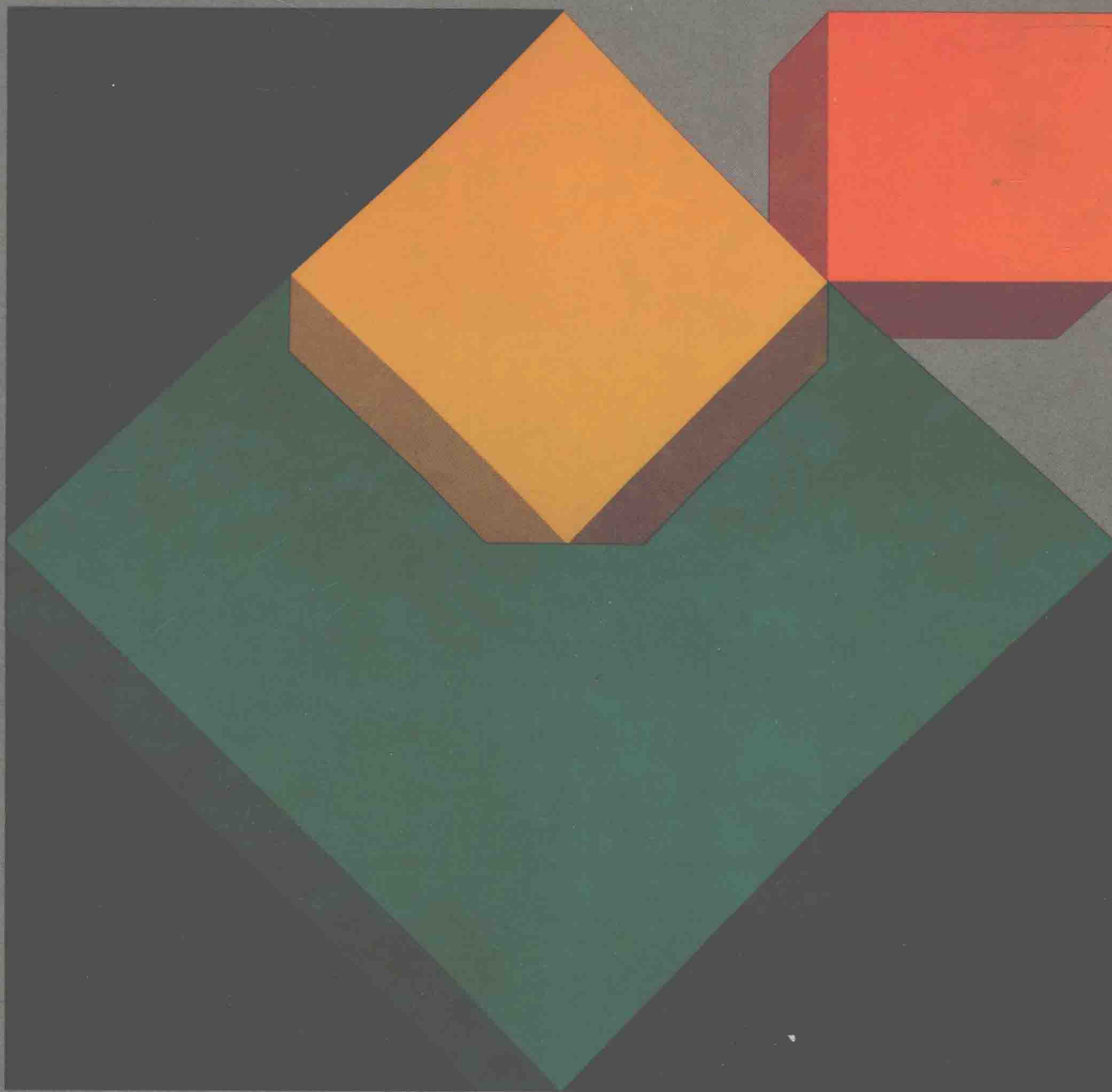
Daniel D. McCracken

A SECOND COURSE IN COMPUTER SCIENCE WITH PASCAL



A SECOND COURSE IN COMPUTER SCIENCE WITH PASCAL

Daniel D. McCracken
City College of New York

John Wiley & Sons
NEW YORK CHICHESTER BRISBANE TORONTO SINGAPORE

To Ruth, Harvey, David, Horace,
and the memory of John

Cover design by Rafael Hernandez

Copyright © 1987, by John Wiley & Sons, Inc.

