Modern Methods for COBOL programmers

J.R. Pugh and
D.H. Bell

Modern Methods for COBOL Programmers

John Pugh
School of Computer Science
Carleton University, Ottawa, Canada

and

Doug Bell
Department of Computer Studies
Sheffield City Polytechnic, England

Englewood Cliffs, New Jersey London New Delhi Rio de Janeiro
Singapore Sydney Tokyo Toronto Wellington

Library of Congress Cataloging in Publication Data

Pugh, John R., 1950–
Modern methods for COBOL programmers.
Includes bibliographies and index.
1. COBOL (computer program language).
I. Bell, Doug H., 1944– . II. Title.
QA76.73.C25P83 1983 001.64'24 82-22978
ISBN 0-13-595215-8

British Library Cataloguing in Publication Data

Pugh, John R.
Modern methods for COBOL programmers.
1. COBOL (computer programming language).
I. Title. II. Bell, Doug H.
001. 64'24 QA76.73.C25
ISBN 0-13-595215 8

ISBN 0-13-595215 8

Prentice-Hall International, Inc., *London*
Prentice-Hall of Australia Pty, Ltd., *Sydney*
Prentice-Hall Canada, Inc., *Toronto*
Prentice-Hall of India Private Ltd., *New Dehli*
Prentice-Hall of Japan, Inc., *Tokyo*
Prentice-Hall of Southeast Asia Pte., Ltd., *Singapore*
Prentice-Hall Inc., *Englewood Cliffs, New Jersey*
Prentice-Hall do Brasil Ltda., *Rio de Janeiro*
Whitehall Books Ltd., *Wellington, New Zealand*

Printed in the United States of America

10 9 8 7 6 5 4 3 2 1

PREFACE

The last decade has seen the production of software emerge as a major problem for the data processing industry. Less expensive but more powerful computer hardware has made feasible projects which would not have been contemplated only a few years ago. Many of these projects, however, will either not be implemented or be delayed due to the spiralling costs both of maintaining current software and of developing new software. Missed deadlines, cost overruns, shortages of qualified personnel, and badly designed systems are familiar problems in what has come to be termed the 'software crisis'.

The need for new tools and techniques in software development has been apparent for some years. Many organizations, groups and individual researchers have attacked the software problem, resulting in an overwhelming number of improved programming practices being suggested. Many of these practices are no longer in their infancy, they have been tried and tested on major projects from a wide variety of application areas. Some have proved very successful; but despite the demonstrated benefits of these new programming ideas they are still generally unknown or misunderstood.

This textbook is aimed at the practicing business application programmer. It discusses proven programming tools and techniques and shows how they can be applied to the development of commercial data processing systems. An objective is to present this material from the viewpoint of the applications programmer wishing to upgrade his or her knowledge of modern programming practices, although the text will also be of interest to systems analysts and managers. COBOL is used throughout this text as it remains the most extensively used business application programming language.

This book is not an introduction to programming in COBOL: it is aimed primarily at readers who have a working knowledge of COBOL but who may not have a thorough grounding in the fundamentals of program design. It is only in recent years that educational institutions have begun to emphasize this important topic. The text is also suitable for a second-level undergraduate course which provides an introduction to business data processing using COBOL. These students should be fluent in a high level language other than COBOL and have access to a COBOL language reference manual.

Several chapters are devoted to case studies, where solutions to typical data processing problems are developed using techniques introduced in earlier chapters. We intentionally do not present nicely packaged textbook solutions. It is important for the reader to fully understand how and why a particular solution developed and why other potential solutions were discarded.

Chapter 1 describes the problems involved in developing and maintaining software and presents an overview of the programming practices followed in dealing with these problems.

