SOFTWARE ENGINEERING: ANALYSIS AND DESIGN

SOFTWARE ENGINEERING: ANALYSIS AND DESIGN

Charles Easteal
University College London

Gordon Davies
The Open University

McGRAW-HILL BOOK COMPANY

London · New York · St Louis · San Francisco · Auckland · Bogotá · Guatamela
Hamburg · Lisbon · Madrid · Mexico · Montreal · New Delhi · Panama
Paris · San Juan · São Paulo · Singapore · Sydney · Tokyo · Toronto

Published by
McGRAW-HILL Book Company (UK) Limited
MAIDENHEAD · BERKSHIRE · ENGLAND

British Library Cataloguing in Publication Data
Easteal, Charles
Software engineering: analysis and design. —
(International software engineering series)
1. Computer systems. Software. Development
I. Title II. Davies, Gordon III. Open
University IV. Series
005.1
ISBN 0-07-707202-2

Library of Congress Cataloging-in-Publication Data
Easteal, Charles.
Software engineering.
(The McGraw-Hill international series in software engineering)
Bibliography: p.
Includes index.
1. Software engineering. I. Davies, Gordon.
II. Title. III. Series.
QA76.758.E27 1989 005.1 89-2598
ISBN 0-07-707202-2

1234 WL 8909

Typeset by STYLESET LIMITED · WARMINSTER · WILTSHIRE

Printed and bound in Great Britain by Whitstable Litho Ltd

PREFACE

This book is based on material that was originally written for the Open University course *Fundamentals of Computing*; that course assumed no previous knowledge of computing whatsoever. However, the book is aimed more at students who have had some experience of computing, perhaps as a general introduction in a science or engineering degree, and who wish to gain further knowledge of how software is produced in the real world. Such students will not necessarily specialize in computer science while at college or university, but may well see their future employment as being in the computer industry.

The book deals with two phases of the Software Life Cycle. After two introductory chapters, three chapters are devoted to the analysis and specification of requirements; the final seven chapters deal with software design.

The book has been written with specific objectives for each of the main topics in mind. At the end of the chapters on analysis and specification of requirements the reader should have a general appreciation of the importance of this phase and, in particular, be able to:

1. Analyse a set of initial user requirements into four major categories, and be aware of the role that each category plays in software development.
2. Use one major technique for further detailed analysis of user requirements and employ three useful notations for recording the results of the analysis.
3. Compile a detailed specification of requirements to enable the design of the new software to commence.

On completing the chapters on software design the reader should have acquired an appreciation of the general principles involved and of the fundamental difficulties that are encountered. In particular, the reader should be able to:

1. Apply a particular and generally reliable design strategy to a representation of functional requirements in order to arrive at a first version of an initial design.
2. Refine the initial design by using design heuristics while appreciating the limitations of this procedure.

3. Convert each of the modules identified by 1 and 2 into algorithmic form and record the resulting detailed design by means of a natural language or graphical notation.
4. Design appropriate data structures for the algorithms designed in 3 to work upon.
5. Construct the appropriate documentation that terminates the design phase.

A number of exercises of varying difficulty are interspersed in the text and the reader is encouraged to spend a little time in attempting to answer these, as they are encountered, before studying the solutions which are included at the end of the book.

ACKNOWLEDGEMENTS

We would like to acknowledge the use of the following figures and quotations.

The following extracts from BS 6224:1987 are reproduced by permission of British Standards Institution. Complete copies of the Standard can be obtained from them at Linford Wood, Milton Keynes, Bucks., MK14 6LE, United Kingdom.
Figure 9.1b (BS Fig. 3); Figure 9.4 (BS Figs 5 and 6); Figure 9.9 (BS Fig. 7); Figure 9.11 (BS Fig. 8); Figure 9.7 (BS Fig. 10); Figures 9.2 and 9.3 (adaptation of BS Fig. 44b); Figure 9.5 (BS Fig. 45b); Figure 9.15 (adaptation of BS Fig. 47); Figure E13 (adaptation of BS Fig. 48).

The example in Section 7.2.3 (page 46) is adapted from J. C. Emery, *Organizational Planning and Control Systems*, Macmillan, London, 1969 (pages 18–19).

