THE CRAFT OF SOFTWARE ENGINEERING

Allen Macro

John Buxton

THE CRAFT OF SOFTWARE ENGINEERING

Allen Macro
Independent Consultant
John Buxton
King's College London

ADDISON-WESLEY
PUBLISHING
COMPANY

Wokingham, England · Reading, Massachusetts · Menlo Park, California
Don Mills, Ontario · Amsterdam · Bonn · Sydney · Singapore
Tokyo · Madrid · Bogota · Santiago · San Juan

Cover graphic by kind permission of Apollo Computer, Inc.
Typeset in 10/12 pt Times by Columns, Reading.
Printed and bound in Great Britain by TJ Press (Padstow) Ltd, Cornwall.

British Library Cataloguing in Publication Data

Macro, Allen
The craft of software engineering.–
(International computer science series)
1. Electronic digital computers –
Programming 2. Computer engineering
I. Title II. Buxton, John III. Series
005.1'2 QA76.6
ISBN 0–201–18488–5

Library of Congress Cataloguing in Publication Data

Macro, Allen
The craft of software engineering.
(International computer science series)
Includes index.
1. Computer software – Development. 2. Electronic digital computers – Programming. I. Buxton, J.N.
II. Title. III. Series.
QA76.76.D47M33 1987 005.1 86–32076
ISBN 0–201–18488–5

INTERNATIONAL COMPUTER SCIENCE SERIES

Consulting editors **A D McGettrick** University of Strathclyde
J van Leeuwen University of Utrecht

OTHER TITLES IN THE SERIES

Programming in Ada (2nd Edn.) *J G P Barnes*
Computer Science Applied to Business Systems *M J R Shave and K N Bhaskar*
Software Engineering (2nd Edn.) *I Sommerville*
A Structured Approach to FORTRAN 77 Programming *T M R Ellis*
An Introduction to Numerical Methods with Pascal *L V Atkinson and P J Harley*
The UNIX System *S R Bourne*
Handbook of Algorithms and Data Structures *G H Gonnet*
Office Automation: Concepts, Technologies and Issues *R A Hirschheim*
Microcomputers in Engineering and Science *J F Craine and G R Martin*
UNIX for Super-Users *E Foxley*
Software Specification Techniques *N Gehani and A D McGettrick* (eds.)
Data Communications for Programmers *M Purser*
Local Area Network Design *A Hopper, S Temple and R C Williamson*
Prolog Programming for Artificial Intelligence *I Bratko*
Modula-2: Discipline & Design *A H J Sale*
Introduction to Expert Systems *P Jackson*
Prolog *F Giannesini, H, Kanovi, R Pasero and M van Caneghem*
Programming Language Translation: A Practical Approach *P D Terry*
System Simulation: Programming Styles and Languages *W Kreutzer*
Data Abstraction in Programming Languages *J M Bishop*
The UNIX System v Environment *S R Bourne*
UNIX™ is a trademark of AT&T Bell Laboratories.

Allen Macro (BSc.; FBCS), the main author, has over 28 years of practical experience in software engineering and its management. He is sometime employee of the UK Atomic Energy Authority, founder and director of CEIR NV (now part of Scicon) and LOGICA BV, and consultant to major international companies including Philips Electronics and Royal Dutch Shell. He is a specialist in continuing education in software engineering and can be contacted at:

Serend (BV)
Valerius Rondeel 215
2902 CG Capelle a/d Ijssel
Nederland
Tel: 010 4507519

John Buxton (MA; FBCS), Professor of Information Technology at King's College London University, is a specialist in software development environments and languages. He was formerly Professor of Computer Science at the University of Warwick, and has been consultant to major international organisations including Philips Electronics and others; as advisor for the 'Stoneman' initiative of the US Department of Defense he was main author of the 'Stoneman' report on requirements for an Ada language support environment.

Preface

To most people, including a surprising number who program computers, software engineering is a mystery. Many adopt the attitude of the orchestra conductor who, when asked if he had ever performed work by a certain composer, replied: 'No, but I once nearly trod in some of it!' As Sir Thomas Beecham might well have agreed, such freedom of comment is often based on ignorance.

Software engineering has become, in fact, well known as a most difficult, costly and hazardous part of information technology. Some would say it is the most problematical technology of all, so many and severe have been the problems of software quality and adherence to estimates.