Early chapters concentrate on the development of an organized and disciplined approach to the difficult task of program design. Chapter 2 introduces the basic components of a design methodology which is referred to throughout the text as top-down stepwise refinement. This is closely aligned with, although not restricted by, the functional decomposition approach to program design. The text may thus be considered an alternative to texts concentrating on the data structure or data flow approaches. Chapter 3 applies the methodology to the design of a multi-level report program. Additional guidelines for successful program design are presented in Chapter 4 before the design of two classical data processing problems are considered in Chapters 5 and 6.

Chapter 7 describes a subset of COBOL which may be used to produce clear, maintainable and reliable COBOL programs. This chapter is chiefly aimed at programmers with little or no knowledge of COBOL. Chapter 8 describes how COBOL may be used to implement program designs developed in line with the methodology presented in earlier chapters. Further guidelines for programming in COBOL are given in Chapter 9; this chapter also includes the full COBOL texts for the case study designs developed earlier. The current standard version of COBOL, as adopted by the American National Standards Institute (ANSI) in 1974, is used throughout the text.

Chapter 10 describes a top-down approach to the implementation and testing of program systems. This technique is applied to the development of an on-line update program in Chapter 11.

The final two chapters deal with practices which are increasingly being introduced by managers to organize and control software projects. Chapter 12 discusses structured walkthroughs, a method of reviewing the progress of a project at various stages. Chapter 13 deals with the operation of programmer teams.

Exercises and further reading lists are provided at the end of many of the chapters. Some of the exercises raise problems which will be discussed later in the text.

We thank people at Carleton University and Sheffield City Polytechnic for their help, particularly Linda Guay, Mike Hollingsworth, Linda Latham, Mark Shackleton and Neil Willis. In preparing this book we used the facilities provided by the computer centers at Carleton University and Sheffield City Polytechnic: we appreciate the help given by people in those centers. Giles Wright of Prentice-Hall is thanked for his support and encouragement.

The ideas discussed in this book are the result of the efforts of many dedicated computer professionals and we would like to acknowledge their work.

CONTENTS

1

Introduction

1.1 OVERVIEW

The term "Software Crisis" is familiar to everyone involved in data processing today. In recent years the cost of computing power has plummeted dramatically. During the same period the cost of producing software has continued to rise at such an alarming rate that it is not uncommon for software costs to account for 75% or more of the costs of a new computer system. Software costs in the US are now measured in tens of billions of dollars each year. The rapid advances in hardware technology have significantly increased the scale of projects which are now being tackled and have tended to nullify any improvements gained from recent developments in software technology. These are very disturbing trends which unfortunately seem likely to continue for the foreseeable future.

Though progress has been slow, significant advances have been made in the development of tools and techniques to improve the process of producing software. These developments have covered all elements of the Software Life Cycle but in this text we concentrate on those areas which have most impact on the practicing business application programmer, namely, software design, coding and documentation, operation and maintenance, and the management of software projects. Despite the fact that the techniques discussed in this text are no longer in their infancy and that many proven benefits have accrued from their use, they have not been universally adopted and widespread ignorance and misunderstanding of the methods still exists.

Some confusion is easily understood as in recent years programmers have been overwhelmed by the sheer volume of literature and "jargon" associated with these improved programming practices. Though certainly

not familiar with the intimate details of what lies behind some of this technology, phrases such as structured programming, top-down design, stepwise refinement, program design language, HIPO charts, chief programmer teams, program development libraries, structured walkthroughs and many others will be very familiar to most programmers. Before presenting an overview of some of the available tools and techniques, it is important to discuss and fully appreciate the nature of the problems facing programming professionals today.

1.2 SOFTWARE DEVELOPMENT PROBLEMS

It is a measure of the depth of the software crisis that we can confidently predict that every practicing programmer has been involved in the development of a software system that has gone awry for one reason or another. It is probable that the system suffered from a combination of the following common maladies:

(a) Missed project deadlines, late system delivery
(b) High development costs, budgets exceeded
(c) User dissatisfaction, system performs inadequately or does not meet user requirements
(d) Error-prone and unreliable, requiring excessive corrective maintenance
(e) Overly-complex, difficult and expensive to perform adaptive maintenance.