Figure 8.3 is adapted from G. J. Myers, *Reliable Software through Composite Design*, Van Nostrand Reinhold, Wokingham, 1975 (Figs 4.8 and 4.9).

The quotations on page 28 ('It is imprecise, wordy ... and innuendo' and 'It is pidgin language ... programming language)') are both taken from T. DeMarco, *Structured Analysis and System Specification*, Prentice-Hall, Englewood Cliffs, New Jersey, 1979 (pages 177 and 179, respectively).

The following are taken from E. Yourdon and L. L. Constantine, *Structural Design: Fundamentals of a Discipline of Computer Program and Systems Design*, Prentice-Hall, Englewood Cliffs, New Jersey, 1979:
Quotation on page 57 ('A module is a lexically contiguous ... identifier' from page 37); the adapted definition of functional cohesion on page 68 (from page 127); Figure 8.9 (similar to Figs 8.7, 8.8, 8.9, 8.10 from pages 153 and 154); Figure 8.19 (adaptation of Figs 10.5 and 10.6 from pages 193 and 196); Figure 8.20 (adaptation of Fig. 10.10 from page 200); Exercise 8.7 (adaptation of Figs 9.9a and 9.9b from pages 175 and 176).

The following are taken from M. Page-Jones, *The Practical Guide to Structured Systems Design*, 1st edn, Yourdon Press, New York, 1980:
Figure 8.4 (adaptation of Fig. 6.3 from page 105); the quotation on page 85 ('A transaction ... a new time slice' from pages 207–208).

Figure 11.1 is taken from R. S. Pressman, *Software Engineering: a Practitioner's Approach*, 1st edn, McGraw-Hill, London, 1982 (adaptation of a whole figure from 'Software Design Specification', pages 133–135).

CONTENTS

1

INTRODUCTION

As a member of our target audience, you will almost certainly have written and run some computer programs. But it is important at this juncture to make three major points about your achievements:

(a) The programs that you wrote and handled were small. Although, no doubt, you were quite impressed with what could be achieved with a few lines of code, it is important to realize that many computer applications involve hundreds or even thousands of program statements. Indeed, an application consisting of a quarter of a million statements, say, is by no means unusual.

(b) In writing and running your programs you followed a detailed list of instructions that had been prepared by computer professionals, such as your lecturers or instructors. As a result, the instructions were free of ambiguity and left you in no doubt as to what you were required to do. In practice, of course, the vast majority of computer programs are not prepared for computer professionals. They are written by computer professionals for other types of professional, e.g., managers, engineers, accountants, etc. These individuals are generally known collectively, and perhaps rather loosely, as 'users'. And this is the term that will be used in referring to any organization or individual who requests or in any way sponsors, the development of computer software for practical purposes.

(c) Once the more obvious errors had been removed from a program that you had written, you were able to regard the job as finished. In the real world of computing, this is almost never the case. Just about every substantial piece of software that is released still contains errors. These will appear intermittently during the life of the program and will need to be corrected. As the working life of the program will often far exceed the life of the equipment on which it is first implemented, then it is clear

that each program written represents a substantial future commitment for an organization's computer personnel.

Exercise 1.1 One reason has been given above (and another hinted at) as to why a program will need intermittent attention throughout its working life. Can you think of any other reasons why this must be so?

It should be apparent from the solution to the above exercise that the practical development and continuing efficient running of computer programs is a sizable on-going task, involving many people. Further, it is a multi-stage process in which the outputs of one stage are the inputs of the next. This sequence of stages has come to be known as the *software life cycle*.

2

THE SOFTWARE LIFE CYCLE

In order to discuss fully the activities involved in the life cycle, it is necessary to adopt a *model* of the cycle. The term 'model' may need some explanation. A model of something is merely a representation of that something that enables one to investigate its properties in a remote way. For instance, if the headlight on your motorcycle fails, even though you know that the bulb is working, you might be tempted to try and trace the wiring and hope to find a loose connection. You might be better employed, initially any way, in looking at the wiring diagram of the machine. You would then be studying something (the motorcycle wiring) remotely (in a warm kitchen rather than a cold garage) by means of a model (the wiring diagram). In the example case the model is constructed by means of a graphical notation, i.e., it is a picture. But often modelling is achieved in other ways. Natural language and mathematical symbols, for instance, are among the wide variety of notations that may be used for this purpose.