This book is for people who need and want to know what software engineering is, and how to do it. As well as practising software engineers and some more intermittent practitioners, we envisage amongst its readers a wide variety of people who are not specialists in the subject but who affect the software engineering process in one way or another. For instance, users who specify requirements; commercial and legal staff who may decide timescale and cost limitations on a software development; hardware engineers determining and working to a subsystem interface with software engineers; quality assurance departments and field support groups involved with software maintenance; personnel officers recruiting and helping to retain 'rare resources'; and so on down an extensive list.

Information Technology (IT) is now a major factor in everyday life. Governments see it as 'the sunrise sector' compensating for declining industries and, through its products, rejuvenating some whose means and methods are obsolete. Competition is fierce for a share in, or even domination of, its lucrative markets and, in view of the strategic issues involved, both economic and military, no industrialized society wants to become a large and permanent net importer of IT systems.

Thus, major national and international research and development policies are set, and the education sector is reformed and selectively financed in some countries as a palliative for supposed shortages in skills. For many practitioners and their managers, this is 'jam tomorrow'. In the meantime, IT developments go on, and – within them – so does software engineering, still beset by many of its old problems. For, in many cases, they are old problems – over three decades old in fact – now exacerbated by the expansion rate of IT applications and product innovation.

Some day – no one quite knows when, of course – the research initiatives now undertaken – such as the Strategic Defense Initiative in the USA, Japan's 'Fifth Generation' projects, the EEC's Esprit and Eureka programmes, the work of the Alvey Directorate in the UK and similar ones elsewhere – may further revolutionize IT and cure some (if not all) of the problems of making complex software. Until then software engineers, along with their managers, commercial and technical colleagues, will have to make software systems as best possible. That 'best' must not be as poor as it has been to date in many places, if IT is to have its beneficial effect.

For software engineering to be improved at this time, even within the known limitations of technical means and methods currently applying (and likely to do so for some time yet), it is imperative that the process of software engineering is understood between its exponents and the other populations involved. This book is intended as a bridge between people of different backgrounds, who are active parties to making software systems. Software engineering is not intrinsically unmanageable, but it may be made so. Software developments are not inevitably of questionable or poor quality, nor do they always over-run by 400% or more on cost budget – but these things may, all too easily, occur.

The problems of shared understanding and a shared vocabulary for expressing it are at the root of mismanagement and misdirected effort in software engineering, and are major causes in our view of the somewhat traumatic history of the subject. The problems are illustrated in the well known sketch:

Two deaf men are on a train as it approaches a London suburb:
'Is this Wembley?'
'I thought it was Thursday.'
'So am I!'

We submit this work to its readers and critics in the modest hope that it helps to alleviate deafness.

Acknowledgements

The whole of this text was prepared by one lady only; she has put up with highly volatile source material and the sort of behaviour one might expect to go with it, shameful handwriting from both authors (not to mention accompanying attitudes), under-estimated timescales, and a word processor whose software clearly had been written by amateur programmers. Naturally, having typed Section 6.3 on team structure, the

lady (in the best spirit of teamwork) wished to remain anonymous. But we are most grateful to this honorary software engineer.

A. MACRO
Capelle a/d IJssel,
The Netherlands

J.N. BUXTON
Kings College, London

The Publishers wish to thank the following for permission to reproduce extracts from published material:

Professor F.L. Bauer, for a quotation from *Software Engineering* (Amsterdam: North-Holland, 1972).

Barry W. Boehm, for quotations from *Software Engineering Economics*, pp. 75, 118, 374, 460, adapted by permission of Prentice-Hall, Inc., Englewood Cliffs, New Jersey.

Professor M.M. Lehman, for permission to adapt definitions given in *Programs, Programming and the Software Lifecycle*, Dept. of Computing and Control, Imperial College, London.

G.J. Myers, for a quotation from *Reliable Software Through Composite Design* (New York, Van Nostrand Reinhold, 1975).

C.H. Smedema, *et al.*, for a passage adapted from *The Programming Languages Pascal, Modula, Chili, Ada* (London, Prentice-Hall, 1985).

T.A. Thayer, *et al.*, for a quotation from *Software Reliability* (Amsterdam, North-Holland, 1978).

The *IBM Systems Journal*, for permission to reproduce a table from the article by Walston and Felix, 'A method of program measurements and estimation'.

Professor A.I. Wasserman, for a quotation from *Towards Improving Software Engineering Education* (New York, Springer Verlag, 1976).

