SOFTWARE ENGINEERING EXPLAINED

SOFTWARE ENGINEERING EXPLAINED

Mark Norris and **Peter Rigby**
BT, UK

JOHN WILEY & SONS
Chichester · New York · Brisbane · Toronto · Singapore

Other Wiley Editorial Offices

John Wiley & Sons, Inc., 605 Third Avenue,
New York, NY 10158-0012, USA

Jacaranda Wiley Ltd, G.P.O. Box 859, Brisbane,
Queensland 4001, Australia

John Wiley & Sons (Canada) Ltd, 22 Worcester Road,
Rexdale, Ontario M9W 1L1, Canada

John Wiley & Sons (SEA) Pte Ltd, 37 Jalan Pemimpin #05-04,
Block B, Union Industrial Building, Singapore 2057

Library of Congress Cataloguing-in-Publication Data

Norris, Mark.
Software engineering explained / Mark Norris, Peter Rigby.
p. cm.
Includes bibliographical references and index.
ISBN 0 471 92950 6
1. Software engineering. I. Rigby, Peter. II. Title.
005. 1—dc20 91-41133
CIP

British Library Cataloguing in Publication Data

A catalogue record for this book is available from the British Library

ISBN 0 471 92950 6

Typeset in 10/12pt Palatino from author's disks by Text Processing Department,
John Wiley & Sons Ltd, Chichester
Printed and bound in Great Britain by Courier International, East Kilbride

Preface

This book aims to introduce and explain some of the more important aspects of modern software development. It is not a detailed guide to any one part of the software development process, nor is it a reference manual for the software engineer. There are many excellent texts listed at the end of the book with more detail in specific areas . This is meant to be an overview of basic good practice in the specification design and operation of quality software.

Some chapters of the book are designed to stand alone—for instance the middle chapters (4 to 7) each deal with one major part of the software development process and are supported by a catalogue of techniques and a checklist to help their implementation. The book as a whole, though, is designed primarily as a source of information rather than a software reference manual.

The intended readership is fairly broad. Our aim was to write a primer for people with no formal background in software whose jobs have become dominated by it. In addition to this, the practical bias of the information here would be of use both to managers of software projects and to students about to embark on a career in software.

Above all, we hope it is interesting, useful to dip into, but mainly fun to read.

We would like to thank several people whose co-operation and forebearance have helped with this book.

Malcolm Payne for his constructive review of the initial drafts and his continual encouragement. Bob Higham for his careful reading of the completed text. Debbie Legassie for converting our hieroglyphics into English (American) text... and always with a smile on her face. To our many friends and colleagues in BT Software Development and Technology divisions whose experience and advice has been invaluable, especially Ray Lewis, Sinclair Stockman, John Foster, Dave Bustard, David Horncastle, Trevor Matthews and Jonathan Mitchener. Last, but not least, our wives, Fiona and Liz, for their forebearance, patience and support throughout.

M. Norris
P. Rigby

Foreword

Software, or rather the ability to produce it, is now recognised as the key to competitive advantage in a high technology business. This book is a guide for those who want to win that advantage. It is not theoretical—there are already many excellent textbooks that address those aspects of software engineering.

The book aims to highlight the key concerns that have to be addressed by anyone involved in the development of software-dominated systems. For each of these concerns the book explains the current state of the art, what the known pitfalls are and what tools and techniques are available to help.

The book brings modern software engineering techniques to professionals who are working in the software industry. This does not mean to say that it looks solely at the software engineering professional—many engineering professionals wish to adopt software engineering as a second string to their bow.

A second category of audience for this book is those undergraduates who are intending to take up a career in the commercial software field, as it introduces the techniques required to scale theory into practice.

Before we look at the possible solutions that this book may offer, it is interesting to spend a few minutes reflecting on the nature of software itself. It is characterised by the fact that the whole product is documentation in one form or another. Indeed, in many instances, it is just a very long piece of program text, generated with no concept of how it can be broken down into configuration items and subsequently rebuilt to form the product.

The principle of structured software, that each portion is simple to understand and modify, is crucial. This can only be achieved in software—the product—if the process by which it is produced is well founded. The main part of this book therefore addresses the main process steps in producing reliable, long-lived software, from requirements capture, through to maintenance and eventual retirement.