Some of the major problems impacting software development are described in the remainder of this section.

Complexity

The fall in hardware costs and developments in hardware technology have made feasible increasingly ambitious projects and generated an almost insatiable demand for complex, sophisticated software. This trend will continue as we move from isolated stand-alone systems to integrated systems and as computing makes inroads into hitherto untapped application areas. The complex and intricate nature of these advanced software systems is often underestimated by project managers and results in wildly optimistic cost and time estimates. The design and implementation of these systems is carried out in an environment where costly time delays must be anticipated. Software systems evolve over time. Initial requirements specifications are often vague and incomplete, they need to be modified as the project proceeds. Additionally, the user may clarify, change, or add functional requirements. These modifications are all sources of delay and increased cost.

The design of complex software systems remains a fundamental problem despite the advances that have been made in software technology. There is no well-defined recipe which, if followed, guarantees a good design. Software design is an intellectually challenging and creative human task. Inventive, experienced designers are priceless assets of any programming shop. Whilst recognizing the importance of skilled individuals it has become increasingly evident that designers need helpful tools and techniques if they are to master the complexity of large scale application systems. When application systems were such that a small number of individuals might assume complete responsibility for the design and implementation of a system, software design tended to be a rather undisciplined process. Little distinction was made between design and programming. Programmers considered time spent on design as largely unproductive, believing that design and detailed coding were the same and could be performed concurrently. When applied to the design of large scale systems involving teams of programmers and hundreds of interacting program modules this *ad hoc* approach has proved disastrous. There are now, however, numerous useful tools and techniques which can assist programmers in the battle against complexity and which, when carefully selected and used in the right combination, can bring much needed discipline to the software development process.

Maintenance

Another daunting problem facing data processing installations is that each new application system put into operation immediately generates its own maintenance workload. In many installations 60 per cent or more of the workload is taken up with the maintenance of existing systems. This leaves fewer resources available for the development of new systems. New staff must be recruited or new developments delayed. Maintenance may be broadly classified into two types:

Adaptive maintenance

(a) to meet changes in the application environment, e.g. a change in taxation regulations
(b) to meet changes in the operation environment, e.g. the installation of a new compiler
(c) to satisfy requests from users for enhancements or modifications to a system.

Corrective maintenance

(a) to identify the cause of and correct bugs not discovered during system testing, i.e. repairing unreliable and non-robust software
(b) to improve a poorly performing system.

In many programming installations there has been a tendency to regard maintenance as a training ground for raw, inexperienced recruits freeing the more experienced programmers for more attractive, creative development work on new application systems. Maintenance has been seen as an activity requiring little skill. In fact, maintenance is a difficult task requiring high levels of skill, creativity and experience. What makes the task of the maintenace programmer so difficult?

The maintenance programmer must be able to quickly understand what the major functions of a program are and how these functions are accomplished. This is made more difficult because programs are poorly documented, poorly organized and the program code reflects the individual style and favorite programming "tricks" of the original program author.

Most programs requiring maintenance will already have undergone numerous modifications. Each modification makes the next one more difficult. The quality of a program's documentation, structure and reliability declines over the life of the program.

Designing a program so that it can be maintained easily is difficult. Unfortunately, ease of maintenance is often not a major consideration in the mind of a program designer. Consequently maintenance programmers find that programs do not accommodate changes easily. Modifications are not localized, they impact on seemingly unrelated parts of the program and cannot be implemented without major surgery on the structure of the program. Rather than carry out this time-consuming restructuring it is commonplace for the maintenance programmer to succumb to the temptation to implement the change with a "quick and dirty" fix or patch.