Contents

Chapter 1 Introduction

Synopsis

Definitions are given for information technology and software engineering.

An account is given of the problems experienced in computer programming over the past three and a half decades, and how software engineering became Big and Bothersome.

The adverse factors currently affecting software engineering are summarized.

The approach adopted in this book is described and explained. The authors offer some advice to different types of readers on how to use this work.

It is taken for granted that the reader will be familiar with basic terms in computer technology (such as 'bit', 'byte', 'compiler', 'high-level language', and so on), or will have access to a lexicon of these terms such as the *Dictionary of Computer Science* (Glaser et al. 1983). Other terms more to do with software engineering as a process (e.g. 'lifecycle', 'functional specification', 'outline systems design', and a variety of others) will be defined and described in the appropriate chapters of this book.

1.1 Definitions: IT, software engineering, and programmers

To begin with, the terms 'information technology' and 'software engineering' need to be clarified. They are often used in the most imprecise fashion and their meanings thought to be common knowledge when, in fact, they are not. For instance, Information Technology (IT), Informatique, Informatica and (no doubt) many other variants, are apparently synonyms with only a loosely defined, generic meaning as they are commonly used. Given the scope and importance of the activities they seem to include, this state of affairs is unsatisfactory.

Recently, a working group in the United Kingdom – under Mr John

Butcher, MP – produced a definition of IT in the course of reporting on its 'skills shortages'. This definition (Butcher *et al.*, 1984) comprised

- Electronic systems and consumer electronics
- Telecommunication and radio frequency engineering
- Computing (software and hardware)
- Computing services
- Artificial intelligence
- Communication between electronic data processors
- Design and production of manufacturing systems (as distinct from their applications)

Classification can be difficult, particularly when categories overlap, are contiguous with unclear demarcation, or are cognate. The Butcher committee's definition is, with all its defects, a useful and succinct indication of the scope of IT. In the process it also makes clear the central importance of computers (essential to all eight components of the taxonomy) and, in doing so, highlights the central rôle of software engineering.

If, as is the case, software engineering is a 'problem subject', then that matter is as central to the interests of a company or country as is IT itself. If we must innovate to thrive economically, then we must solve the problems of software engineering with some urgency.

What then is 'software engineering'? In fact the term is more often used to denote programming – or even just the coding part of programming – than anything more extensive. The reasons for this are historical. Getting computer hardware to solve particular problems by executing sequences of commands was known as 'programming' and its exponents were known, therefore, as 'programmers'. As the first programmers were the problem solvers anyway (applied scientists and engineers) there were no problems with the definition. When, later, a tendency developed for the problem formulators to have programs written (often in a high-level language) by others, this task became known as 'coding' in some places. The inevitable confusion between two not explicitly defined terms was not helped when, around 1968 or so, the term 'software engineering' took on widespread use to denote a set of activities including 'programming' and 'coding'.

Nowadays, it should be taken that 'coding' is that part of programming that has to do with producing the sequence of instructions in a computer language to get the hardware to do whatever is required; 'programming', on the other hand, is the design of computer programs, their coding and testing by their authors as single programs or in combination with others as necessary.

Software engineering itself is the whole set of activities needed to

produce high-quality software systems (programs or suites of programs), within known limitations of resources such as time, effort, money, equipment, etc. These activities include specifications and feasibility (including prototyping if necessary), programming as defined above, quality control and assurance, and documentation.

As long ago as 1972 Bauer defined software engineering as:

> *The establishment and use of sound engineering principles in order to obtain, economically, software that is reliable and works on real machines.*

No doubt 'sound engineering principles' include management considerations, but there is a definite need to make that explicit. Also, the Bauer definition is less than clear on the need for adequate 'tools' to support the craft, as distinct from 'engineering principles'.

An extension of Bauer's definition in the current epoch might reasonably be that software engineering is:

> *The establishment and use of sound engineering principles and good management practice, and the evolution of applicable tools and methods and their use as appropriate, in order to obtain – within known and adequate resource provisions – software that is of high quality in an explicitly defined sense.*

A more detailed description of software engineering, including its main perceived properties as a process, is given in Chapter 2, along with a definition of software systems types resulting from the software engineering process.

Software engineers are people of sufficient relevant aptitude and ability who perform software engineering (in the sense in which it is defined above) as the whole or a major part of their vocation.

