Short-term cyclic effects, 174
 - Signal-flow-graph, 195
 - Simulators, 276
 - pilot training, 466
 - Sledge hammers, 409
 - Slipped release, 176
 - Smooth data, 126, 141
 - Socio-economic systems, 14, 261, 301, 377
 - Software
 - change, 280
 - complexity
 - categories, 332
 - examples, 335
 - meaning, 343
 - measures, 173
 - summary, 337
 - creation, 383
 - engineering, 7, 65, 170, 201, 218, 254, 273, 296, 312, 321, 327, 329, 375, 377, 419, 439, 448, 459, 460, 473
 - applied, 291
 - community, 327, 360
 - concept, 3
 - discipline development, 473
 - industry, 439
 - measurement, 291
 - Yorktown, 360
 - factory, 224
 - families, 221, 227
 - industry, 417
 - interfaces, 328
 - maintenance, 280
 - manager responsibility, 423
 - normalized cost, 276
 - organization, 299
 - physics, 173
 - process

- Software (*continued*)
 - knowledge, 315
 - technology, 498
 - repair, 358
 - services, 316
 - system, 301
 - development, 474
 - engineering, 472
 - nature, 324
 - properties, 470
 - technology, *see* Technology
 - tools, 459
- S-shaped curve, 417
- Solar energy, 157
- Solution model, 475
 - defined, 475
 - identification, 477
- Sorting, 459
- Spanning tree, 336
- SPE, 452
 - program classification, 405, 407
- Speciation, 94
- Specification, 326, 420
 - authoritative, 453
 - code, 464
 - error, 407
 - formal, 76
 - model, 479, 483
 - process, 445
 - system module, 420
- S-program, 398, 400, 407
 - A-program elements, 406
 - code inspectors testing, 452
 - correctness, 405, 406
 - full process, 455
 - specification, 397
 - verification, 454
- Stabilization time, 272
- Staffing problems, 275, 287
- Stateplane trajectories over two releases, 353
- Steady state, 190
- Step
 - paradigm, 28, 482
 - code, 28, 30
 - structure, 482
- Stepwise refinement, 474
 - process, 489
- Stochastic, 170, 174, 178
 - components, 180, 181, 270
 - locally, 170, 306
 - modelled, 248
 - terms, 182
 - variations, 140, 167
 - growth trend measures, 144
- Stock control, 402
 - system, 315
- Stoneman
 - programming support, 423
 - proposal, 459
- Street modification, 355
- Strong typing, 420
- S-type program, 9, 23
- SU, 223
- Subactivities, 111
- Subdivision of labor principle, 277
- Sub-releases, 19
- Sub-step paradigm, 32
- Sun, 156
- Synchronization, 283
- Syntax-oriented documentation, 65
- SYSGEN program, 235
- System
 - age, 182, 309
 - approach, 165
 - architecture, 258
 - behavior optimization problem, 322
 - characteristics, 414, 416
 - complexity measure, 211
 - configuration, 447
 - continuous dispatcher, 402
 - delivery, 282
 - design, structure model, 480
 - evolution observables, 172
 - families model, 221, 225, 228, 231
 - fission, 391
 - generator (SYNGEN), 235
 - integration, 171
 - management, 309
 - modification program (SMP), 235
 - module parameters, 172
 - performance execution dynamics, 322
 - pollution, 325
 - release, 222
 - restoration, 395
 - size, 205
 - age, 206
 - declining growth rate, 181
 - history, 205
 - indicators, 295
 - parameters, 172, 205
 - structuring process, 480
 - technology, *see* Technology
 - test, 446
 - totality, 387
 - tuning, 359