Management

The problems that beset software development are divided between those that are technical in nature and those that are caused by poor management. When software projects were small and project teams consisted of only a few people, informal methods of project planning and maintenance were adequate. For large complex projects, a formal integrated approach to management is necessary for successful project implementation. Some of the consequences of ineffective management are listed below:

(a) poor estimating of project schedules leading to missed deadlines and late system delivery
(b) poor cost estimation and management leading to budget overruns
(c) poor project visibility making it impossible to assess the progress of the project and identify schedule slippages and trouble spots
(d) ineffective project monitoring leading to a lack of adherence to

installation standards and poor discipline in using software tools and techniques

(e) poor personnel management and ineffective communication channels.

In this text we concentrate our discussion on how these modern project management techniques affect the work of the programming practitioner.

Education

The demand for new application systems, fuelled by the fall in hardware costs, increased hardware capability, and increased user requirements, has created an acute shortage of skilled personnel. Data processing installations find themselves caught in a software cost spiral. New application systems are more complex than existing systems and therefore require more software development time. Once implemented, new systems are added to the existing software needing to be maintained. Systems now more rapidly reach the stage where they become too expensive to maintain and need to be completely replaced: the consequences of this are that the supply of suitably trained personnel entering the industry is inadequate to meet the demand, and that intense competition exists for the skills of the more experienced and talented professionals. Skilled programmers are able to command good salaries and change jobs easily.

The problems for those who manage software development projects have been compounded by the fact that only marginal improvements in software productivity have been achieved in recent years. Although individual programmer productivity varies greatly, rates of fewer than ten lines of finished code per day are commonplace. There are many factors which influence productivity but the sheer size and complexity of today's application systems are among the most significant. Most software practitioners have received little or no training in how to deal with these problems. For many, their only course in programming will have emphasized mastering the intricacies of a particular programming language rather than the fundamentals of problem solving, program design and coding techniques. Even fewer will have received any formal education in such desirable areas as design methodologies, communication skills, or the ergonomics or human factors involved in designing computer systems. Prior to entering the workplace it is unlikely that they will have been involved in the design and implementation of a large software system or in working within a project team. Fortunately, today's computer science graduates are generally far better prepared to meet the challenges of the data processing industry than their predecessors. For those overworked practitioners already in the field the acquisition of modern software skills is a slow and difficult process. Often new ideas are misunderstood and the implementation of new practices meets with considerable resistance.

1.3 MODERN PROGRAMMING PRACTICES

In this section we present an overview of recent major developments in software technology.

Structured Programming

The first use of the term *structured programming* can be traced back to the work of Dijkstra in the mid-1960s. Since then it has become one of those terms which means all things to all men. Structured programming, as first enunciated by Dijkstra,[1] does not lend itself to a rigid precise definition although many attempts at such definitions exist in the literature.

The major principles of structured programming can be summarized as follows:

(a) A recognition that programming is a complex, intellectual activity and that we can no longer rely on idiosyncratic methods of program construction.

(b) The introduction of organization and discipline into the programming process to master this complexity and attain the goals of correct, reliable and maintainable programs.

(c) Program design is a distinct activity from coding and should be carried out in a systematic fashion using the following broad guidelines:

  (i) The initial refinement of a problem decomposes it into a number of highly abstracted subordinate problems. Each of these are then themselves refined in a similar way into a set of less abstracted problems. This process, known as successive or stepwise refinement, continues until the problem solution is described at a level where translation into the required programming language can be achieved easily. This method of program design is known as top-down design and results in a hierarchic or tree-structured solution.

  (ii) The transition between levels of refinement is kept as small as possible in order that each step can be understood easily and the correctness of the solution at each level can be informally verified.

  (iii) During program design and coding only three basic control structures, sequence, selection, and repetition are generally necessary.

Although they leave many practical implementation issues unanswered these principles capture the spirit of the early pioneers of structured programming.

The advent of structured programming was the cause of widespread controversy amongst the programming community. Much of the controversy

was the result of grossly oversimplifying the ideas being put forward and misunderstanding their true objectives. Two of the more common misconceptions are discussed below.

