Software Engineering

INFOTECH State of the Art Report 11

Software Engineering

International Computer
State of the Art Report

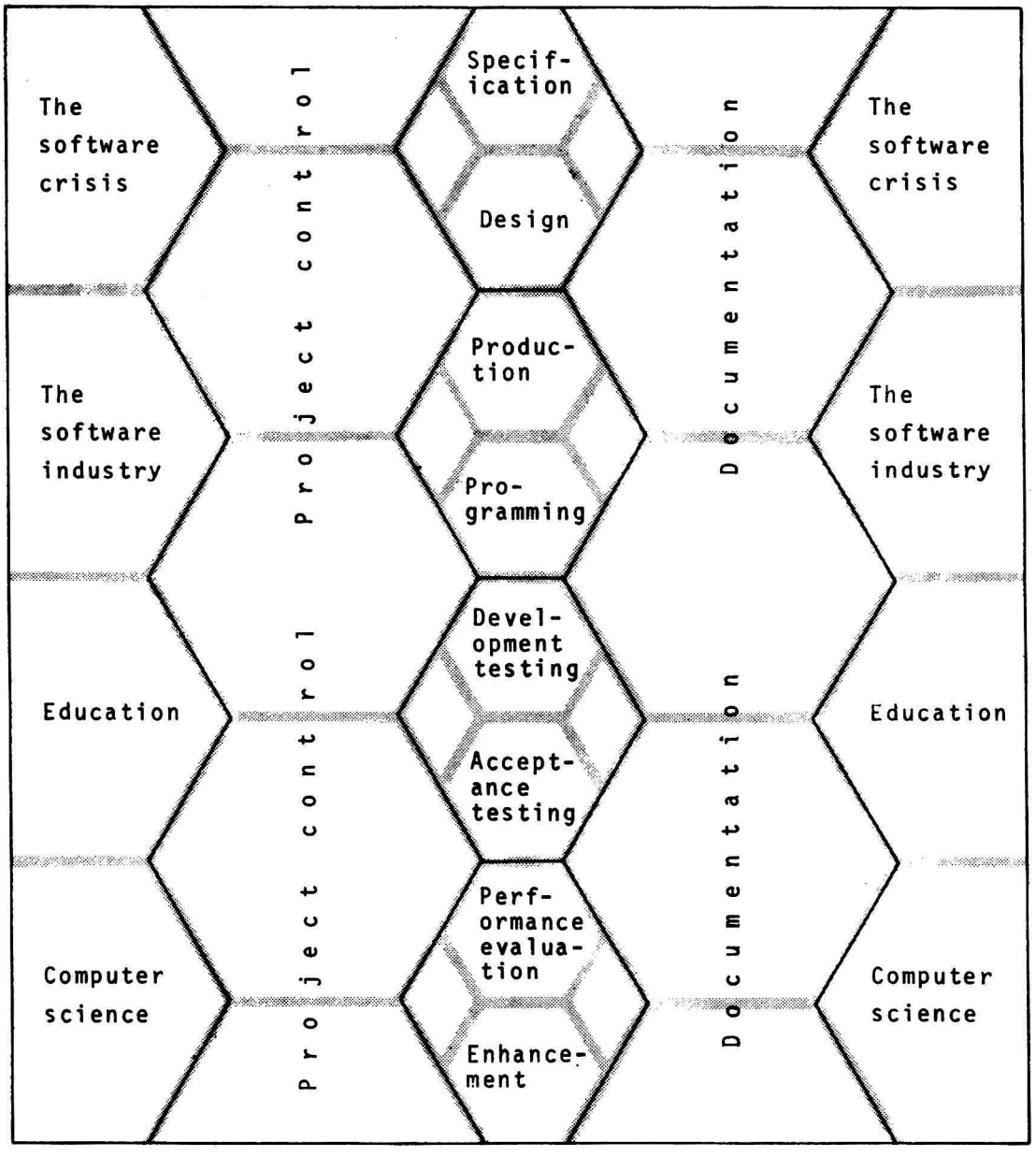
INFOTECH Information Limited Maidenhead Berkshire England