At the end of the day, whatever the software application and however it

is defined, its objective is to instruct a machine to do a certain specified operation. It must be able to do this operation time and time and time again, without fail, to the satisfaction of the people who are using it. The important point that has emerged in software development over the last few years is that the creation and support of a system is a demanding intellectual process. This book brings together several years of experience and investigation into the process, and I would like to think that reading it will save the reader from reinventing the wheel, and more especially from coming up with a square one without giving thought to the option of rounding the corners.

A.G. Stoddart

Contents

1 Why Software Matters

There are only two commodities that will count in the 1990s. One is oil and the other is software. And there are alternatives to oil

Bruce Bond

Software engineering is all about producing what the customer wants within time and cost constraints. A quality product to do this requires a mix of technical and organisational skills. That is what this book is all about. Later chapters look at some of the important tools and techniques that can be used in developing a software system. Before we start, though, there is an important question that needs to be answered:

Does software quality really matter that much?

There is no straightforward answer to the question but there are ways of gaining some insight on it. The first is in terms of the damage that can be wreaked by faulty software, the second is simply based on the cost of producing and maintaining software. The next two sections give a few examples of the importance of software quality from the 'catastrophe' and 'cost' perspectives. The reader is left to assess the relevance of the examples in their own environment.

1.1 SOFTWARE FAILURES

For obvious reasons, the developers and owners of software systems—especially those in safety or life-critical areas—are usually unwilling to discuss failures which have occurred. Nevertheless, some notable instances of failure have been recorded and they provide valuable evidence of the critical need for software quality.

Radiotherapy equipment

A radiotherapy machine was designed to operate in two different modes. In the first mode, the machine delivered a low dosage of radiation; in the second mode, the machine delivered a much higher dosage to a smaller area, with a mask in place to screen the rest of the patient. In the reported incident, the high dosage was given, without the mask in place, and the patient died.

According to the inquest, the control system for the radiotherapy machine was a software-controlled replacement for an earlier hardwired version. In the earlier version of the machine, there was a safety interlock which prevented the high dosage unless the mask was positively in place; it appears that the software version of the system lacked this interlock.

Misguided torpedo

A conference on safety and security of software systems featured a report on the safety system in a torpedo that was designed to prevent it returning in error to destroy the ship which had launched it. It achieved this by detecting that the torpedo had turned through 180° and was threatening its source. If this was the case, it was automatically detonated before it could return to do harm.

Unfortunately, when the torpedo was being tested, it was launched with a live warhead, but its motor failed and this left the live torpedo lodged in the torpedo tube. The ship's captain decided to abandon the test and return to port. As soon as the ship was turned round, the torpedo did its duty and exploded in the tube.

Autoland system

A prototype automatic plane landing system was designed to work in two stages. In stage 1 the plane flew down a beam which determined the approach path. If the plane lost the beam, the system applied power and flew the plane around for a retry. In stage 2 a ground detector sensed that the plane was within a few feet of the ground, cut the engine and raised the nose to land the plane. In separate testing both systems worked perfectly.

In the first live test, both systems again worked perfectly. The plane flew down the beam until the second system detected that it was near the ground, then the engine cut back, the nose was raised, and the plane started to sink. Then it lost the beam. The engine was boosted to high power and the nose was lowered by the first system. Before the second system cut back in, the plane flew into the ground.

A similar system, developed a few years later, demonstrates a less dramatic but no less serious problem. In this instance, as before, the software worked

perfectly. During tests the plane landed every time with unerring accuracy. After a few weeks, however, revision of the system (i.e. rework of the software) was required, not because of malfunction but because the test runway was beginning to break up where the plane unfailingly alighted.

This second example illustrates an important point. Despite the fact that not all system failures are the fault of the software, it is the system element that is, almost invariably, changed to cope with such unforseen circumstances as described above.

Chemical plant

The specification to the programmers stated that, if an error was detected, they should keep all controlled variables constant and sound an alarm. The programmers were not chemical engineers.

In the reported incident, the system received a signal from an oil sump that the oil level was low. In accordance with the specification, the system stopped changing any values and issued an operator alarm. By coincidence, the system had just released a catalyst into the reactor and was in the process of increasing the water flow to the condenser. The water flow was held at a low level, the reactor overheated and the pressure release valve vented a quantity of noxious fumes into the atmosphere. Meanwhile the operators investigated the alarm, discovered that the oil level was actually correct, and did not notice the reactor overheating.

None of these examples are exceptional. In fact they appear to be fairly reproducible. A recent report of a high-speed-train braking system revealed that the prototype implementation was designed to operate in one mode up to 50 mph and in a second mode above 50 mph. Unfortunately, attempts to stop the train when it was travelling at precisely 50 mph were fruitless as neither mode of braking would operate at this speed. The problem, in essence, was the same as the first of the autoland incidents described earlier.