The 'relevant aptitude' attributes are far from easy to define, but are generally agreed to include literacy as well as numeracy, and mental characteristics such as associative abilities, capability in abstract thought, good memory and painstaking precision. As temperamental stability is a desideratum too, the unlikelihood of the combination is one reason why there is a shortage of good software engineers.

The 'ability' of software engineers, given the aptitude factors, is very much dependent on the variety and difficulty of software engineering tasks with which they are confronted, and the environment in which they make software (i.e. whether it has a strong 'culture' in good software engineering practice, as defined, at all levels; where it lacks such a strong culture it may be said to be a 'weak' software engineering environment).

Ability also concerns the stage that software engineers have reached in consolidating their subject knowledge into a framework of practice sufficient for most eventualities, yet flexible enough to be extended and modified when necessary. A career trajectory of increasing ability in this

sense begins with an apprenticeship, and proceeds through junior and senior software engineer levels to that of 'master' in the craft. This latter status is quite widely known as 'chief programmer' for historical reasons; properly the term should be 'chief software engineer', or even 'master software engineer'.

The topics of career development and subject education for software engineers are taken further in Sections 10.3 and 10.4. One major problem, due for passing comment here, is that the 'consolidation' of acquired knowledge may be inordinately delayed or even prevented altogether in weak software engineering environments. When this happens the 'software engineers' may more properly be categorized as 'amateur programmers'.

Not all computer programs are written by people whose main job it is to do software engineering. Many hardware engineers, whose main vocation is electronics, are intermittent programmers spending in the range of 10–40% of their time programming.

As with software engineers in weak environments, intermittent programmers are unlikely to have the strong background of experience, means and methods to permit the consolidation of acquired knowledge into a viable and practical conceptual framework. Merely knowing how to program in ABC language on an XYZ computer, having done it for a few weeks last year and the year before, is not enough – even if the results seemed to 'work'. The result is not software developed as a strong process of software engineering, by what may reasonably be viewed as software engineers, but programs written by 'amateurs' – people whose vocational specialization is, in many cases, in another subject altogether. As a result, its exponents are known as 'amateur programmers' – a category that contains a wide variety of people including some 'software engineers' and computer scientists, and many hardware (electronic) engineers.

The growth rate of the IT sector has led, ineluctably, to a very rapid growth in amateur programming. This is, of course, a major factor within the so-called 'software engineering crisis' one reads and hears about. For, whereas software engineering will be likely to produce software of requisite quality within the limits of resources provided (so long as these are adequate), amateur programming will be unlikely to do either on a consistent basis.

A third category of people who write computer programs is that of hobbyists. Whereas software engineers and amateur programmers write programs as part of their job of work, hobbyists do it for instruction or entertainment. Typically, a hobbyist will learn to program in BASIC on a Personal Computer (PC).

One major positive use of hobbyism is for managers and others who are not specialists in software engineering, as a part of an orientation in the subject. One negative effect of hobbyism is when it leads to amateur

programming. 'Give the job to Fred – he knows all about computers' is a grave mistake if the problem is large and complex, and all Fred has ever done is to write a couple of 20 statement programs in BASIC on his PC. Hobbyism has received a great boost in recent years through the policy of governments, enthusiastically supported by equipment manufacturers of course, to popularize computer technology. Putting a PC in every classroom, or a terminal on every desk, is a great idea for encouraging the development of some basic aptitudes and subject interest in secondary-school students. If, as it may, it leads to more and more amateur programming in a few years' time, it will have been rather counter-productive.

There is a software engineering 'crisis' at the present time. A summary of contributory reasons for it is set out in Section 1.3. Some of them are evident from what has already been said. What is not always realized, or admitted by those who know, is that there has always been a crisis in software engineering since the inception of computing as a wide-scale activity from about 1950 onwards. In some periods it was recognized; in others it was the object of comment and endeavour; in some epochs it was believed (or asserted) to be solved.

1.2 A short history of software engineering

The programming of non-trivial applications is, without doubt, an extremely complicated and difficult affair. The fact that a child can program its PC in something like BASIC does not confute that point – it confuses it. The control program in the washing machine that has just flooded the kitchen floor is a few thousand instructions of assembler code, but it took people a year and a half to develop. The SDI 'Starwars' research will require a software system of anything between 10 million and 100 million source statements of code and must work right (or at least adequately) first time – unless it is to be tested on a spare planet. Nor will it be done in BASIC, or at least it is to be hoped not.