All rights reserved. Published simultaneously in Canada.

Reproduction or translation of any part of
this work beyond that permitted by Sections
107 and 108 of the 1976 United States Copyright
Act without the permission of the copyright
owner is unlawful. Requests for permission
or further information should be addressed to
the Permissions Department, John Wiley & Sons.

Library of Congress Cataloging in Publication Data:

McCracken, Daniel D.
A second course in computer science with PASCAL.

Bibliography: p. ■■
Includes indexes.
1. PASCAL (Computer program language)
I. Title. II. Title: 2nd course in computer science with PASCAL.
QA76.73.P2M363 1987 005.13'3 86-32586
ISBN 0-471-81062-1

Printed in the United States of America

10 9 8 7 6 5 4 3 2 1

PREFACE

This is an introductory text on the central subject of computer science—the theory, implementation, and applications of data structures and algorithms.

Key features:

- There is a strong Abstract Data Type (ADT) approach. For each of the major data types the sequence is: a definition in terms of objects and operations, an illustration of usefulness, *then* one or more possible implementations, followed by larger applications and/or an overview of related issues.
- Almost all implementations are informally analyzed using the Big-Oh notation. The concept is introduced in the first chapter, and applied throughout.
- Recursion receives heavy emphasis. One chapter is devoted to its explanation and it offers many examples. Recursion is then used freely in the book, where it makes the presentation of other ideas clearer.
- There is heavy emphasis on program readability as a key feature of good programming style. The point is made explicitly in the first chapter and reinforced by example throughout. All programs have been tested and run, to produce the results shown and explained in the text.
- Program verification, through assertions and loop invariants, is introduced and explained at an informal level.
- The major illustrations are chosen with an eye to making the course serve, among other things, as an overview of what computer science is about. Parsing, simulation, expression simplification, BNF, state transition diagrams, backtracking, the database concept, solving recurrence relations, and several graph algorithms are introduced in this way, at a level appropriate to a second course.

It will be seen that the coverage is that recommended in the Koffman Committee report on a revision for CS2 of the ACM Curriculum 78.

I believe I have written a text that is in the spirit of the best recent work on curriculum revision; I have tried not to fall into the trap of trying to cover everything. The examples, along with their underlying abstract and implemented data structures, provide real-life applications of many theoretical topics. Thus the text complements a separate course in discrete mathematics but can be used separately.

About 15 percent of the entire text is devoted to a variety of exercises. Some test mastery of concepts; some are programming exercises that reinforce the data structures and algorithms covered in the chapter; some extend the mathematics and/or computer science topics of the chapter; and, some are suitable for projects that require consideration of software engineering issues. Many make most sense if programmed and run, but many others will help the student master the material

with only paper and pencil to support careful study. About a quarter of them involve discrete mathematics, permitting the instructor to give that emphasis if desired.

Every chapter concludes with a "Suggestions for Further Study" section, which is a selective annotated bibliography. The references chosen include other texts covering about the same material, for comparison and alternative exposition; primary sources; standard works that every student should know (e.g., Knuth); and indications of applications.

An Instructor's Manual will provide answers to exercises, suggestions for teaching the material, a sample syllabus, sample examinations, and ideas for larger projects for those who wish to emphasize the software engineering component of the course. A disk containing all programs and data from the book may be obtained by writing me care of John Wiley & Sons, Inc., 605 Third Avenue, New York, NY 10158.

Daniel D. McCracken
New York, New York
January, 1987

试读结束：需要全本请在线购买： www.ertongbook.com

ACKNOWLEDGMENTS

Many people rendered invaluable help in producing this work, but first on the list must be Doris C. Appleby (Marymount College, Tarrytown, New York). Her primary task was writing most of the exercises and the annotated bibliography, but as the work progressed she did much more. Her help in finding a meaningful presentation of the ADT concept to my chosen readership was especially fruitful. It is a pleasure to acknowledge her contributions.

Charles L. Baker (Science Applications, Inc.) gave me the benefit of reactions from someone who has been a practicing programmer since 1950.

Paul P. Clement (Advanced Systems, Inc.) made a great many helpful suggestions in the early stages of writing, especially on the use of graphics.

Charles Engelke (University of Florida) read every word of every draft, made innumerable suggestions (both large and small) for improvement, and saved me embarrassment by catching many slips in early drafts.