The reason for revisiting these sad cases here is simply to illustrate the fragile divide between satisfactory operation and disaster.

In order to learn from these mistakes, it is important to examine the root causes of failure—missing requirements, a misunderstanding of the function of the system, and the like. A point that will come up several times in this book is that software quality is not simply a matter of well structured code or accurate design. It relies on a systematic approach that covers the entire system development and allows inconsistencies to be revealed [Hil88].

Returning to the original question of whether software quality matters, there are many applications that rely to some extent on software, from nuclear reactors to fly-by-wire passenger aircraft to banking systems. Failure as described above could be catastrophic in any of them It would be unrealistic to expect perfection on every occasion but the value of eradicating any error should be clear.

1.2 SOFTWARE COSTS

To some people, money is almost as emotive an issue as life itself. For that select band (and for the information of everyone else), there are some interesting figures that show just how much software costs now and is likely to cost in years to come.

Before going into detail, a few trends taken from a number of the strategic reports published [ACA86] highlight the central importance of software to all companies involved in any form of information technology. Some of the more striking points made are:

- Software currently accounts for about 5% of the UK gross national product and, given the trends over the last ten years (shown in Figure 1.1a), this is likely to grow.
- The proportion of IT costs attributable to software rose from 40% in 1980 to 80% in 1990 (see Figure 1.1b).
- The European software services market, estimated at 40BEcu in 1990, is set to rise to more than 60BEcu by 1993.

In addition to these gross-size figures, these strategic reports have identified a number of key problems, the main ones being [Hob90]:

- On average, large software systems are delivered a year behind schedule.
- Only 1% of major software projects finish on time, to budget.
- 25% of all software-intensive projects never finish at all.
- Over 60% of IT product managers have little or no experience of modern software engineering practice.

The effects of the above factors have been estimated, in cost terms, to be of the order of two billion pounds a year in the UK alone. The overall picture is of a costly area of technology, growing rapidly in which failures are rife.

So far we have set the general environment and trends. To give a better feel for the actual scale of investment in software, we can look at a 'typical' small software company. Since there is no apparent source of reproducible information on industrial software productivity, the line taken is to state what is known, or can be reasonably assumed and to derive a reasonable overall picture.

Assumptions

(a) There are about 100 engineers in the company involved with software development and maintenance.

(b) The split of effort on development and maintenance is 47:53. This figure is an average across a number of software maintenance surveys and accords reasonably well with published data from the US.

(c) In terms of lines of code (loc), the average for development is about 20 loc/person/day. For maintenance an average programmer load is about 17 000 loc/person/pa. The former is a guess based on hearsay and experience, the latter is from published data.

Observations

(a) Given the number of software engineers engaged on maintenance, 53, and the amount that each can maintain, we can derive that there are about 900 Kloc deployed at any one time.

(b) The other 47 engineers must be developing code at a rate of 20 loc/day for 45 weeks in a year. This equates to around 210 Kloc of new code a year. There are a number of factors that come into play at this point: the amount of code taken out of service (about 90 Kloc every year, assuming an average lifetime of 10 years); the attrition rate of projects (if 25% never complete, only about 155 Kloc are usefully added every year; 90 Kloc replacement, 65 Kloc new).

(c) Taking the example figures used above, our small company increases the amount of software deployed by 65 Kloc per annum. This rise of just over 8% equates quite closely with the general trend for increasing proportions of software in systems (but ignores any increase in the number of systems, so is probably on the low side).

For all their subjectivity, the above figures are not at all unexpected—they are reasonably in line with common experience and can be backed up with published material. There are two key points that emerge:

- That if our small company had to replace its entire software base, it would, by any standards, be a costly exercise.
- The annual bill for software development and maintenance is large (simply based on the volume of code) and is growing.

Both of these points indicate that the cost of poor quality is potentially high: if a software product is not fit for its purpose it can become more of a liability than an asset [Boe88].