UDC 681.3

Dewey 658.505

ISBN 8553-9100-6

Library of Congress Catalog Card Number 72-189688


The software crisis
The software industry
Education
Computer science
Project control
Project control
Specif-ication
Design
Produc-tion
Pro-gramming
Devel-opment testing
Accept-ance testing
Perf-ormance evalua-tion
Enhance-ment
Documentation
Documentation
The software crisis
The software industry
Education
Computer science

CONTENTS

FOREWORD

The term 'software engineering' is not clearly defined; most people who use it have their own, usually unique impression of what they mean by it. Many people, of course, disagree even over the meaning of the word 'software'. Nevertheless, this does not detract from the value of the analogy between building and maintaining software and the activities associated with traditional fields of engineering, which derives from two main sources. First, simply that software is specified, designed, produced, tested, and maintained makes the analogy of value. Second, the analogy is of particular value since software has traditionally embodied the most difficult and least understood aspects of system implementations and so the building of software used to be, and in many instances still is, approached more as a form of art or abstract science than an applied science: the engineering analogy brings with it a necessary change in attitude.

Although there is a consensus as to what the qualities of good software are, the design of software, and in particular large scale software, is a complicated and difficult task, with almost as many approaches as there are principal software designers. Most of these approaches do however have at least one common facet in that they are all in some sense hierarchic. The use of hierarchic structures has been a prime influence in all scientific fields since before Aristotle. The field of software is no exception: hierarchic design approaches have proved so far the most convenient way of simplifying the connection patterns in complex software. Of course, not every software designer draws the same implications from use of the hierarchic structure, or makes use of such structures with the same degree of explicitness; for example, compare the approaches of Woodger, Dijkstra, Pyle, Jackson, and Morris in the State of the

Art Reports entitled High Level Languages, Software Engineering, Application Technique, and The Fourth Generation. However, some seeds of doubt regarding the use of hierarchies are being sown. Some people see rigid use of the tree structure as a limitation, imposed from within ourselves by our perception of things. Also, there is even a growing feeling amongst a small minority of people that hierarchic structures are not as universally applicable as we have tended to think in the past and that we should be searching for alternative, horizontal structures.

The production of software is probably the area where the engineering analogy has so far had the most effect. The effect here has two main aspects. First, the engineering analogy has led to the realization that, since the equivalent of production in traditional engineering fields is straightforward replication in software, what is referred to as software production is really analogous to building a prototype. This obviously changes what one can justly expect of the first version of a piece of software. Also, it has important implications for management in that management of a software project is more research and development management than production management in the traditional sense of the phrase. Second, developments in the last decade have led to the production of software becoming an activity of an increasingly industrialized nature. With talk of software components, software engineer's workshops, software factories, and with the developments in what is now referred to as automatic programming, software production techniques have evolved rapidly and the software programmer now has a wealth of tools and aids, at least potentially, at his disposal.

In testing also such aids are available, although here the overall approaches used have come under severe criticism from some quarters. The most famous attack on the traditional debugging approach to software testing has come from Dijkstra, orignator of the discipline known as structured programming and proposer of correctness proving for programs. In the words of Micheal Jackson, himself greatly influenced by the thinking of Dijkstra, the many debugging aids available "are really just cosmetics for a dying program", since software systems are not designed so that they are testable; if software is not structured in such a way that systematic testing at the various levels of detail of the software, rather than testing complete programs, can be accomplished then the number of test cases necessary rises to such an extent that fully effective testing becomes impossible. The criticisms made by Dijkstra and his followers certainly are valid when one looks at the unrealiability of much

existing software in use today, and structured programming is becoming more and more recognized as a laudable, and practical, discipline to follow; however, the potentially profound effect of correctness proving in programs has yet to be felt, and many feel that it never will, at least with any significance for the majority of users. Others, although they cannot see correctness proving as generally applicable in practice at the present state of the art, take the view that such an approach will ultimately prove indispensable.

The maintenance of software is an area of particular concern simply because, with the average lifespan of pieces of software tending always to lengthen, maintenance can account for as much as 70% of total software cost. The main problem here is that maintenance can be made an intolerably difficult task if the preceding design approach ignored testing and maintenance needs. If maintenance is viewed as an enhancement activity then the situation is improved since this involves iteration through the whole cycle of software building.