Gene Fisch (New York State Institute for Basic Research in Developmental Disabilities) taught from the manuscript at City College of New York and gave much help and support.

Henry Ledgard (Amherst, Massachusetts) provided a burst of good advice at a very early stage when it was most helpful, and provided a model of programming style that I recommend.

Haim Kilov (Latvian Light Industry Research Institute) also read every word of every draft. He provided reams of detailed suggestions for improved programming and better presentation and never ceased urging more mathematical rigor. I look forward to the day when students in a course at this level will benefit by the advanced approach he would have preferred. I also hope some day to meet him.

James J. Piccarello (formerly City College of New York and now AT&T Information Systems) taught from the manuscript and offered dozens of helpful suggestions of a wide variety of types.

William I. Salmon (University of Utah), my coauthor on other books, solved a troublesome problem in one program at a critical point, and was a supportive colleague in ways too numerous to detail.

Jeffrey R. Sampson (University of Alberta), before his untimely death in 1985, had begun to work on the project and surely would have been at least as helpful as he was on a previous book. He is missed.

Laurie White (University of Florida) taught twice from the manuscript, had many helpful ideas, and passed along the highly useful reactions of her students.

Gregory P. Williams (Woodbury, Connecticut), an old friend from our General Electric days, could be depended on to call within days of receiving anything, with an earful of sharply focused advice. It has been a real pleasure working with him.

My thanks also to Stefan Burr (City College of New York) and David Gries (Cornell University) for many detailed suggestions on Chapter 8.

It is a pleasure to acknowledge the many helpful ideas of the following people who served as reviewers through Wiley. Anonymous to me at the time, they provided both encouragement and valuable criticism. Their inputs made a real difference.

Maria Balogh (Portland State University), George Beekman (Oregon State University), Christopher M. Brown (University of Rochester), Henry A. Etlinger (Rochester Institute of Technology), Arthur C. Fleck (University of Iowa), Gary A. Ford (Carnegie-Mellon University), Raymond Ford (University of Iowa), Paul Gormley (Villanova University), Marc Graham (Georgia Institute of Technology), Adam O. Hausknecht (Southeastern Massachusetts University), Brian L. Johnson (University of New Hampshire), Michael P. Johnson (Oregon State University), John L. Lowther (Michigan Technological University), Louise Moser (California State University at Hayward), Richard E. Pattis (University of Washington), and Justin Smith (Drexel University).

Helen Blumenthal, who is my wife, handled the preparation of the manuscript for submission in magnetic form and many other production matters.

The editorial and production people at Wiley were capable and supportive as they always are, and I am grateful for their efforts.

Finally, it is a pleasure to convey my deep appreciation to my students at City College of New York. They were patient and supportive as, at times, we all learned together. The book would not have been possible without them.

D.D.M.