One final point that needs to be considered in answering the question of whether software quality matters is that of liability. It is likely that, in future, suppliers of software systems will be accountable for the sort of failures

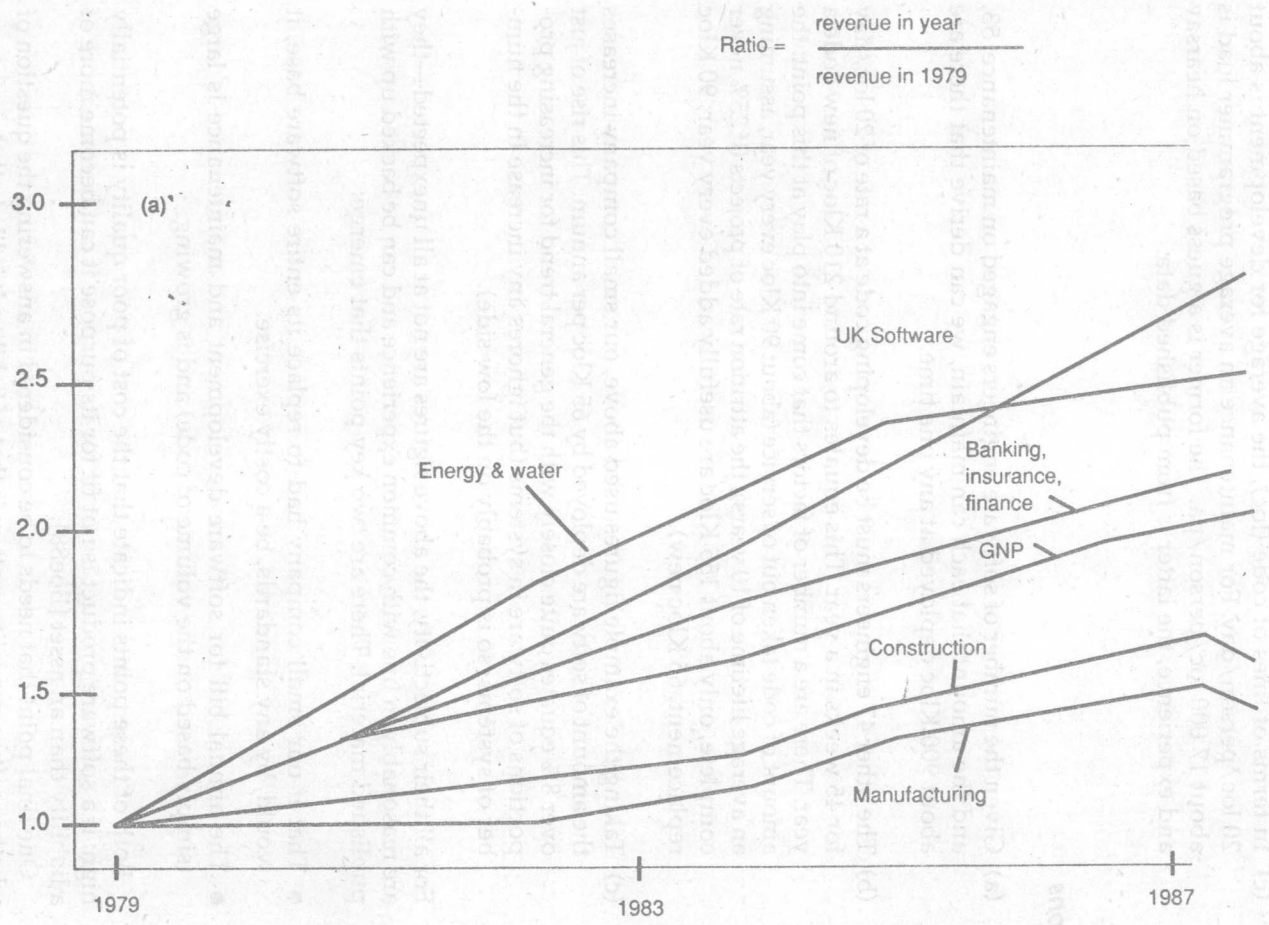

Figure 1.1(a) The increasing economic importance of software through the 1980s in the UK