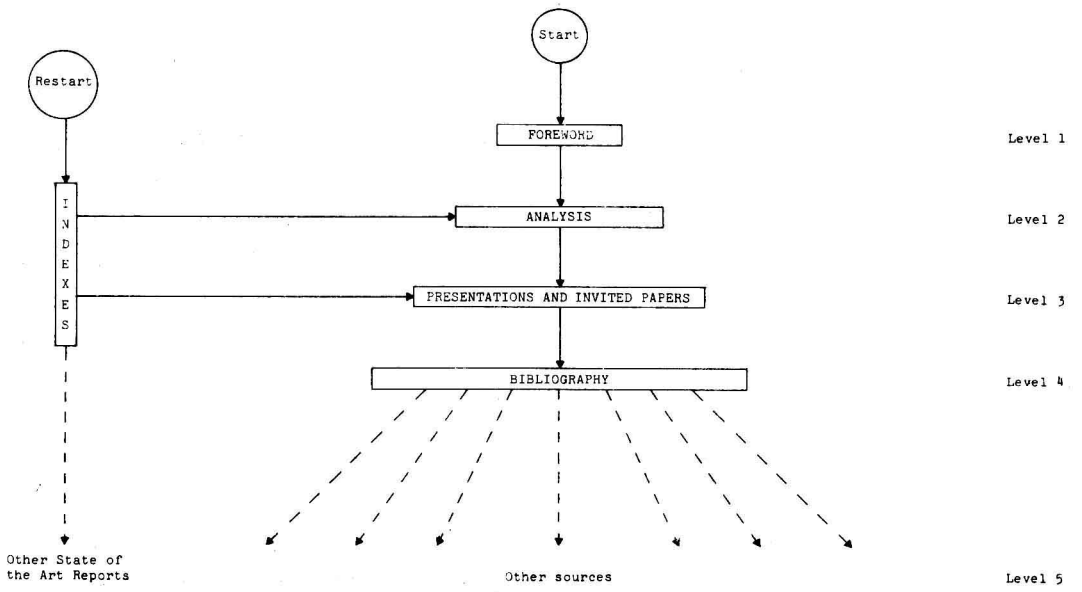
The extended maintenance/enhancement phase of the software life cycle has increasingly led to the desire to evaluate system performance, both for tuning and planning purposes. In a climate where many users feel that they are living with inefficient software, interest in performance evaluation has spread rapidly and a number of system measurement tools have come onto the market. Many great savings have been achieved through use of such tools, although there are very real dangers in choosing what to measure and in interpreting measurement results.

Apart from the main activities of specification, design, production, testing, maintenance, development, and tuning, the engineering analogy applied to the field of software has broader, and possibly more profound, implications in the areas of education and industry structure. It is probably in these two areas where any changes are likely to have the greatest effect on the business of building software.

C Boon: Editor

REPORT STRUCTURE

Information is a function of time as well as data. Therefore, just as considerable thought has been given to selecting the most important contributors and significant contributions to this Report, so has great attention been paid to enabling the reader to find his way round the Report quickly and easily. The diagram below illustrates the paths through the Report.



The editorial content of the Foreword is designed to set the scene for the Report and should be read for a broad, summary view of the subject. The Analysis, which bears the Report title, brings together the essential material of the subject of the Report, structured by topics: *it is strongly recommended that everybody read the Analysis*. From topics within the Analysis, pointers create a path through other material on the same or related topics in the rest of the Report. The Analysis has been designed to convey, in as short a space as is compatible with rendering the content easily assimilable, the principal issues in the subject under discussion.

The Presentations and Invited Papers together form a further level of information. These may be read in their entirety or a selection of passages on related topics may be obtained via pointers from the Analysis.

The Bibliography, which is arrived at via pointers from the previous two levels of information, contains an introduction as well as references and abstracts. The introduction will help to ensure that the most suitable reference is selected before any effort is made to obtain the corresponding document.

The Bibliography, of course, leads to a further level of information, outside the Report.

To refer back to material within the Report, it is recommended that a search be made of the indexes, which contain pointers to the Analysis and to the Presentations and Invited Papers. Once a suitable reference inside this material has been found, pointers to related material will be found as before. To provide further assistance, the normal subject index is supplemented by a contributor index and a cumulative subject index containing back pointers to previous State of the Art Reports.