CONTENTS

1 THE PROGRAM DEVELOPMENT PROCESS 1

1.1 TO THE READER: ABOUT THIS BOOK 1

1.2 THE SOFTWARE DEVELOPMENT CYCLE 2

1.3 THE CHARACTERISTICS OF A GOOD PROGRAM 6

1.4 PROGRAMMING STYLE 7

1.5 AN INTRODUCTION TO ANALYSIS OF ALGORITHMS 10

EXERCISES 14

SUGGESTIONS FOR FURTHER STUDY 17

2 ARRAYS AND RECORDS 19

2.1 INTRODUCTION: THE ABSTRACT DATA TYPE (ADT) APPROACH 19

2.2 THE ADT ARRAY 20

2.3 STORAGE ALLOCATION FOR SEQUENTIAL REPRESENTATION OF ARRAYS 22

2.4 NON-RECTANGULAR MATRICES 26

2.5 THE ADT APPROACH TO ARRAYS AGAIN 28

2.6 RECORDS 29

2.7 APPLICATION: LETTER-SEQUENCE FREQUENCIES AND A PERFORMANCE PROBLEM 32

2.8 THE USER REQUIREMENTS STATEMENT 32

2.9 THE FUNCTIONAL SPECIFICATION 34

2.10 THE PROGRAM DESIGN 34

2.11 THE IMPLEMENTATION: A PROGRAM 36

2.12 TESTING 43

2.13 AN ATTEMPT TO RESCUE PERFORMANCE 44

2.14 INFORMAL ANALYSIS OF SELECTION SORTING 47

2.15 SOLVING THE PERFORMANCE PROBLEM 48

2.16 CONCLUSION: THE LESSON OF THIS EXAMPLE 52

EXERCISES 54

SUGGESTIONS FOR FURTHER STUDY 58

3 SETS AND STRINGS 61

3.1 INTRODUCTION 61

3.2 THE ADT SET 61

3.3 AN IMPLEMENTATION OF SETS USING ARRAYS 64

3.4 ANALYSIS OF THE ARRAY IMPLEMENTATION OF SETS 68
3.5 THE PASCAL IMPLEMENTATION OF SETS 68
3.6 ANALYSIS OF THE BIT-VECTOR IMPLEMENTATION OF SETS 70
3.7 THE ADT STRING 71
3.8 TWO ILLUSTRATIONS OF THE USE OF THE ADT STRING 73
3.9 DESIGN CONSIDERATIONS IN IMPLEMENTING THE ADT STRING 78
3.10 IMPLEMENTATION OF A PACKAGE OF PROCEDURES FOR THE ADT STRING 80
3.11 ANALYSIS OF THE ARRAY IMPLEMENTATION OF STRINGS 90
3.12 CONCLUSION 90
EXERCISES 91
SUGGESTIONS FOR FURTHER STUDY 94

4 STACKS AND QUEUES 95
4.1 INTRODUCTION 95
4.2 THE ADT STACK 95
4.3 A FIRST STACK EXAMPLE: TESTING FOR PALINDROMES 96
4.4 AN ARRAY IMPLEMENTATION OF STACKS 104
4.5 A SECOND STACK EXAMPLE: EVALUATING AN EXPRESSION IN POSTFIX FORM 107
4.6 A FINAL STACK EXAMPLE: CONVERTING INFIX EXPRESSIONS TO POSTFIX 111
4.7 THE ADT QUEUE 120
4.8 AN EXAMPLE: THE PALINDROME PROBLEM AGAIN 121
4.9 AN ARRAY IMPLEMENTATION OF QUEUES 123
4.10 AN APPLICATION: QUEUES IN A WAITING LINE SIMULATION 128
4.11 CONCLUSION 132
EXERCISES 133
SUGGESTIONS FOR FURTHER STUDY 137

5 RECURSION 139
5.1 INTRODUCITON 139
5.2 A FIRST EXAMPLE: PRINTING A NUMBER IN REVERSE 140
5.3 HOW RECURSION IS IMPLEMENTED 142
5.4 A RECURSIVE FUNCTION: THE FACTORIAL 145
5.5 A HIGHEST COMMON FACTOR FUNCTION 146
5.6 RECURSIVE ADDITION FUNCTIONS 147
5.7 THREE VERSIONS OF A RECURSIVE FIBONACCI NUMBER FUNCTION 147
5.8 AN APPLICATION: COUNTING CELLS IN A BLOB 150
5.9 BACKTRACKING VIA RECURSION: THE EIGHT QUEENS PROBLEM 152