A widespread misconception was that structured programming could be equated to programming using only a restricted set of control structures and avoiding, at all costs, the use of the unconditional branch or **go to** statement. This narrow view of structured programming generated a heated debate as to whether the **go to** statement should be banished from high level programming languages and deflected attention from the main issue of how to design programs systematically.

Other programmers equated structured programming with modular programming. This technique arose as an alternative to the early common method of designing and coding programs in a monolithic fashion. Broadly stated, modular programming is the partitioning of a program into functionally independent modules. Provided the interface between modules is well defined, each module can be designed, coded, and tested independently, perhaps by different programmers. Few programmers argued with these ideas and saw structured programming as simply a restatement of the principles of modular programming. Today, modular programming is seen as a valuable component, but only a component, of structured programming. It lacks the discipline of structured programming in a number of areas, notably on how design is to be carried out within individual modules.

Structured programming inspired a resurgence of interest in the programming process. This has led to the development of a wide range of tools and techniques which have enabled the principles of structured programming to be put into practice in the workplace. In particular, guidelines have been proposed for the implementation of structured programs in COBOL, still by far the most predominantly used programming language for data processing applications. Also, a number of design methodologies have been developed and new project documentation and management techniques proposed. The term structured programming is now often used to refer collectively to the whole plethora of tools and techniques which have evolved from the initial ideas.

Design Methods and Principles

In recent years a number of approaches have been proposed to systematize the program design process. They are extensions and elaborations of Dijkstra's basic structured programming principles. Some approaches have been developed to the stage where they consist of an integrated collection of methods and principles and form the basis of a methodology for program design. The three design methodologies most used in the workplace are *Functional Decomposition*, *Data Flow Design* and *Data Structure Design*. Each methodology has its own strengths and weaknesses, and some are best

applied to particular types of problem or to particular application areas. No preferred, generally applicable methodology has emerged or can be expected to emerge in the near future.

Two principles, top-down design and modularity, underly most of the methodologies. *Top-down design* (Wirth[2], Mills[3]) starts from a problem specification and generates a hierarchic or tree structured design through the process known as successive or stepwise refinement as described earlier. Each level of refinement of the design corresponds to a particular level of understanding of *what* the program has to do, independently of *how* the result will be achieved at lower levels of refinement. The method advocates that initial levels should concentrate on critical broad design issues and that details should be postponed until lower levels. This is in contrast to bottom-up design, a technique often associated with early versions of modular programming. *Bottom-up design* again results in a hierarchic solution structure but in this case the lower level detailed modules are identified and refined first and subsequently used as a foundation on which to build the design. In practice, design is never solely top-down or bottom-up but rather a combination of the two. In this text we adopt the view of most of the popular methodologies that design should be predominantly top-down but it may be necessary at certain times to practice bottom-up design to examine the feasibility of some low-level module before design can proceed.

The decomposition of a problem into well-defined modules is a powerful tool in the fight against complexity. A "separation of concerns" can be achieved which allows each module to be understood independently, the impact of design modifications localized to as few modules as possible (preferably one), and modules to be developed independently. An important principle known as *data hiding* or *encapsulation* (Parnas[4]) can assist in realizing the benefits of modularity. It suggests that each module should be constructed so that the function it accomplishes and the interface information required to use it are clearly visible but that the internal code and data structures used may be hidden away inside the module in the sense that they need not be known to utilize the module. That is, *what* the module does is clearly visible but *how* the module accomplishes its function should not be. In particular, each data structure (or file structure) involved in a program system should have the structure itself, the statements that access it, and the statements that modify it, enclosed in a single module.

The major difference between methodologies can often be identified by examining the technique each uses to decide how refinements are to be made at each level of a design. Three major schools of thought have emerged. Functional Decomposition suggests that design decisions should focus on the operations which need to be performed to solve a problem. Data Flow Design suggests that an analysis of the flow of data is of paramount importance whilst Data Structure Design suggests that the program structure