For example, consider the Lake District, a beautiful part of Nothern England. We can model the Lake District in two different ways, one using a graphical notation and one using natural language. An obvious graphical model would be an Ordnance Survey map; although photographs, provided that they concentrated on the scenery rather than bands of happy hill-walkers, would be an equally acceptable example. A guide-book would be a good instance of a natural language model, although you might suggest Wordsworth's poetry as being more in keeping with the spirit of the Lake District.

It is convenient for us to use a graphical notation to describe the software life cycle model and this is shown in Fig. 2.1.

You should note two basic points at this stage. First of all, the model that we are using is a graphical representation of a physical process. Not everyone will view the process in exactly the same way, so that you will find many variations on this model in textbooks. You should not allow this to confuse you, for the fundamental activities that

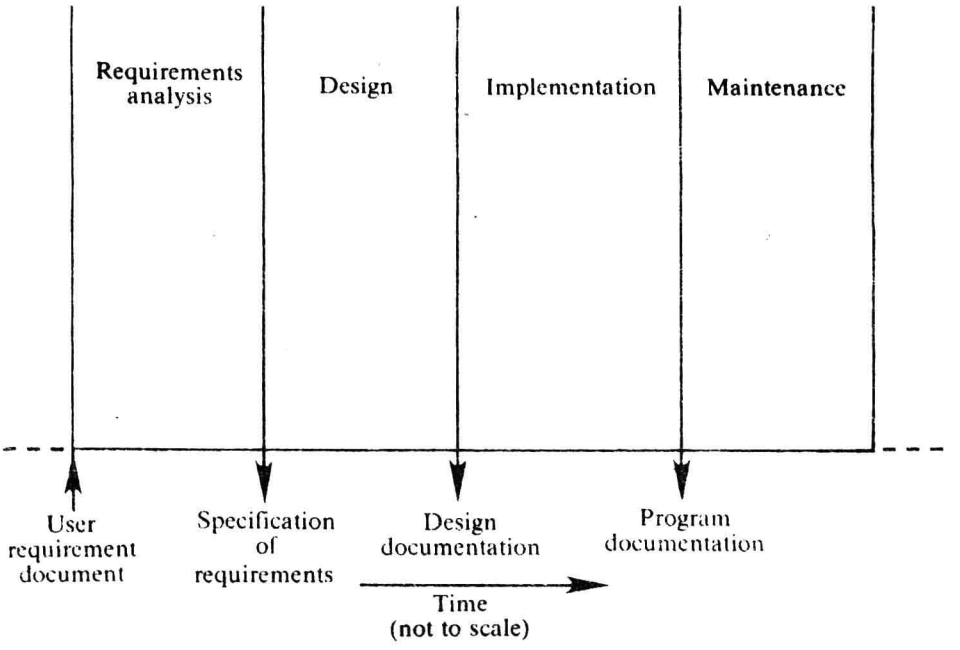

Figure 2.1 A simple model of the software life cycle.

are modelled will always be the same. The second point relates to the use of the adjective 'simple' in the figure title. You will probably infer that the life cycle may sometimes be more complex and, of course, you will be quite correct. But the reasons for the added complexity will not become clear until later in the book. As we have modelled the cycle it is apparent that it is a consecutive sequence of phases: requirements analysis, design, implementation and maintenance. The beginning and end of each phase is marked by the introduction or production of documents: the user requirement document, specification of requirements, design documentation and program documentation. The essential aspects of the simple model will now be described briefly.

2.1 USER REQUIREMENT DOCUMENT

The cycle commences with the receipt of this document by the body of individuals who are mandated to develop software for the user organization. The body may be an external contractor or a separate department of the user organization. The document should include statements of functional requirements. In other words, it should state clearly and unambiguously *what* is required of the software but not *how* this is to be achieved. For instance, a requirement might be that the proposed software should update a table. This is a perfectly acceptable expression of desired functionality. A statement which specifies the representation of the table in the computer is not, and it may be necessary to argue strongly for its exclusion. Just to reinforce this point, consider each of the following statements: is each one a 'what' or a 'how'?

Pop down to the newsagent and get me *The Times*.
Pop down to the take-away on your bike and get me a portion of the special fried rice.