5.10 RECURSION REMOVAL 158

5.11 APPLICATION: RECURSION IN CONVERTING AN EXPRESSION FROM INFIX TO POSTFIX 162

5.12 CONCLUSION 170

EXERCISES 170

SUGGESTIONS FOR FURTHER STUDY 173

6 LINKED LISTS 175

6.1 INTRODUCTION 175

6.2 THE ADT LINKED LIST 176

6.3 A NOTE ON TERMINOLOGY 177

6.4 LINKED LIST IMPLEMENTATIONS OF STACKS AND QUEUES 178

6.5 INFINITE PRECISION ARITHMETIC WITH LINKED LISTS 179

6.6 A POINTER VARIABLE IMPLEMENTATION OF LINKED LISTS 183

6.7 ANALYSIS OF LINKED LIST OPERATIONS 194

6.8 AN ARRAY IMPLEMENTATION OF LINKED LISTS 195

6.9 STRINGS IMPLEMENTED AS LINKED LISTS 202

6.10 SETS IMPLEMENTED AS ORDERED LINKED LISTS 205

6.11 OTHER FORMS OF LINKED LISTS 208

6.12 CONCLUSION 54

EXERCISES 210

SUGGESTIONS FOR FURTHER STUDY 216

7 TREES 217

7.1 INTRODUCTION 217

7.2 BINARY TREES: GENERAL CONCEPTS 218

7.3 TRAVERSAL OF BINARY TREES 220

7.4 THE ADT BINARY SEARCH TREE 222

7.5 THE ADT BINARY SEARCH TREE IN A RUDIMENTARY DATABASE APPLICATION 224

7.6 A POINTER VARIABLE IMPLEMENTATION OF BINARY TREES 228

7.7 EXPRESSION TREES 237

7.8 DETECTING IDENTICAL SUBTREES IN AN EXPRESSION TREE 244

7.9 THREADED TREES 247

7.10 A SKETCH OF GENERAL TREES 252

7.11 CONCLUSION 254

EXERCISES 254

SUGGESTIONS FOR FURTHER STUDY 264

8 SORTING 265

8.1 INTRODUCTION 265
8.2 A NOTE ON TERMINOLOGY 267
8.3 SELECTION SORT 267
8.4 INFORMAL PROGRAM VERIFICATION OF SELECTION SORT 269
8.5 SORTING COMPLETE RECORDS 273
8.6 INSERTION SORT 276
8.7 ADDRESS TABLE SORTING 279
8.8 SHELL SORT 281
8.9 THE DIVIDE AND CONQUER STRATEGY 285
8.10 QUICKSORT 287
8.11 HEAPSORT 295
8.12 BUCKET SORT 308
8.13 COMPARING THE METHODS 310
8.14 CONCLUSION 311
EXERCISES 312
SUGGESTIONS FOR FURTHER STUDY 320

9 SEARCHING 323

9.1 INTRODUCTION 323
9.2 SEARCHING IN ADT TERMS 325
9.3 SEQUENTIAL APPROACHES 326
9.4 BINARY SEARCH 327
9.5 HASHING 330
9.6 LINEAR PROBING HASHING 332
9.7 QUADRATIC HASHING 335
9.8 CHAINED HASHING 337
9.9 AN OVERVIEW OF AVL TREES 339
9.10 COMPARISON OF THE METHODS 343
9.11 AN OVERVIEW OF THE ISSUES IN EXTERNAL SEARCHING 344
9.12 APPLICATION: LETTER-SEQUENCE FREQUENCIES REVISITED 348
9.13 CONCLUSION 356
EXERCISES 357
SUGGESTIONS FOR FURTHER STUDY 364

10 GRAPHS 365

10.1 INTRODUCTION 365
10.2 TERMINOLOGY AND NOTATION 366
10.3 THE COMPUTER REPRESENTATION OF GRAPHS 367

试读结束：需要全本请在线购买： www.ertongbook.com

10.4 GRAPH SEARCH STRATEGIES: DEPTH-FIRST AND BREADTH-FIRST 369
10.5 A DEPTH-FIRST SEARCH ALGORITHM 370
10.6 A BREADTH-FIRST SEARCH ALGORITHM 375
10.7 AN APPLICATION: SEARCHING A MAZE 381
10.8 SHORTEST PATH IN A DIRECTED GRAPH: DIJKSTRA'S ALGORITHM 382
10.9 THE MINIMUM COST SPANNING TREE: KRUSKAL'S ALGORITHM 389
10.10 CONCLUSION 399
EXERCISES 399
SUGGESTIONS FOR FURTHER STUDY 403

BIBLIOGRAPHY 405

INDEX OF FUNCTIONS, PROCEDURES, AND PROGRAMS I1

INDEX OF SUBJECTS AND AUTHORS I4

CHAPTER 1

THE PROGRAM DEVELOPMENT PROCESS

1.1 TO THE READER: ABOUT THIS BOOK

We begin this text with the assumption that you have had one computer science course on algorithm development, with programming in Pascal. You may, we hope, have had a course in discrete mathematics or you may know other programming languages. You may also know something about how computers operate at a level below programming languages like Pascal. But that you have had one good Pascal-based computer science course in which you wrote five to ten programs and ran them—that is our starting assumption.

Building on that background, study of this book will advance your growth in computer science in four areas:

1. **Abstract data types (ADTs).** You have been introduced to arrays, records, sets, and strings in your previous study. We shall look at those topics afresh, taking a broader viewpoint, and study five others: stacks, queues, linked lists, trees, and graphs. Our major new viewpoint will be to distinguish between the *concept* of a given abstract data type and the various possibilities for its *implementation*. We call this the *abstract data type* (ADT) approach.
2. **Algorithms.** An algorithm is a precisely stated set of steps that terminates in finite time, the execution of which solves a stated problem. You are probably already accustomed to simple algorithms, perhaps when you had to evaluate a polynomial, find the roots of a polynomial, solve a small system of simultaneous equations, or find the greatest common divisor of two integers.