What is it that makes programming difficult? The *Dictionary of Computer Science* (Glaser *et al.*, 1983) defines a computer and its programs as:

> *A device that is capable of carrying out a sequence of operations in a distinctly and explicitly defined manner. The operations are frequently numerical computations or data manipulations but also include input/output; the operations within the sequence may depend on data values. The definition of the sequence is called the program.*

By this definition computers and programming are of some antiquity. For example, Leibnitz in the seventeenth century; Jacquard, Babbage and

Hollerith in the nineteenth century; Turing and von Neumann in the mid-twentieth century, all made substantial contributions to computers as we know them.

In the period 1939–45 computers, incorporating electrical components as distinct from mechanical ones, were developed in the United Kingdom for applications in decryption, and in the United States of America mainly for computing gunnery tables. By about 1950, computers were becoming available from several commercial sources, and wide-scale use of computers – programming in fact – dates effectively from that period. In the following three and a half decades, computers and their usage went through many major evolutions. Looked at solely from the viewpoint of programming and programmers, the history divides into four epochs correlated with major developments of one kind or another (but not synonymous with so-called hardware 'generations').

Epoch 1 (*circa* 1950–58) This was the period during which the first amateur programmers acted as users, programmers and even hardware engineers, for problem solving in the field of applied science and engineering. They used large and expensive 'mainframe' computers and programmed them in number code or simple mnemonic assembler code. This programming was often looked on as a fine intellectual sport for physicists, chemists, mathematicians and engineers of various kinds. It was certainly a lot of fun, and these (we!) first 'amateur programmers' produced a lot of very clever programs which were for the writer's own use, and were then either thrown away or left until needed again by their author.

Epoch 2 (*circa* 1959–67) This was the period during which high-level languages were first introduced and then extensively used, and operating systems were evolved to make the use of facilities more efficient. Off-line input/output (character) orientated machines were brought out to delimit the processors from these functions, and as a result the world of business data processing (DP) was added to that of applied science and engineering computations. Formal career structures began to be seen for operators, coders, designers and systems analysts in this period, and the user (the amateur programmers from Epoch 1 and new users in the DP sector), became removed at several stages from the equipment and its programs. The motive for developing high-level languages, such as FORTRAN, ALGOL and COBOL, at that time was to make the task of programming easier for the users themselves. Thus instead of having to write something like:

0 00000001010 101 00000000101010011101

in binary number code, or

+ 21, 5, 5405

in decimal number code, or

ADD B, 5

in mnemonic assembler, one could write statements such as

D = (A + B(I))*C

in a high-level language, and this would be translated by a compiler in the computer operating system software into something like:

CLA A
ADD B, 5 (which is the 'instruction' given earlier in different forms)
MPY C
STO D

This assembler code would then be further translated into binary machine code and executed on the computer (i.e. it would regulate the operations of hardware circuitry in a prescribed manner). Coders tended to use high-level languages increasingly. Users faced with learning several languages and the increasing complexities of operating systems began to specify the problems to be solved rather than to be simply the programmers and computer operators themselves. Around this time the 'virtual machine' concept was developed, in which the computer was seen as being defined by the procedures of the operating systems software and its programming languages. The extension of these facilities progressively cocooned the hardware within increasingly large and complex software regimes.

IBM brought out a range of computers in this epoch – the '360 series' – intended to be modular (i.e. compatible at the software level, from the small and cheap level of the range upwards); to have 'time sharing' facilities for simultaneous multi-user operations; and to possess a full set of virtual-machine features of the operating system and several high-level languages such as COBOL, FORTRAN, PL/I etc. The 360 was an immense commercial success but its operating software – OS 360 – was something of a mess. It had taken 5000–6000 person years to develop and cost $50 million per annum during its development (source: E.E. David in Naur and Randell, 1968).

The first general appreciation of a crisis in programming resulted. If the world's most prominent manufacturer couldn't get it right, who could? The first NATO-sponsored conference on 'Software Engineering' was held (in 1968) as a result. The proceedings of these conferences make salutary reading, there being much of value for the present epoch in the questions raised and the answers given; they may be found in Buxton *et al.* (1969); and Naur and Randell (1968).

Epoch 3 (*circa* 1968–78) This was the period during which the minicomputer was invented and became widely used. For the first time the computer could be taken to the problem. Attempts to understand and