The different nature of the purchases mentioned here is, of course, quite irrelevant. What is different is that the first statement is expressing a need; the second is expressing a need and giving explicit instructions as to the means of transportation to be used. Thus the first requirement is a 'what' and the second is a 'how'.

Usually, the user requirement document will also include some non-functional requirements. Most of these are acceptable and, indeed, may make life easier for the computer professionals of the development team later in the life cycle. A requirement that the average response to an enquiry of the database should not exceed five seconds, would come into the non-functional category. The distinction that we make between functional and non-functional requirements will become more clear in Sec. 3.1 of the next chapter.

2.2 REQUIREMENTS ANALYSIS

This represents a period of interaction between the user and the analyst, the latter being the computer professional assigned to work with the user during this phase. The original requirements are examined and tested for internal consistency. In other words, any contradictions or ambiguities among the requirements are discussed with the user until they are resolved to the satisfaction of both parties. The requirements are then refined until the user and the analyst are in complete agreement as to the expected detailed behaviour of the new software. It is important to remember that the majority of real-world programs that are written are replacements. They are intended to replace procedures that for various reasons have become obsolete. The procedures to be replaced may be manual or, increasingly, automated processes that no longer meet current needs. In these circumstances the behaviour of the existing system needs to be examined, for much of its existing functionality will need to be duplicated in the new system.

Exercise 2.1 How would you set about determining the behaviour of an existing procedure such as a group of clerical routines?

From the answer to Exercise 2.1 you will gather that requirements analysis can be a time-consuming exercise. Many man-months of effort may be necessary before the functionality of an existing system can be thoroughly understood. Finally, you should note that the emphasis of the work is still on what the software should do. The question of how it is to do it, is premature at this stage.

2.3 SPECIFICATION OF REQUIREMENTS

The production of this document ostensibly signals the end of the requirements

analysis stage. In fact, the matter is not quite that simple but the reservation implied in the previous sentence need not concern us at the moment.

It is important to appreciate that the specification of requirements fulfils a dual role. On the one hand, it represents a form of contract between the user and the agency responsible for developing the new software. Consequently, the very substantial part of the specification that deals with the expected function must be written in a notation that is familiar to, and understandable by, the user. A notation based on English, or the appropriate natural language, suggests itself. On the other hand, it represents the starting point for the design phase and must therefore be easily interpreted by the software designer; a more formal graphical notation with less scope for ambiguity would seem more suitable. A notation that perfectly fulfils both roles has never been developed and probably never will be, although many attempts have been made. However, the combination of notations suggested in Chapter 3 enables a fairly good attempt to be made at achieving the ideal.

Of course there are some circumstances where a graphical notation is used publicly in preference to natural language. This is most likely to happen where there is a need to address people who have different natural languages, particularly if the message is to be put across without delay — hence the international convention for road signs and symbols on dashboards.

2.4 DESIGN

With the commencement of the design stage the attention of the software developers focuses on the question of how the user's requirements are to be implemented. This means that ideas on the structure of the programs and the data structures on which they will work are generated, and the best ideas are selected for further development. This is not as simple as it sounds. Consider a fairly simple design problem, the need to add three numbers. In how many ways could this be accomplished? (In other words: how many possible designs are there?)

The answer is four. If we let A, B, and C represent the numbers, we could add A and B, and then add the sum to C. Similarly, we could calculate $A + C$ and add it to B, or $B + C$ and add it to A. Finally, we could calculate $A + B + C$ in one fell swoop, thus making the fourth design.

But let us now take this design problem a little further and ask 'Roughly how many designs are there if we wish to add fifty numbers?'

If you arrived at the answer, 'quite a lot', then this would be acceptable; for the answer is 6.85×10^{81}. As a matter of interest, this is several magnitudes higher than the estimated number of atoms in the universe.

This gives some idea of the true magnitude of apparently simple design problems (Emery, 1969). In the case of software the number of distinct designs that could be considered is also immense — probably running into thousands for a small system; millions for a large one. This implies that a designer needs to use a design strategy in order that design may be accomplished in a reasonable time. The second feature that complicates the design process is the need to demonsrate that one design is superior to all others. This can only be achieved convincingly if the designer is able to measure

competing designs against some appropriate 'yardstick'. This illustrates the need for design criteria, of which, unfortunately, there is no shortage.