In this book we shall study a number of algorithms for solving some of the "standard" problems that arise frequently in computer science. You have probably seen simple examples of some of these, such as elementary sorting methods or binary search. Here we shall not only investigate sorting and searching in greater depth, but also study a variety of algorithms that arise in connection with the new (to you) ADTs. These involve, for example, ways of "visiting" all the nodes of a binary tree or finding the shortest path between two vertices in a graph. Several of the algorithms will have the dual purpose of illustrating a new ADT and introducing a fundamental area of computer science study.

3. **Better programming.** Computer science is more than the amassing of programming skills and languages, but programming is nonetheless one of the essential tools of a computer scientist. Consequently, in this book we will help you improve your programming skills in a number of ways. One major way is to give you many examples to read and study.
4. **The use of mathematics in computer science.** Computer science as a discipline is in the process of becoming more firmly grounded in mathematics. In various ways, appropriate to your background, we shall demonstrate some of the ways mathematics is useful in computer science and deepen your ability to work with the combination of the two.

In other words, you are going to learn some of the tools of the trade of a computer scientist. Many of these tools have direct application; others are building blocks that will find full value after study of more advanced topics such as compiler construction, database implementation, or computer graphics. A number of the applications that we use to illustrate data structures and algorithms provide "sneak preview" of some of the areas of computer science that you will be studying later.

1.2 THE SOFTWARE DEVELOPMENT CYCLE

In writing programs to this stage, you have probably gone through a process something like this:

1. Get the assignment. The instructor states a problem to be solved.
2. Devise an algorithm. With more or less help depending on your experience, and with more or less difficulty depending on the problem, devise an algorithm for solving the problem.
3. Express the algorithm as a computer program, using the programming language specified for the course. Type the program into the computer.
4. Compile the program. Revise it to correct errors detected by the compiler, run it with sample data for which you know the correct answers, and correct errors discovered in testing. Run the program with actual data and turn in program and results.

And, typically, *never so much as look at the program, ever again.*

This is an acceptable way to start learning about programming, computer science, and software engineering. However, software development outside of the classroom is vastly different. Many organizations develop programs that are far too large to be written by one person or even by one small team. In projects of such size, a majority of the effort may be devoted primarily to communication among team members. Any one team member may be working on a procedure of only a few hundred lines, but that procedure must *interface* (communicate correctly) with other procedures in the program. As a result, interface standards and programming style standards are rigidly enforced. When all the costs have been tallied, including the planning, the waste caused by bad planning, and the meetings to coordinate interface definitions, etc., the cost of a program can easily be in excess of $100 *per line.*

If you have ever written a 200-line program by staying up the night before an assignment was due, you may find this figure incomprehensible. That is because you have seen only a part of the whole job of program development so far.

You are not expected to make the transition from writing "student-sized" programs to the industrial world of multi-person projects in one giant leap. For the purposes of this book, you will be writing programs of a few hundred lines, for the most part, and working alone. But one major goal of the book is to help you make several big steps—if not the entire journey—toward your ability to develop software of realistic size.

One of the essentials you will have to acquire is how to become much more systematic about the program development *process.* It won't do, for instance, to approach program development in an informal way. Rather, you will have to start going through a series of well-defined steps, using the intellectual tools that are the main subject of the book, in which the coding is only one phase.

There is no single formula that fits every situation, but we can list a sequence of steps that will provide a starting outlook—a framework that will serve in many cases—and that is a good foundation for discussion of the variations.

Here, then, is a listing of the steps in developing a software system.

1. **User requirements analysis.** The software developer or development team, working closely with the *user* (also called the *client* or *customer*—the person or group requesting the software), produces a definition of the problem. A document called the *user requirements statement* is jointly agreed to by the user and the developer, and becomes a sort of contract stating what the program is to accomplish. However, this document contains essentially nothing about how the requirements are to be met, which is worked out later. "What, not how" summarizes this step.
2. **Functional specification.** Next comes a technical statement that shows the major components of the system, data flows between them, required outputs, errors to be checked for and procedures to follow after detection, and any constraints such as required processing rates. A formal document is also produced at this stage, part of which is a first draft of the *User's Guide,* which is what the end users will work from when using the system.
3. **Design.** For each of the major components identified in the functional specification, the developer now chooses data types and algorithms, and breaks

the processing into precisely defined subprograms (procedures or functions, in Pascal). Key algorithms may be subjected to *verification* at this stage, to build confidence that they are correct.