- ultimate, 475
 - Systemization techniques, 496
 - System-relative stochastic influences, 174
 - Systems-programming language, 74, 75
- T**
- Table-driven design, 209
 - Target system, 471
 - Technology, fifth generation, systems
 - and software
 - problem areas, 492
 - complexity, 494
 - correctness, 496
 - cost, 497
 - evaluation, 493
 - identifiers, 492
 - requirements, 492
 - responsiveness, 497
 - understanding, 494
 - VLSI, 492
 - Technology-oriented programming
 - methodology, 62
 - Telecommunications support, 435
 - Test, 165, 419, 464
 - designers, 177, 302
 - fit, 173
 - programs, 96
 - run, 283
 - Tester, 386
 - Theory, program evolution, 27
 - Thermodynamics, second law, 129, 238, 255, 304, 382, 413
 - THE system, 474
 - construction method, 475
 - partitioning, 474
 - Three-dimensional indexing scheme, 233
 - Three-release cycle, 180
 - Time, 129, 166, 193, 304
 - behavior, 151
 - cost per unit, 193
 - dependent relationship, 149
 - devoted, software maintenance, 303
 - integral, 108
 - interval, 88, 123, 346
 - over-runs, 385
 - series models, 141, 168
 - Tools, 66, 242, 250, 256, 263, 269, 286, 471, 493, 498
 - accessible, 460
 - conceptual, 354
 - control, 383, 396
 - integrated families, 67
 - languages, 394
 - life cycle management, 377
 - maintenance, 395
 - management, 247
 - modifying, 360
 - new needs, VTAM, 365
 - planning, 383
 - refine existing, 361
 - support, 460
 - Tool-oriented processes, 66
 - Tool-supported total process, 464
 - Top-down analysis, 472
 - Total-process-oriented methodology, 62, 68, 71
 - formal specification step, 76
 - Total-system oriented methodology, 80
 - Tower of Babel, 287
 - Traditional indicators, 292
 - Trajectories, stateplane, 353
 - Transformation
 - compound, 474
 - procedures, 497
 - sequence, 476
 - simple, 474
 - Transformational-step paradigm, 483
 - Transportation system, 301
 - Trends, 166, 178
 - cyclically self-regulating, 257
 - long-range, 170, 306
 - long-term, 174, 175
 - non-linear, 217
 - oscillatory, 217
 - present, and work rates, 180
 - projected, 292
 - systems, 140, 141
 - T-region threshold, 390
 - TRUE, 223, 230
 - TSS-360, 69, 70
 - Turn-around time, 239
 - Two-dimensional indexing scheme, 232
 - Two-level hierarchy, 314
 - Type rules, 370
- U**
- U, constant, 349, 350
 - Ultimate system, 475
 - Unbundling, 53
 - Uncertainty, 243
 - complexity measure, 342
 - coping technology, 495
 - develop, 441
 - evolution cause, 408
 - principle, 244
 - Understanding, 166, 242, 493, 494

Understanding (*continued*)
 anti-regressive activity, 159
 attributes, 159
 demonstration, 160
 insight, 159
 need, 161
 orientation consequences, 161
 total, human, 494

Unlimited growth potential, 182

Union controls, 384

Unit change, 209, 396

Unit cost, 282, 396

United States, 39, 43, 161, 355, 373, 497
 computer and programmer census, 293
 GNP, 393
 1977 programming expenditure, 393, 395

Univariate regression, 168

Unplanned release, 176

USSR, 293

V

Validation, 31, 35, 66, 484
 judgmental activity, 464
 mathematical model, 347
 based on measurement, 464
 methods, 476
 process, 454

Value judgments, 399

Variation, 86
 program statistics, 299

Variety, 248, 291, 300
 imperfection generated, 302
 large program property, 375
 perfection generated, 300
 uncertainty related, 333

VDU, 359
 specific type, 397
 terminals, 363

Verification, 68, 76, 302, 464
 consistency and completeness, 464
 methods, 476
 partial, 496
 simple, 496
 vertical, 31, 483, 484

Vertical verification, 31, 483
 transformation process, 484

Victoria and Albert Museum, 133

Viewpoint model, 438

Virgin territory, 285

VLSI, 4, 13, 491

hardware, 492

VonNeuman, 496

software, 493

techniques, 496

VTAM, 362, 372

existing system, 362

experimental redesign, 369

function after redesign, 370

research vehicle, 362

subcomponents, 365

W

Walkthrough, 446

Waste, 276

Waste collection, 157

Waterfall model derivatives, 473

Weather prediction, 399

Welding guns, 409

Western world, software community, 395

Wirth, 32

Work curve, 108

budget controlled, 108, 109

Work force reduction, 197

Work-input rate, 296

Work-output

average rate, 253

constant rate, 146

Work rate, 180, 209, 266, 268, 270, 425

function of age, 212

declining, 185

invariance, global, 268

peak, 430

present trends, 180

pressures, 432

process dynamics, 430

second operating system, 184

Worst case, 354

X

XPL/I, 368

Y

Yorktown model, 147

Z

Zero tolerance, 422

Zoom lens, 461

A.P.I.C. Studies in Data Processing
General Editors: Fraser Duncan and M. J. R. Shave

14. Software Engineering
R. J. Perrott
15. Computer Architecture: A Structured Approach
R. W. Doran
16. Logic Programming
Edited by K. L. Clark and S.-A. Tärnlund
17. Fortran Optimization*
Michael Metcalf
18. Multi-microprocessor Systems
Y. Paker
19. Introduction to the Graphical Kernel System—GKS
F. R. A. Hopgood, D. A. Duce, J. R. Gallop and D. C. Sutcliffe
20. Distributed Computing
Edited by Fred B. Chambers, David A. Duce and Gillian P. Jones
21. Introduction to Logic Programming
Christopher John Hogger
22. Lucid, the Dataflow Programming Language
William W. Wadge and Edward A. Ashcroft
23. Foundations of Programming
Jacques Arsac
24. Prolog for Programmers
Feliks Kluźniak and Stanislaw Szpakowicz
25. Fortran Optimization, Revised Edition
Michael Metcalf
26. PULSE: An Ada-based Distributed Operating System
D. Keeffe, G. M. Tomlinson, I. C. Wand and A. J. Wellings
27. Program Evolution. Processes of Software Change
M. M. Lehman and L. A. Belady

*Out of print.