To find a specific paper, presentation, or section of the Analysis, reference may simply be made to the Contents list.

EDITORIAL CONVENTIONS

To avoid any misconceptions, the following editorial conventions concerning the Analysis should be noted.

1 Material in italics is attributable to the editor.

2 Other material is attributable to the person whose name precedes it. It should be noted that a contributor's remarks are attributable to that person only and not to any organization with which he or she is associated.

3 Consecutive passages in the Analysis imply no chronological sequence whatsoever; passages are juxtaposed entirely at the editor's discretion. However, any given extract always represents consecutive speech, except where 3 stops separate text. These stops indicate that separate extracts from the same contributor have been juxtaposed by the editor.

4 Many of the quotations used in the Analysis derive from presentations and discussions that took place at the State of the Art Lecture. Speakers at the Lecture can be identified in the presentations; the names and affiliations of the chairman and delegates appear on page 567. Other quotations derive either from invited papers, in which case the author can be identified from the front page of his paper, or from other sources, in which case the source is given with the quotation.

5 Where a paper is attributable to more than one author, only the first of the authors' names is given as a reference.

One further convention should be noted. Numbers in brackets and bold type appearing in the text refer to items in the Bibliography.

ACKNOWLEDGEMENTS

It is stressed that all contributors submitted material in an entirely personal capacity; their remarks must not be taken to represent necessarily the views of any organization with which they are associated.

Infotech would like to thank all contributors for the material submitted and is grateful for the cooperation of the following organizations, members of whose staff gave presentations at the State of the Art Lecture.

Software Sciences Limited
Bell Telephone Laboratories Incorporated
UK Atomic Energy Authority
Honeywell Information Systems Incorporated
IBM (UK) Limited
Esso Petroleum Limited
Computer Analysts and Programmers Limited
Barclays Bank Limited
Mathematisches Institut der Technischen Universitat München
University of Warwick
Boole and Babbage Incorporated

Infotech would also like to thank F Ford of The National Computing Centre Limited (in the UK) for providing material for the bibliography.

COPYRIGHT

INTRODUCTION

The nature of software engineering

Software engineering can be said to exist in that there is a strong analogy between the whole business of building and maintaining software and traditional branches of engineering.

PYLE: This State of the Art Report on software engineering causes us to think carefully about the total cycle of events in a computer-based system. The sequence of events is not much different from that in any other sort of engineering, but many of the details and relative importances are different when a substantial amount of software is involved.

The whole cycle is characterized by the acronym DITHER.

Design
Implement
Test
Handover
Evaluate
Replace

We design the system, implement it, carry out tests, and then hand-over to the user, who evaluates it and then sooner or

later decides to replace it and repeat the cycle.

Software engineering is concerned with all events in the cycle, related to systems substantially involving software.

Application of the engineering analogy to the building and maintenance of software implies the existence of a science of which software engineering is a practical application, in the same way that electronic engineering is a practical application of particular branches of the sciences of physics and mathematics. Computer science is, of course, the usually assumed theoretical basis for software engineering, although it is still debated whether or not computing is sufficiently advanced for it to be said that such a science yet exists. A substantial body of knowledge has, however, been developed in the comparatively short history of computing and those who doubt the existence of a computer science have their opinion stem either from the point of view that computing is in its early infancy and has not as yet realized anything like its full potential for development or from the observation that the current state of affairs leaves much to be desired when one looks at computing in practice. (In the latter, they are, in fact, talking about the engineering rather than the science.) Although, throughout the computer community, there is disillusionment after much initial optimism, such a slow rate of development with such a lag between theory and practice is usual in emergent branches of knowledge and expertise.

The term 'software engineering' has come into use since it was chosen as the title of the Garmish NATO conference in 1968 and subsequently the title 'Software engineering techniques' was chosen for the Rome NATO conference in 1969 (001,002). *A popular view at the time of the first of these conferences was that there existed a software crisis and that a software engineering was required to solve this crisis (see The software crisis, p19). The prevailing attitudes at the NATO conferences are discussed by McIlroy and Oestreicher. McIlroy discusses the attitude at Garmisch in terms of the*

development of a software components industry (see also p128).