4. **Implementation.** The design is translated into code, usually in a high-level language such as Pascal.
5. **Testing.** The developer then compiles the code and corrects errors detected by the compiler. Next, the compiled programs are run with data for which the correct results are known; errors detected in this stage are also corrected. The programs are then run with data containing all the errors that the requirements ask to be detected. (The programs may also be run with completely random data, to see if they reject unanticipated bad data.) The programs are finally run with real data supplied by the client, which will often disclose errors of understanding of user requirements.
6. **Installation.** The software is placed on the *target computer* (the client's machine, on which the application will run in regular use), which is usually not the one on which development (coding and testing) is done. The client personnel who will operate the system are trained. The system is run in parallel with the one it replaces, if that is the situation. (Discontinuing an old system before its replacement has been thoroughly proven has long been recognized as a recipe for disaster, but it still happens.)
7. **Maintenance.** This is an umbrella term for everything that is done to the software after the user has accepted the initial product—correction of errors not detected earlier, addition of new features, and modifications necessitated by hardware changes. "Maintenance" is a decidedly awkward term for this collection of almost unrelated activities, none of which is similar to the "maintenance" of a car or a TV set, but the terminology is entrenched.

The amount of time devoted to each of these steps obviously varies, but the following percentages of the total effort are generally indicative of what is needed:

User requirements: 10%

Functional specification: 30%

Design: 20%

Implementation: 15%

Testing: 15%

Installation: 10%.

No percentage has been shown for maintenance, which requires further explanation and emphasis. The trouble with trying to pin down a time allotment is that maintenance itself is poorly defined. Pragmatically, it often turns out to cover everything from the time the user accepts the system until an agreement is reached to undertake a major revision. By this definition, maintenance is usually reported as something in the range of 50–90% of the total cost of software development. Astonishing, perhaps, but generally accepted.

Several lessons may be drawn from this discussion.

1. Coding, possibly the only part to which you have given serious attention so far, is a relatively minor part of the job. One reason for this is that real life applications often require a great deal of effort just to get a good definition of what is to be done, in part because the user often doesn't really know at the outset. The user can't write the requirements in isolation, without knowing the constraints imposed by the computer, a subject about which he or she is not expert. Likewise, the software developer obviously can't define the task in isolation. Furthermore, the two people (or two groups) often have quite different vocabularies and areas of expertise, requiring considerable time, effort, and good will just to be sure they are really talking about the same thing. The draft of the User's Guide, which should be written long before coding begins, is an excellent device to help detect misunderstandings early in the development process.
2. Just about anything that can flush out errors of whatever type, *early in the development process,* is beneficial. The earlier an error is caught, the easier and cheaper it is to correct, sometimes even by a huge margin; correcting an error after installation may cost tens or even thousands of times as much as correcting it at the design stage. This is one reason that the percentages of project effort devoted to requirements analysis and functional specification are so much larger than you might have expected: much wasted effort can be saved by getting them right the first time. In sum, delay coding as long as possible.
3. We described this process as a linear sequence of separate steps, but it is really a *cycle:* normally there will be considerable looping back to repeat portions of earlier steps.
4. Because many people are involved in development over periods of months or years, with a great deal depending on an accurate understanding of what is to be accomplished and what has already been done, documentation is a crucial part of the software development process. Documentation does *not* mean a rushed, after-the-fact job that is done carelessly to meet contract requirements. Most of the stages of the development process haven't really been completed at all until an appropriate document has been created. The ability to communicate easily and clearly is a crucial vocational skill for a software developer.

Because so many people of diverse backgrounds and concerns use the computer for so many different applications, it is to be expected that there will be many variations of this process. In fact, some developers feel that there is *so much* variability that it is better not to pretend that there is a "standard" "life cycle" at all. Such an objection will be viewed with considerable sympathy in these quarters,[1] but you need to have an idea of the traditional approaches before you can deal with the variations. Consequently, we have presented one formulation of the conventional life cycle.

[1]See Daniel D. McCracken and Michael A. Jackson, "Life-Cycle Concept Considered Harmful," *Software Engineering Notes,* **7**, 2 (April 1982).

试读结束：需要全本请在线购买： www.ertongbook.com