The need for design criteria arises in many other situations. For example, what criteria do you think are used by the designer of motor cars?

Without knowing anything about this subject whatever, we presume that the design criteria include economy of running, certain aspects of road speed, safety, comfort, emission rates of noxious fumes, and so on.

The point that we wish to make is that comparing designs becomes increasingly difficult as the number of criteria increases. Hence the word 'unfortunately' in the sentence at the top of the page. The answer to Exercise 2.2 takes the matter a little further.

Exercise 2.2 What qualities, attributes or properties do you think should be taken into account in comparing software designs?

Detailed consideration of strategy and criteria must await a later chapter. For the moment it is sufficient to note that when the design stage is completed, the necessary documentation is available to enable the next phase, implementation, to proceed.

2.5 DESIGN DOCUMENTATION

The *design document* provides a channel of communication between the designer and the programmers who will convert the design into working computer programs. As such it will need to be expressed in a notation, or combination of notations, that leaves no shadow of doubt in the programmer's mind as to how the programs should function. Clearly, the notations need to be more formal at this stage and, consequently, less intelligible to the casual reader. Strictly speaking, there is no necessity for the user to understand this particular document so that its rather insular format need be of little concern.

However, it is sometimes the practice to issue other documents at this stage and these may have a wider readership. The *systems manual* is intended to provide a guide should changes need to be made to the system once it is in operation. It may need to be targeted, in part, at the non-expert user, particularly if the new software interfaces closely with clerical or manual routines. A *user manual* must be produced at some stage and the end of the design phase is often most appropriate. As its name implies, it is intended as the key reference to the system for the people who will actually use it. Finally, the user personnel need to learn how to use the system. Accordingly, a *user guide* or *tutorial* is often produced at this time.

2.6 IMPLEMENTATION

This phase involves a number of activities, of which writing and documenting programs is only one of the more important. It is commonly acknowledged that large software systems are best designed as a set of small, more easily handled pieces known as

modules. As most of you will, by now, be used to writing computer programs, you will have encountered the concept of program modules before. The name that was given to them will vary, depending on the language you used.

Modules are equivalent to sub-routines, in the broadest sense, and there are specific programming language variants, e.g., procedure (Pascal and PL/1); function (Fortran); sub-program, section or paragraph (Cobol).

The testing and debugging of individual program modules is a critical activity, as is the *system test*. The latter is a full-scale attempt to ensure that all the program modules work together harmoniously and satisfy the user's requirements.

If the new software is a replacement for existing procedures, a possibility that we mentioned in Sec. 2.2, then the introduction into service of the new system may be regarded as being part of implementation. This is commonly referred to as *cut over* or *conversion*. It needs to be planned very carefully to ensure that the transition takes place smoothly and does not entail any loss or corruption of data.

Exercise 2.3 Can you think of one or two possible strategies for conversion?

2.7 PROGRAM DOCUMENTATION

A number of documents may be produced at this stage and various authorities have different views as to what they should be. Essential to the following maintenance phase are listings of the *source code* for the program modules and the *test log*, the latter being an account of the test procedures to which the modules have been subjected, and the results obtained. Also emerging from implementation may be updated versions of documents that originated earlier in the cycle. For instance, it may have been necessary to modify the user and systems manuals and issue new versions.

2.8 MAINTENANCE

In Chapter 1 we noted that software will continue to receive attention throughout its working life and in Exercise 1.2 we examined the reasons why this should be so. An interesting statistic is provided by a number of practitioners and researchers in the subject. They seem to be largely in agreement that maintenance, on average, accounts for about 70 per cent of the total life-cycle costs of a piece of software. In other words, more than twice as much is spent on changing it as on building it in the first place. It is not surprising, therefore, that in recent years considerable effort has been devoted to reducing the costs incurred in maintaining software. Much work has been done on improving documentation techniques and providing better *software tools* for the developers. Software tools are computer programs that are intended to improve the efficiency of analysts, designers and programmers. They include text editors, arguably the most important, and packages to assist with debugging, testing, and so on. However, it has been realized that one of the best ways to check excessive maintenance costs is to accept that maintenance will always be necessary, and design with this fact in mind. In