McILROY: The Garmisch NATO Conference was a heady group therapy session, full of breastbeating about the software crisis and the general malaise of our trade and the way we practise it. I am not convinced there is any special crisis. The software I use seems to work as often as the airplane I used to own, the principal difference being that normally I was able to find out whether or not the airplane was sick before I took off.

We all have seen good software and bad software, and if you understand the least bit about programming from the inside you can readily distinguish the one from the other. Except in numerical analysis, good-and-bad in software is a night-and-day matter. Good means it works. Good means it works at a reasonable speed; and, for the areas of software where anyone can appropriately talk about engineering, the meaning of "reasonable" is usually obvious. Good means it works as you expect it to without resort to soothsaying or other magical invocations. Also, "good" has corollaries. If it is good, it is probably readable, or else deeply mathematical or legal. (Although you cannot turn labour agreements or fast fourier transforms into really perspicuous code, even these need to be intelligible to the initiate.) If it is good, it probably is not the first time its author has done it. If it is good, the author has probably done other good things too.

One intent of the industry is to institutionalize the good. I admit, but choose largely to ignore, that the industry intends also to make money. Marketing people working for the software giants have somehow ascertained that to make money you must stock a super-set of all other software; this is where we run into some trouble with the components idea. My NATO scheme was to have a giant catalogue of parts from which to build your software, all systematically classed by function, precision, speed, and robustness. Now this marketeer's union-or-all-good-things plan calls for a pile of hand-worked idiosyncratic pieces. You know them well and

I shall spare you a feast of excruciating examples.

Oestreicher discusses the attitudes at both conferences in terms of building what he calls 'critical systems', that is, systems in which a large part of the computer resources are dedicated to the project, the success of the enterprise depends on the correct and reliable functioning of the system, the services provided go beyond what could be done by manual means, and in which a lot of money is involved.

OESTREICHER: The first NATO conference, at Garmisch, I understand, left the participants with some feeling of excitment. At last we were getting somewhere. The second conference, at Rome, was rather optimistically entitled *Techniques of software engineering*, which seemed to promise some solid achievements, and indeed there were many extremely interesting papers submitted describing really excellent work, using advanced and powerful techniques, in the fields of portability, ensuring correctness, the use of measurement and other tools to improve efficiency, and so on. These represented achievements in the field of software engineering. However, I think it would be fair to say that very few of these achievements were made in the realm of what I have called critical systems.

An exception, you might think, was the report by J D Aron on the work done by IBM's Federal Systems Division for the Project Apollo space program. Now there was a critical system according to the definition if ever there was one. But the system Aron described had two distinctive features: extremely traditional programming methods and extremely good project managment. Also, Aron said that there were two factors that contributed greatly to making the project a success. These were: good relations with the client (NASA) and an effectively unlimited budget. For example, when it began to seem as though the late delivery of the System/360 operating system, OS, might delay the project, they (reluctantly) produced their own operating system (RTOS).

The reason I am mentioning the NATO conference at Rome is that I believe it was a rather disillusioning experience for many of those attending. There was, to put it very roughly, an unexpected gulf between theory and practice. The difficulty of applying sound engineering principles to the construction of critical systems was unexpected and little understood. I myself find it extremely difficult to describe what the difference is between a pilot project and a real project.

What we are likely to forget, of course, is how many pilot projects fail completely and never see the light of day. Such projects do not get described in papers presented at NATO conferences: they die very quickly. Therefore, we observers tend to get a very biased view. We ask, "Why is it that so many small projects are so well implemented, using such sound and advanced techniques, and no serious, large scale, critical projects have been brought off so neatly?"; the answer is that there was no foolproof way for management to distinguish between new, sound, software engineering principles, on the one hand, and good sounding new ideas that would never work in practice, on the other hand. Software technicians will sometimes blame commercial management for being wilfully blind, but management is understandably conservative when it comes to critical systems: better the devil you know.

So, on reflection, we should not be too disillusioned with this cultural lag we find between the demonstration of the efficacy of a technique or principle in an experimental situation and its wholehearted adoption throughout the industry.

Application of the engineering/science relation to the field of software is discussed by Kolence who proposes, rather than the software engineering/computer science relation already mentioned, the less loose relation of software engineering/ software physics.