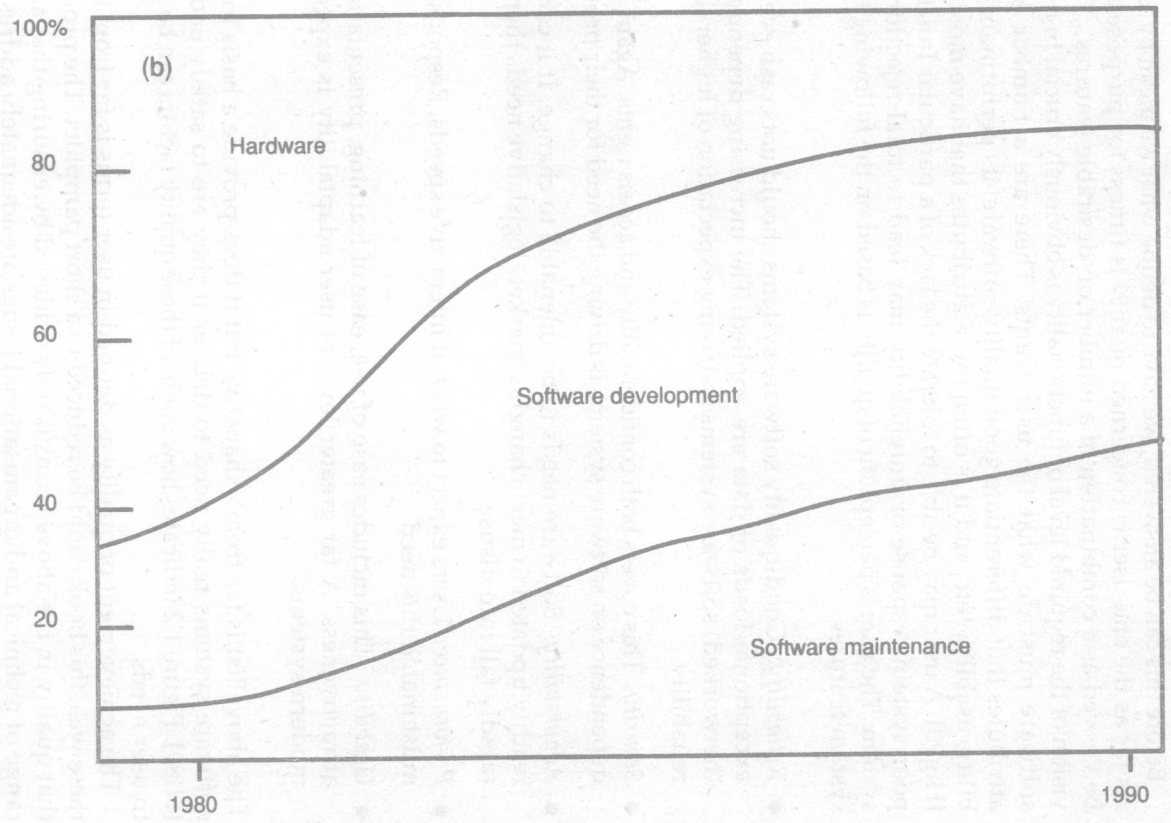

Figure 1.1(b) The increasing dominance of software as a system component

illustrated at the beginning of this chapter. In Europe it is already the case that due care and attention has to be shown by the supplier. The prospect of law suits could well introduce a new clarity to the costs of failure!

Overall, it would seem that there is more than sufficient cause for concern over the quality of software, both in terms of safety and cost. The question now is what can be done to achieve 'quality'.

Before this can be answered, we have to define what we mean by 'quality'. As far as the end user is concerned quality is fitness for purpose and can be viewed as a combination of a number of 'desirable features'. The provision of the required level of functionality is obviously crucial in this—the software must do what the user wants. There are a number of other attributes that differentiate good quality software; its performance, reliability, usability, etc. and it is often these attributes that have most impact [Hig90]. A user may be able to tolerate the lack of a particular function but poor system response or unreliability may lead to total rejection of the system. The user's perception of quality is based on the following types of system features.

- *Reliability.* Good-quality software systems should not crash every time exceptional loads or data are applied. The increasing dependence on (networked) software systems is driving expectation of higher levels of reliability.
- *Security.* This covers both confidentiality and access rights. Again, greater dependence on software systems is driving the need for their protection.
- *Adaptability.* Software needs to be amenable to change. If it cannot be readily updated to meet changing market/legislative needs then it will rapidly fall into disuse.
- *Performance.* Users expect to work at interactive speeds. Response times must match this need.
- *Usability.* This includes ease of use, ease of learning, presentation and attractiveness. A far greater level of user adaptability is expected in modern systems.

The above list is far from exhaustive but it does provide a basis for what software systems really need to deliver if they are to satisfy end users [FII89]. Figure 1.2 indicates how some of these quality factors can be related to user needs.

The achievement of quality as defined in user terms is far from easy. If there was, this book could be reduced to a short pamphlet. The problem is that quality in the above terms is only achieved by ensuring that a whole range of technical and organisational issues are adequately addressed by the supplier. And there is no formula or recipe to ensure that the tools and techniques used to produce the software will result in what the user really wanted.